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, 15]. 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 [9]. 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 [13]. 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

One might not expect a DSL to prioritize generation of human-readable, idiomatic, or documented code. After all, the main purpose of a DSL is to allow the user of the DSL — who is probably a domain expert but not a programming expert — to avoid having to write GPL code, so why would they want to look at the generated GPL code? For programs written in most DSLs, the generated code is not itself the goal. The goal is to do something, and the form of the generated code is unimportant, as long as running it achieves that goal. In this respect, GOOL differs from most DSLs. Since the domain of GOOL is OO programming, the user of GOOL is expected to have some programming expertise. In a sense, GOOL is both a GPL and a DSL. GOOL's main purpose, rather than allowing the user to not have to write GPL code, is to allow the user to not have to write GPL code in multiple languages. As with other DSLs, completing some required task will certainly be a goal of a program written in GOOL. But since GOOL is general-purpose, we cannot assume that the generated code will not be an additional goal unto itself. For example, if it is to be integrated into an existing system, other programmers working on the system may need to look at and understand the code and how it impacts their own tasks. Or perhaps the generated code is a library, intended to be used by any project who sees a need for it.

In light of cases like these, the importance of our requirements for GOOL to generate human-readable, idiomatic, and documented code should be clear. These all contribute to understandability of the generated code. Documentation can explain code such that those using a program or library can do so quickly, without even having to look at the code. Documentation may take the form of comments added ad-hoc throughout the code, or more formal, structured comments that can be compiled into PDF or HTML documents using a tool like Doxygen. Documentation cannot be guaranteed to provide all of the information a user of the generated code may need, however, which is why generating human-readable code is still important.

Details not revealed by the documentation should be discernible by examining and understanding the code. Readable code is also important when it comes to debugging. Being unable to understand parts of a system would make diagnosing bugs very difficult. Pretty-printing and reducing complexity are effective ways to enhance code readability [6]. Some DSLs complicate generated code by generating it into a custom framework, but GOOL must avoid this to retain readability.

It should be noted that our requirements for the generated code to be documented and readable only hold if the code written in GOOL is documented and readable. Inevitably, a user will be able to write undocumented or overly complex code if they choose to do so. Our requirement is that GOOL at least offers a way to document the code, does not strip away code comments, pretty-prints the generated code, and does not introduce more complexity than required to meet the GOOL specification.

The importance of generating idiomatic code is perhaps not as obvious as readable and documented code. Consider a project written entirely in one language. If it uses generated code from GOOL and the code is non-idiomatic, it will be inconsistent with the rest of the code base and less understandable, both because it would be more complicated than it needs to be and because it would not be expressed naturally in the target language, not using the idioms with which the developers would be most familiar.

Our last requirement is altogether different from the others, but is typical of DSLs: GOOL should abstract over common OO patterns, allowing them to be expressed naturally. The patterns in question could abstract over small-scale idioms or could be larger-scale OO design patterns. While the motivation for this requirement is partly to make it even more efficient to write code in GOOL than in any of its target languages, there is more to it than that. This requirement is also necessary to make GOOL truly language-agnostic and to generate idiomatic code. This fact will be demonstrated with the examples of patterns shown in Section 5, but to understand why abstractions over patterns are needed, consider Python's ability to return multiple values with a single return statement. Other OO languages do not allow this, so either GOOL could also disallow it or GOOL could provide syntax for it but throw an error if it was used when targeting one of the languages that does not support it. In the first case, if a user of GOOL wanted to return multiple values and target Python, they would have to work around

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

the lack of a multiple-value return statement and the generated code would therefore be non-idiomatic. But in the second case, if the user wanted to target both Python and a language that does not support multiple-value return statements, the GOOL code would have to be different when targeting Python than when targeting the other language, so GOOL would not be truly language-agnostic. The higher-level pattern that would require a multiple-value return statement is a function with multiple outputs. If GOOL abstracted over that pattern and provided syntax for defining multiple-output functions, the code generator can generate a multiple-value return statement in Python and something different in the other language. In this way, the same GOOL code can truly be compiled into idiomatic code in any target language.

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.

