The GOOL system:

A lightweight Object-Oriented Programming language translator.

Pablo Arrighi Johan Girard Miguel Lezama Kévin Mazet
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
pablo.arrighi@imag.fr

Abstract

The GOOL system is a lightweight translator between OOP languages (`Java`, `C++`, `C#`, `Objective C`, ...). It relies upon a minimal, abstract OOP language called GOOL (General Object-Oriented Language) in order to represent the common features between these languages.

Keywords Code migration, porting

1. Introduction

Problem description. Having so many computer languages to choose from used to be a good thing. Each would bring new abstractions, convenient syntaxes for specific purposes — yet all of them would produce executables for the same platform. The tower of Babel problem was avoided. There has been relatively little incentive for code migration between all of these languages so far — except in order to ease code maintenance. Lately, however, each new platform comes with its own development language. `Java` is the language for the `JVM`, `C#` that of the `.NET` platform, `Objective C` is required for `iOS`, the `Android` language corresponds to the `Android OS`... The advantages of this new trend, in terms of new abstractions or convenient syntaxes, are unclear: besides GUI and the specifics of each platforms, these languages are quite similar. The costs for developers and the software industry, on the other hand, are quite clear: each new app has to be ported, and maintained, if it aims to reach the different platforms. We are faced with a tower of Babel problem. What can we do about it?

Bad solutions. One could seek to generalize these commonly-used OOP languages (`Java`, `C++`, `C#`, `Objective C`, ...) into a new OOP language. But this would be counter-productive, as it would create yet another roughly equivalent OOP language, adding to the problem instead of solving it. Consider n languages L_i, each targeted for a platform P_i. The traditional approach would be to develop one compiler per platform, for each language L_i, but this means $n(n-1)$ compilers to write. Similarly, one could seek to port each language L_i into L_j, but this means n^2 translators to write. Actually, there are some initiatives in this direction [1–5], some of which are heavyweight commercial solutions [6–9]. These face an issue: seeking to write a semantically correct, exhaustive translator, that takes into account every specifics of L_i (in terms of GUI, its interface with P_i, etc.), is a daunting task. It is bound to produce unreadable L_j code, hard to trust, adjust, maintain. Understandably, developers who want to keep control on the output code prefer to perform the translation by hand. But even this giving up is a bad idea: porting a large software project from L_i to $(L_j)_{j\neq i}$ manually is a boring task. These language have so much in common: except for GUI or platform specific aspects most changes are syntactic and thus clearly calling for automation.

Our approach. Instead of generalizing the commonly-used OOP languages into yet another concrete OOP language, we have sought to capture whatever is common to them, into an abstract OOP language, which we call GOOL (General Object-Oriented Language). The GOOL language has no syntax: it only aims to serve as a representation of those programming constructs which are common to these commonly-used OOP languages. Thus, it does not add to the tower of Babel problem, but constitutes a common socle between these languages. Based on this common socle, one can then consider parsing L_i into GOOL, and generating L_j from GOOL, thereby solving the above described problem with just n parsers and n generators. The solution, of course, is only partial: if GOOL is to represent $(L_i)_{i=1...n}$, it cannot represent any of the specific constructs of L_i, as a consequence those constructs will not be translated into L_j. These constructs do not have to be lost in translation, however. GOOL will hold them as "unrecognized constructs" in its abstract syntax tree. They will be generated as comments in L_j, which the developer can then translate by hand.

The GOOL system is indeed such a lightweight OOP language translator between the commonly-used OOP languages. The translation is performed in the natural manner, with the aim of always producing the most readable code. It is performed in a conservative manner: no information is deleted; whatever is 'unrecognized', gets 'passed on'. In other words, the GOOL system performs only the boring part of the translation, but in a readable conservative manner, so that the developer can focus on the interesting part (semantic adjustments, GUI, platform specific aspects). Hence the developer remains in control of the output code. The GOOL system itself is free, open, and designed with a horizontal, modular architecture so that input languages parsers and output languages generators may easily be added. It is also a platform for cross-compilation, and may serve for cross-language static analysis.