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 `bodyStatements` 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 `param` 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
  [param num1, param 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

GOOL is embedded in Haskell in the finally-tagless style originally described in [7]. In finally-tagless style, the internal representation of an embedded DSL is a group of functions defined in the host language, rather than the more traditional use of data constructors to build an interpretable abstract syntax tree (AST) for the DSL. Whereas an AST of data constructors must have a separately-defined interpreter, the functions themselves in finally-tagless compose to form the interpreter; the definitions of the functions describe how each is interpreted. Thus, the GOOL "keywords" referred to in the previous section were really Haskell functions that, when resolved, yield representations of the code to be generated. The finally-tagless style grants us more flexibility in the code we can generate and better extensibility since we do not need to pattern-match on AST nodes.

Finally-tagless facilitates development of a family of interpreters for a DSL by having the functions act on representations, where a type class in Haskell is used to abstract over the representation. The authors of [7] coined the term "symantic" to describe such a type class, because the interface of the type class defines the syntax of the DSL and the instances of the type class define the semantics. A separate type class instance is written for each representation, each corresponding to one member in the family of interpreters. This suits GOOL's needs nicely, for GOOL requires multiple interpreters (one for each target language) of the same language (GOOL).

GOOL is a collection of these "symantic" type classes and instances. For organizational purposes, the functions comprising GOOL's syntax are split across many type classes, roughly based on the internal types upon which the functions act. GOOL's internal types correspond to the types of code being represented, not the types of the values in the target languages, though adding value-types to GOOL's type system is planned for the future, discussed further in Section 7. Examples of internal types in GOOL, then, are `Variable`, `Value`, `Type`, `Scope`, `Statement`, `Method`, `Class`, `Module`, and `Package`. A `Statement` in GOOL, for example, is a representation of a piece of code that is a statement. Below is an excerpt of GOOL's definition for the `VariableSym` type class; that is, the type class containing functions for defining variables in GOOL.

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

Since variables have types, a type for representing variables must also know how to represent types, thus we constrain our `repr` type variable to be an instance of `TypeSym`, the symantic type class containing functions for representing types, like the `int`, `float`, `string`, and `listType` functions described in Section 3.

The types used in the signatures of GOOL's functions have the general form `repr (X repr)` for some type `X`, examples of which can be seen in the type signature for `var`. To understand these types, first remember that we will write different instances of this type class for each different language GOOL targets. That means the `repr` type variable will be instantiated with a type that represents code in a specific target language. In the `repr (X repr)` types, the type variable `repr` appears twice because there are two layers of abstraction: abstraction over the target language, handled by the outer `repr`, and abstraction over the underlying types to which GOOL's types map, handled by the inner `repr`.

This abstraction over the underlying types that GOOL's types represent happens because `X repr` is a type family. The first line of the body of the `VariableSym` type class is `type Variable repr`. This line declares the type family `Variable repr`. Since the type is parameterized by `repr`, each instance we write of this type class can define the `Variable` family member differently. Usually different renderers will use the same underlying type for a given GOOL type, but the ability to change it on a per-target-language basis is useful for when

a language requires storing more or less information than others.

The type signature for the `var` function says that `var` takes a label and a representation of a type and yields a representation of a variable. This should not be surprising given how we used `var` to define variables like `var "foo" int` in Section 3. `Label` is simply a Haskell type synonym for `String`.

Shown below is the instance of the type class excerpt shown above for generating Java code.

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

We have instantiated `repr` as `JavaCode`, which is a monad defined as:

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

This may look like we are tagging our values with `JC`, in contradiction of the finally-tagless style, but Haskell's compiler does not actually treat types defined with `newtype` as tags, so this is valid. `unJC` extracts the underlying value from a value wrapped in `JavaCode`. Calling `unJC` on a program written in GOOL allows Haskell to infer that the `repr` in the GOOL program should be concretized as `JavaCode` and to resolve the functions using the definitions from the `JavaCode` instances of the type classes, thereby generating Java code.