This paper. We first explain the architecture of the GOOL language and the GOOL system (Section 2), and how to deal with library mappings (Section 3). The status and history of the project are given in Section 4. The issue of correctness, as well as related works are discussed in Sections 5. Section 6 provides some perspectives.

2. Main architecture

The overall structure of the GOOL system is shown in Figure 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conf 9th ICOOOLPS, July 28th, 2014, Uppsala, Sweden.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

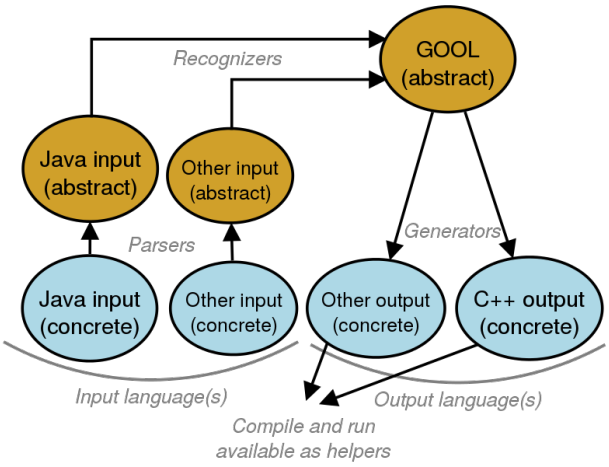


Figure 1. The architecture of the GOOL system

The abstract GOOL language. At the heart of the system is the GOOL abstract language. It serves as an intermediate representation of the input program, before it gets generated into the output program. As such, it must be able to represent the basic, frequent, common features of the commonly-used OOP languages (primitive types, expressions, ifs, whiles, fors, method declarations, calls, variables declarations, initialisations, assignments, class declarations, constructors, instantiations, packages...). In practice, it consists in a set of classes defining an extensible template for creating objects representing the AST of a generic OOP language see Figure 2. It is mostly inspired by the Java AST, but focussing on its essential features, in a way that is reminiscent of [10].

```
Node
  : Dependency
      : ClassDef : ...
      : ...
  : Member
  : Statement
      : Assign : ...
      : Block : ...
      : Dec
          : Field
          : Meth : ...
          : VarDeclaration
      : Expression
          : ArrayNew
          : CastExpression
          : ExpressionUnknown
          : Operation : ...
          : Parameterizable : ...
          : VarAccess : ...
          : ...
      : If
      : While : ...
      : ...
  : IType
      : PrimitiveType
          : TypeBool
          : ...
      : ReferenceType
          : TypeArray
          : TypeClass
          : ...
      : TypeException
      : TypeMethod
      : TypePackage
      : TypeVar
```

Figure 2. Extracts of the AST of the GOOL Language

Parsing: concrete input into abstract input. The commonly-used OOP languages (`Java`, `C++`, `C#`, `Objective C`, ...) that we would like to accept as inputs to the GOOL system have very elaborate grammars and type systems. In theory, formal descriptions of the grammars and type systems ought to be available as formal specifications of the languages, and could be used to write a parser for them from scratch, to feed them into the GOOL system. In practice, such descriptions are often unavailable, incomplete, or ambiguous [11, 12], and the task is daunting. Here one is better off adopting the pragmatic philosophy of the software industry, for which "the semantics of a programming language is given by what it does when you run it" [13]. This pragmatic approach has its limits, yet: what better definition of the C++ grammar and type system is there, than that implemented by `gcc` or `CDT`? What implementation of it is more trusted than theirs? For the commonly-used OOP languages, we are forced to admit that the established existing parsers constitute the best available definitions of their grammars and types. For the GOOL system, this means we can readily use, with the highest level of trust, some `Open JDK`[16] or `CDT` [14, 15] libraries to parse, and type, the concrete `Java` or `C++` input into the abstract input, with the entire type analysis, done.

Recognizing: abstact input into abstract GOOL. The `Open JDK` or `CDT` parsers produce typed AST representations of `Java` or `C++` , which expectedly are quite different from each other. At this stage, the GOOL system must reduce these different representations to a representation that is common to all of them, i.e. a GOOL AST. This is naturally done through a recursive descent in the input AST, via the visitor pattern. One difficulty is that commonly-used OOP languages are very rich. Forgetting about libraries for now, even the cores of these languages frequently have features which are too rare to be supported by the GOOL language (Ex: Operators overloading in C++). To make matters worse, they often have features which are too specific to them to be supported by the GOOL language (Ex: Multiple inheritance) — even though they are of common use. How must the GOOL system react, faced with such unrecognized features? To throw an exception at this stage would make the GOOL system practically worthless (users would need to significantly simplify their input code before the GOOL system would do anything else than just throwing exceptions at them). To insist on exhaustive support would probably make the GOOL system into a heavyweight, producing unreliable unreadable code, etc. Instead, unrecognized input features must simply be 'passed on' in a 'conservative' manner. For example, a pointer `C++` instruction is represented by a node `IASTPointerOperator` in the `CDT`-produced AST, which in turn is represented by an `ExpressionUnknown` node in the GOOL AST. This 'passing on' is done in a conservative manner: there is no information loss between the input AST and the GOOL AST. This whole stage achieves the recognition of the abstract input into abstract GOOL.

Generating: abstract GOOL into concrete input. At this stage, it only remains to generate the corresponding output language code. This again is done through a recursive descent in the GOOL AST, via the visitor pattern. One difficulty which could arise would be that some feature of the GOOL language does not have an equivalent in the output language, or that this part of the code generation has not yet been implemented. Again, to throw an exception at this stage would make the GOOL system practically worthless. Instead, ungenerated GOOL features must again simply be 'passed on' in a 'conservative' manner. In other words, they will trigger the generation of a comment:

```
/* Ungenerated, passed on by GOOL: ... */
```

providing all the information necessary for further, manual translation. Similarly, unrecognized input features which were represented

as `ExpressionUnknown` in the GOOL AST, trigger the generation of a comment:

```
/* Unrecognized, passed on by GOOL: ... */
```

providing all the information necessary for further, manual translation. This whole stage achieves the generation of the concrete output from the abstract GOOL.

Altogether, the GOOL system parses concrete input into abstract input, recognizes abstract input into abstract GOOL, and generates the concrete output from the abstract GOOL. Unless the concrete input is incorrect with respect the input language, the GOOL system does not throw an exception. The GOOL system is designed to be conservative: there is no loss of information between the concrete input and the concrete output. Indeed, if the GOOL system cannot handle some feature, it pass it on with a comment. This may seem like "passing the buck", if the hard part of the porting is simply passed down the chain, what has GOOL really accomplished? Well, the GOOL system has accomplished all boring part of the translation, leaving the fun to the developer. His subsequent work will be eased by the clarity of the translation, which preserves the software architecture, thus leaving him in a good position to tackle the sensitive semantical issues.

3. Dealing with libraries

Core language versus library separation. Core constructs refer to the basic constitutive elements of an OOP language (e.g., assignment instructions, for loops, try/catch blocks, ...). Often these constructs only differ in a matter of syntax, between the commonly-used OOP languages. Therefore, each new core construct can be added to the GOOL system by adding a new GOOL construct representing it, as well as its recognition and generation for each input and output language, as was described in Section 2.

On the other hand, library constructs refer to extensions of an OOP language (e.g., system, printing, files, lists, ...). Those are very often organized quite differently from one commonly-used OOP language to the other. Moreover, libraries are too large, and vary too much in time, for us to consider adding new GOOL constructs for representing them. Indeed, the GOOL language would become way too complex, and hard to maintain. Moreover, the addition of GOOL constructs requires some level of knowledge of the code of the GOOL system, which is too much to ask to the user. In order to achieve a more convenient solution, the GOOL system provides an easy, modular mechanism for the representation of libraries, via the concepts of abstract GOOL libraries and matching files.