The second line of the instance states that the underlying type for `Variables` in Java is `VarData`, which is a record that holds data about the variable to be used later. The definition of `VarData` is shown below.

```
data VarData = VarD {
  varBind :: Binding,
  varName :: String,
  varType :: TypeData,
  varDoc :: Doc}
```

Stored in the `VarData` structure is a variable's binding, either `Static` or `Dynamic`, the name of the variable as a `String`, the type as a `TypeData` structure, which is the underlying type for `Types` in GOOL for Java, and how the variable should appear in the generated code, represented as a `Doc`. `Doc` comes from the `Text.PrettyPrint.HughesPJ` Haskell package and represents formatted text. We use it to pretty-print our generated code, making it more readable. In common between all of GOOL's underlying data structures is that they each contain a `Doc`. For some types, like a `Block`, the underlying type is in fact nothing more than a `Doc`. The `Package`-level `Docs` are ultimately what will be printed to files to generate code. Keeping in mind that a `VarData` structure will usually be wrapped in a `repr`, such as `JavaCode`, the pieces of the wrapped `VarData` can be accessed in one of two ways. The first way is to use Haskell's `fmap` in combination with the built-in accessor functions for `VarData`. For example, `fmap varDoc` will take a `VarData` and return a `JavaCode Doc`. Functions on `Docs` can then be lifted to work on `JavaCode Docs` using Haskell's `liftA` family of functions. This method only works if the `Variable repr` is known to be equivalent to `VarData`, though. That is, it works in the context of `JavaCode` but not in a generic `repr` context. For when one needs to access a piece of `VarData` while in a generic `repr` context, GOOL offers its own "symantic" accessor functions. For example, `variableDoc` is part of the `VariableSym` type class, with signature:

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

and definition for the `JavaCode` instance:

```
variableDoc = varDoc . unJC
```

Since `variableDoc` is part of the type class, it can be used in a generic `repr` context, and since its definition is in an instance of the type class where `Variable repr` is known to resolve to `VarData`, it can be defined simply by calling the `varDoc` accessor on the unwrapped `VarData`. GOOL offers many functions similar to `variableDoc`, for accessing the other pieces of `VarData` and for accessing the pieces of the underlying data structures for other internal GOOL types.

The underlying data types for different GOOL types each store specific information related to the type in question. Some examples are that the underlying type for `Statements` stores a `Terminator` which determines whether the statement should end in a semi-colon, the underlying type for `Methods` stores a Boolean for whether it is the main method, and the underlying types for `Values`, `UnaryOps`, and `BinaryOps` store precedence information so expressions can be printed without superfluous parentheses, leading to more readable code.

The third line of the `VariableSym JavaCode` instance defines the `var` function by dispatching to the `varD` function, defined as:

```
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` constructs the variable using a function called `varFromData`, which is a GOOL function for constructing a variable in the generic `repr` context by explicity passing the pieces needed to create a `varData`, as can be seen by its type signature:

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

This function resides in a GOOL type class called `InternalVariable`. GOOL has a number of these "internal" type classes, none of which are exported as part of GOOL's user-facing interface. They contain functions that are useful for GOOL's various language renderers, but not to users. A user would not want to define a `Variable` by explicitly passing in the `Doc`, for example, which is why `varFromData` is not exposed. The `Doc` representation for the generated code is entirely internal to GOOL, not exposed in any of GOOL's user-facing functions. Examples of other GOOL functions that are only available internally are `printSt` for generating a low-level print statement, because it is superseded by higher-level functions for printing, and `cast` for casting between types, because any required type-casting is handled by higher-level functions.

In the definition for `varD`, the `Doc` for the variable is constructed simply by passing the variable name to `text`, which is a function from the `HughesPJ` package for converting a `String` to a `Doc` of that `String`. This should make sense because you can refer to a variable in Java simply by typing the variable's name. In fact, this is true of most OO languages, and this is why the functionality for `var` is defined in a generic `varD` function instead of defining it directly in the `VariableSym` `JavaCode` instance: the `varD` function can be re-used between language renderers. In the `VariableSym` instances for the other languages GOOL targets, the `var` function is defined as a dispatch to the `varD` function, exactly as it is for the Java instance. By writing generic functions for generating code that is common between target languages, we maximize code re-use and minimize effort required to write a renderer for a different language. The more similarities a language has to those that GOOL already targets, the less needs to be done to write the new renderer. GOOL's Java and C# renderers demonstrate this fact well. Out of 307 functions across all of GOOL's type classes, the instances of 216 of them are shared between the Java and C# renderers, in that they are just calls to the same common function. A further 20 are partially shared, for example they call the same common function but with different parameters. 137 functions are actually the same between all 4 languages GOOL currently targets, which might mean that some of them need not be included in the type class mechanism at all, and can instead be defined globally for all renderers, though this requires further investigation.

Examples from the Python and C# renderers 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 functions 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, however, because most modules are split between a source and header file. One module written in GOOL essentially needs to be traversed twice, once to generate the source file and a second time to generate the header file corresponding to the same module. To accomplish this, two instances of the GOOL type classes were written for C++ for two different types defined similarly to `JavaCode`: `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 is written such that an empty module or file does not actually get generated, and this is true of all GOOL's current renderers.

Since C++ source and header code should always be generated together, we needed to tie the `CppSrcCode` and `CppHdrCode` types together. To do this, a third type, `CppCode`, was created to pair the source and header types, defined as:

```
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 for now because we need to make an instance of a `Pair` type class for `CppCode`, as follows:

```
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
```

In summary, the `Pair` instance allows us to access the source code piece of the pair with `pfst`, the header code piece with `psnd`, or to combine a source and header piece to form a pair with `pair`. We then need to make `CppCode` instances of the GOOL type classes. The instance of `VariableSym` for the excerpt of the type class we have used as an example throughout this section is shown below.

```
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)
```

Rather than write the instance specifically for `CppCode`, we wrote it for any `Pair` to make this code more re-usable. Unfortunately, the types making up the `Pair` had to be concretized as `CppSrcCode` and `CppHdrCode` so that the Haskell compiler could be sure that the `VarData` underlying type declared for the `Pair` matched the underlying types for each piece of the pair. The compiler knows that `Variable CppSrcCode`

and `Variable CppHdrCode` are equivalent to `VarData`, but it cannot be sure that any generic `Variable repr` will be. The actual function definitions for these instances are trivial. The definition for `var` simply calls `var` in the context of `CppSrcCode`, calls it again in the context of `CppHdrCode`, and combines the results in a new `pair`. This same pattern is used for all of the function definitions for the `Pair` instances, so to write a new renderer for another language that requires two files per module, these `Pair` instances can be re-used just by changing the `CppSrcCode` and `CppHdrCode` types to the corresponding types for the new language renderer, and changing any underlying types that are different. This same technique can be used to write a renderer for a language that has even more than two files per module by writing an analog to the `Pair` type class that combines the appropriate number of constiuents.

The last component of the C++ renderer is a function analogous to `JavaCode`'s `unJC`, a function that can be called on a GOOL program to tell the Haskell compiler to concretize the `repr` as `CppCode` and therefore generate C++ code. The function is `unCPPC`, defined as:

```
unCPPC ::
  CppCode CppSrcCode CppHdrCode a -> a
unCPPC (CPPC (CPPSC a) _) = a
```

At the level of a `Program`, it no longer makes sense to consider source and header files independently, so the `Pair` instance of the `prog` function combines the source and header files together in the `CppSrcCode` piece of the `Pair`. This is why `unCPPC` simply extracts the `CppSrcCode` portion of the `Pair`—at that point, that portion contains both the source and header files. Note that the choice to combine them in the `CppSrcCode` half instead of the `CppHdrCode` half of the `Pair` was arbitrary.

After an `unRepr` function, like `unJC` or `unCPPC`, has been called on a GOOL program, the resulting pretty-printed `Docs` are transformed back into a `Strings` using the `render` function, and then written to files using standard Haskell IO.

5 Higher-level GOOL functions

GOOL provides many high-level functions that abstract over common OO patterns, allowing one to write a small amount of GOOL code to generate a large amount code in the target language, conforming to the target language's idioms, which may be very different between languages. This section shows examples of these high-level functions and the code they generate, starting with those that abstract simple patterns on the scale of values, and working up to those that abstract more complex patterns on the scale of entire blocks of code or functions.