Abstract GOOL libraries. Recall that the philosophy of the GOOL system is to use the GOOL abstract language as the pivotal language for porting between the concrete languages (L_i). This was true for core constructs, and must remain true for libraries. As a consequence, if we are to translate from `java.io.File` into `ifstream`, say, we must not do it directly. It is important that, first `java.io.File` be mapped to an abstract GOOL library `gool.io.File`, and then `gool.io.File` be mapped to `ifstream`. This `gool.io.File` is therefore independent from the languages (L_i), and must be thought as a General Object-Oriented Language API for handling files. It is just an abstract GOOL class, because the GOOL system will rely on the libraries of the output languages for implementing it. Of course this class has to be declared, and its signature has to be described somewhere: this is the role of certain configurations files, namely the GOOL library class declaration files. These are structured in packages and have a Java-like name pattern.

Intercepting and mapping libraries directly

Again if we are to translate from `java.io.File` into `ifstream`, `java.io.File` must be mapped to an abstract GOOL library `gool.io.File`, during the recognition of the input language. Concretely, an abstract GOOL class representing the `gool.io.File` abstract GOOL library needs be built. More importantly, the fact that
`java.io.File` matches `gool.io.File`, and how, has to be indicated somewhere. This is the role of certain configuration files, namely the input matching files. There are three types of input matching files, just like there will be three types of output matching files, namely the import matching files (II); the class matching files (IC); and the method matching files (IM). This is because the mapping typically acts at the level of imports, types, and method calls. For example, the necessary input matching files for the `java.io.File` library are:

```
## The file ImportMatching.properties (II)
# in gool.recognizer.java.matching :
gool.io.File  <-  java.io.File, java.io.*

## The file ClassMatching.properties (IC)
# in gool.recognizer.java.matching.io :
gool.io.File  <-  java.io.File

## The file MethodMatching.properties (IM)
# in gool.recognizer.java.matching.io :
gool.io.File.exists
          <-  java.io.File.exists():boolean
gool.io.File.createNewFile
          <-  java.io.File.createNewFile():boolean
gool.io.File.deleteFile
          <-  java.io.File.delete():boolean
```

Figures 4-6 describe their different usages. There is one instance of these files per input language and per GOOL library package, except for the 'input import matching file' which is only per input language.

Moreover, again if we are to translate from `java.io.File` into `ifstream`, `gool.io.File` must be mapped to the `C++` library `ifstream`, during the generation of the concrete output. And, of course, the fact that `gool.io.File` matches `ifstream`, and how, has to be indicated somewhere. This is again the role of certain configuration files, namely the output matching files. There are again three types of input matching files, just like there were three types of output matching files, namely the import matching files (OI); the class matching files (OC); and the method matching files (OM). They play similar roles. Figures 4-6 describe their different usages. There is one instance of these three files per output language and per GOOL library package.

Overall, we see that the declaration of these abstract GOOL libraries and their matching with concrete classes, methods and imports are handled just via configuration files. Thus, the extension of the existing set of GOOL libraries can be achieved solely by adding new configuration files or adding a few lines to the existing ones. This solution is therefore more simple and convenient: the recognition and generation of new library components does not inflate the set of GOOL AST constructs and does not involve any programming.

The GOOL library manager is the software component managing the mappings of library constructs from the input language and their mapping to the library constructs. It parses and retrieves the information given in the configuration files, and uses it to proceed with the recognition and generation of library constructs. It is therefore interfaced with both the recognizer and the generator of each language. Its architecture is shown in Figure 3.

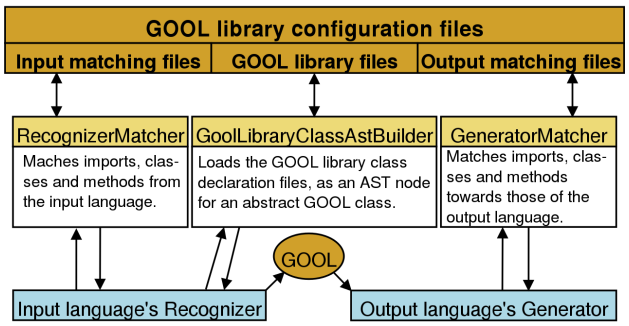


Figure 3. The architecture of the GOOL library manager

We now describe each of these configuration files in more detail. Instead of discussing the input matching files and then the output matching files, we bunch them up according to their types.

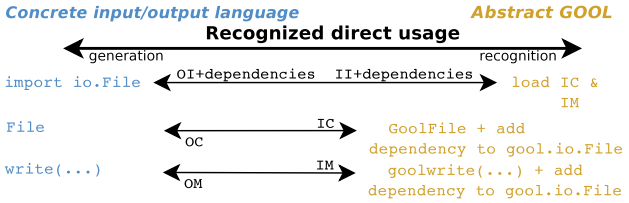
Import matching files. On the input side, there is one of them for each input language. It contains matches between each abstract GOOL class and the corresponding imports in the source code. When visiting the imports of the input code, the recognizer checks whether one is matched with a GOOL library. This is done by calling the GOOL library manager (RecognizerMatcher). If the import is indeed matched with a GOOL class, the interception of the matched GOOL class will be enabled.
On the output side, these files contain matches between each abstract GOOL library, and the corresponding imports of the output language, that need to be generated in order to use the matching library classes/methods of the output language.

Class matching files. Input class matching files and output class matching files match the GOOL classes with the input and output language classes, respectively. The input class matching helps the recognizer to build and annotate library constructs in the AST. The output class matching helps the generator to print the library constructs such as class instantiations in the output language.

Method matching files. They work in pairs with class matching files. On the input side, they contain matches between method signatures (each input language has its own signature syntax) and GOOL method names. They tell the recognizer to match method calls from the input language with GOOL abstract methods. On the output side, these files contain matches between GOOL method names and concrete method names from the output language. The syntax of the method matching files could also be extended to support additional features such as reordering parameters, placing in a constant, etc.

The simplest usage of these files is shown in Figure 4. This direct matching use case does the job whenever the input methods can be matched with GOOL methods, and the GOOL methods to the output method, in a one-to-one correspondence: the same behaviour and a compatible signature. Unfortunately, this is often not the case. Thus, the library manager also features indirect usages, in order to cover the more convoluted use cases.

Indirect matching of output libraries differing from those of GOOL. (see the Figure 5). When GOOL libraries do not provide an interface which can be directly translated into the output language (i.e., if there is no output method compatible by behaviour and signature), the system allows the use of a companion file to the output, which implements a GOOL-like API in the output language, in terms of the original output library. Thus, the implementation class acts as an wrapper of the output language library, so that it appears



Acronyms of the matching files involved:

```
II (input-import) , IC (input-class) , IM (input-method).
OI (output-import), OC (output-class), OM (output-method).

II: A table of goolclass <- imports
IC: A table of goolclass <- input classes
IM: A table of goolmethod <- input method
OI: A table of goolclass -> imports
OC: A table of goolclass -> output class
OM: A table of goolmethod -> output method
```

Ex., with `xxx` an input language and `yyy` an output language:

```
II: gool.io.GoolFile <- xxx.io.*
IC: gool.io.GoolFile <- xxx.io.File
IM: gool.io.GoolFile.goolwrite <- xxx.io.File.write(...):...
OI: gool.io.GoolFile -> yyy.io.File
OC: gool.io.GoolFile -> File
OM: gool.io.GoolFile.goolwrite -> write(...):...
```