One of the simplest examples of a high-level function in GOOL is `sin`, the function for applying the trigonometric sine function to a value. Most OO languages do offer this function, though usually as part of a library and not built in to the language itself. If a GOOL user wanted to apply a sine function and GOOL did not offer a function for doing this, the user would have to use `extFuncApp` and pass it the library name and function name. But the library and function names are likely to be different between target languages, so to target multiple languages, they would have to change the GOOL code for each. A high-level `sin` function eliminates this problem. Each language renderer can define `sin` by calling the specific library and function for that language, but all the GOOL user needs to do is write `sin foo` for some declared variable `foo`. This will yield `math.sin(foo)` in Python, `Math.sin(foo)` in Java, `Math.Sin(foo)` in C#, and `sin(foo)` in C++. In addition to `sin`, GOOL offers functions for applying remaining trigonometric functions and other common functions in mathematics: `log`, `ln`, `exp`, `floor`, and `ceiling`.

Another simple pattern GOOL abstracts is accessing arguments passed to a main function through the command line. The name given to the variable holding the list of arguments is different depending on the target language, so GOOL offers the `argsList` function to represent the value of that variable. GOOL also has functions for certain actions commonly taken with that list, such as `arg` which, when passed an integer index, represents the value of the argument at that index, and `argExists` which represents a Boolean value for whether an argument exists at a given index.

Lists are a very common data structure in OO code, so GOOL provides a suite of functions for working with lists. For example, `listAccess` can be passed a list and index value and will return the value of the element of the list at the given index. The GOOL code `listAccess (valueOf ages) (litInt 1)` for the `ages` variable defined in Section 3 will generate `ages[1]` in Python and C#, `ages.get(1)` in Java, and `ages.at(1)` in C++. Other functions for working with lists offered by GOOL are `listSize` for getting the size of a list, `listAppend` for appending a value to a list, `listAdd` for adding a value to a list at a given index, `listSet` for changing the value of a list at a given index to a given value, `listIndexExists` for checking if a list has a value at a given index, and `indexOf` for getting the index of a given value in a given list. Slicing a list can be done with `listSlice` by passing it a variable to assign the sliced list to, a list to slice, and three values representing the starting and ending indices for the slice and the step size. The values are wrapped in Haskell's `Maybe` monad and default to the start of the list, the end of the list, and a step size of one if `Nothing` is passed. So to take the elements from indices 1 and 2 of `ages` and assign them to a new list, `someAges`, the GOOL code looks like:

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

List slicing is of particular note because this action is much easier to do in Python than the other languages GOOL generates. The generated Python code for the above list slice is:

```
someAges = ages[1:3:]
```

While the generated Java code is shown below. Throughout this section, backslashes in generated code snippets indicate manually inserted line breaks, added so the code fits between the margins.

```
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;
```

The generated C# and C++ code blocks are similar in structure to the Java code. This example demonstrates GOOL's idiomatic code generation. It would have been easy to generate Python code with the same structure as the other languages, and the code would even be correct. However, since list slicing is available as a high-level function in GOOL, the renderers have the freedom to define that function in a way best-suited to the target language, and an idiomatic, more readable version was generated in Python as a result.

GOOL also offers high-level functions for printing. `print` prints a value, `printLn` prints a value followed by a new line character, and `printFile` and `printFileLn` are the same for printing to a file. There are also variations that are shortcuts for when the value to print is just a literal string, such as `printFileStrLn`. Unlike print functions in many of the target languages, these print functions can be used on lists. So the GOOL code `printLn ages` generates `print(ages)` in Python, but in the other languages it generates code to loop through the list and print each element, like the Java code shown below.

```
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("]");
```

This is another example where the renderers are clearly conforming to the idioms of a language. In addition to its functions for printing output, GOOL also offers functions for reading input, such as `getInput` and `getFileInput`.

Moving to larger-scale patterns, GOOL's `inOutFunc` generates a function based on three lists of variables: one of inputs, one of outputs, and one of variables that are both inputs and outputs. 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. It can be written in GOOL using `inOutFunc`, assuming variables `price`, `discount`, and `isAffordable` have been defined:

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

The `price` is both an input and output, so it is in the third list. The `discount` is an input only and `isAffordable` is an output only, so they go in the first and second lists, respectively. 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 [9]. 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

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

ThingML [10] 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 [16]. Clearwater is an approach for implementing DSLs with multiple target languages for components of distributed systems [17]. The Time Weaver tool uses a multi-language code generator to generate "glue" code for real-time embedded systems [8]. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which MobDSL [12] and XIS-Mobile [14] 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] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2009), 546–558.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White*

Paper 5, 8 (2007).

[17] 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 …