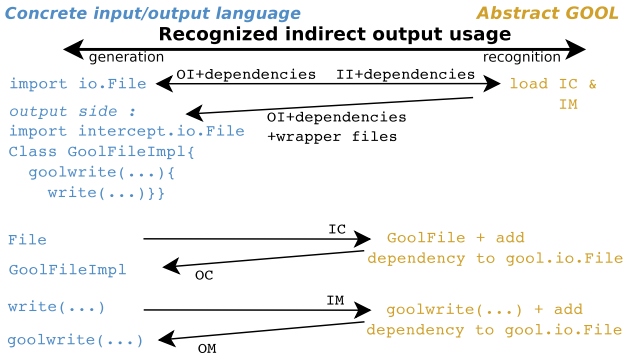
Figure 4. *The direct matching of libraries.* In this use case of the GOOL library manager, the input library and its methods are in one-to-one correspondence with the matching abstract GOOL library, which in turn is in one-to-one correspondence with the matching output library. The matching happens (above) as specified by the configuration files (below).

to follow a GOOL-like API. The output import matching files specify whether to use such companion implementation files. If used, the companion file is simply copied into the generated code. The output class/method matching files must then have been written so that the output class/method name match those of the provided implementation files.

Indirect matching of input libraries differing from those of GOOL. (see the Figure 6) Similarly when GOOL libraries do not provide an interface into which the input language library can be directly translated (i.e., if there is no GOOL method compatible by behaviour and signature), the system allows the use of a companion file to the input, which intercepts the calls to the input language library. Basically, the interception file consists in a new implementation of this specific input language library, by delegation, into a GOOL-like API. The intercepting class thus acts as a wrapper to GOOL's API, so that it appears to follow the original input library's API. The input import matching file specifies whether to use such companion interception files. If used, the companion file is simply copied into the input code. When parsing the input program, the interception files are also parsed. Thus, each class instantiation or method call to the original input library gets intercepted, and rephrased into the GOOL-like calls, which in turn get matched into the abstract GOOL library at recognition. Later, on the generator's side, the abstract GOOL library calls within the wrapper implementation will get matched directly or indirectly into the output libraries.

The combination of these two indirect matching ensures that the GOOL system remains minimal and generic (i.e., it keeps to a minimum the amount of abstract GOOL libraries, and ensures their independence with the input or output languages).

The unrecognized matching of library. (see the Figure 7) When the libraries in the input language are not known within the GOOL



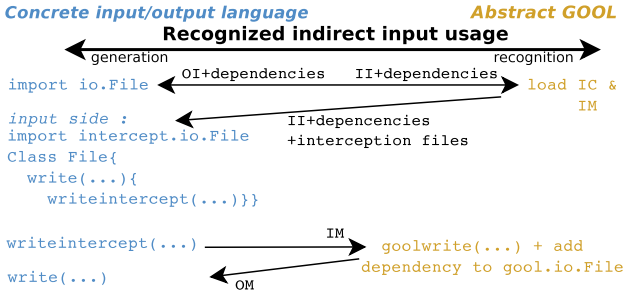
The output-import matching files may now provide extra implementation files:

```
OI: A table of goolclass -> +imports to implementation files
```

Ex., with yyy an output language:

```
OI: gool.io.GoolFile -> +goolyyy.io.GoolFileImpl
OC: gool.io.GoolFile -> GoolFileImpl
OM: gool.io.GoolFile.goolwrite -> goolwrite(...):...
```

Figure 5. *The indirect matching of output libraries.* In this use case of the GOOL library manager, there is a discrepancy between the abstract GOOL library API, and that of the matching output library. This is resolved by providing a companion file to the output, which implements a GOOL-like API in terms of the output API.



The input-import matching files may now provide extra interception files:

```
II: A table of goolclass <- +imports to interception files
```

Ex., with xxx an output language:

```
II: gool.io.GoolFile <- xxx.io.* +xxx.io.File
IC: gool.io.GoolFile <- xxx.io.File
IM: gool.io.GoolFile.goolwrite <- xxx.io.File.writeintercept(.)..
```

Figure 6. *The indirect matching of input libraries*. In this use case of the GOOL library manager, there is a discrepancy between the input library API and that of the matching abstract GOOL library. This is resolved by providing a companion file to the input, which replaces the problematic input library API and implements it in terms of a GOOL-like API.

library manager, the system generates the same input in the output language with an explicit comment of unregonized dependency.

4. Status and history of the project

Heterogeneous distributed systems. The kernel of the GOOL system originates from an implementation of π-calculus project. In this project, a single language would serve to describe a distributed

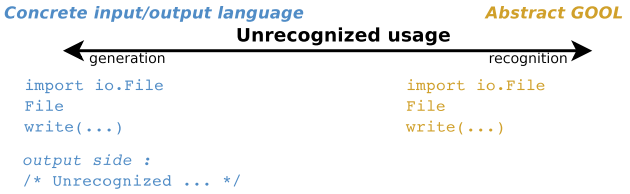


Figure 7. *The unrecognized case of matching a library.* In this use case of the GOOL library manager, there is no match between the input library and the abstract GOOL libraries. The input library imports, types and methods are simply translated syntactically, and a comment is generated.

application, with some processes running on a `JVM`, others on `.NET`, etc.... Hence, the language was to compile into GOOL, which would then compile some processes into `Java`, others in `C#`, etc. It was soon realized that plugging a Java parser into this GOOL component would make it into a standalone tool, useful for civilian purposes.

Compiler project. The GOOL system also originates from a pedagogical experience. First year Master students at the Computer Science department at our University used to do the same Compiler Project, year after year. We wanted to know whether these students (four groups of four over a month) could be made to contribute, incrementally, to a free software project in a useful way. A suitable project for this purpose must necessarily have a horizontal architecture: the cost of understanding the project must not augment as the project grows, and contributions can be made independent of each other. This is the case of the GOOL systems, which has a horizontal growth pattern, essentially consisting in the addition of new input and output languages. As a pedagogical experience, the GOOL system is a partial success. On the one hand students are enthusiastic to be trusted to make useful contributions to a long-term, collaborative, free software project. On the other hand, typically two or three out of their four contributions are discarded every year. Moreover, those which are kept require on average two months full-time work by an intern, for merger and consolidation.

Status. The GOOL system is available under the GNU General Public License (v3), via its GitHub repository [17]. It has had 12 forks and counts more than 13 000 lines of code. It takes, as input languages: `Java`, `C++`, `ObjC`. The last, however, is beta. It produces, as output languages: `Java`, `C++`, `ObjC`, `Python`. The library manager described in the previous section is functional, but only a handful of libraries are being dealt with for now.

5. About correctness and related works

The cost of heterogeneous correctness. Consider the following fragment of `Java` code:

```
int myArray[] = new int[5] ;
Scanner s = new Scanner(System.in) ;
System.out.println(myArray[s.nextInt()]) ;
```

It may result either in a zero or a ArrayIndexOutOfBoundsException exception. This is its `Java` semantics. Hence, if we wish to translate it faithfully into `C++` in a semantics preserving manner, we would need to generate some equivalent of the following code:

```
int i, myArray[5] ;
cin >> i ; if(i >= 5) throw ArrayException
else cout << myArray[i] << endl;
```

Whilst this is the semantically correct translation, this is not the natural, readable translation which most users of an OOP translator would have liked[1]. This is but a simple example of the high cost, in terms of software development, but more crucially in terms of output code readability, of seeking to translate OOP languages in a semantics preserving manner. In fact, we believe that the semantics differences between preserving commonly-used OOP languages seem so numerous and subtle, that no OOP language translator can claim to achieve semantics preservation. Moreover, any semantics preservation attempt would jeopardize the possibility of hinging around a common representation of OOP languages such as abstract GOOL. Indeed, abstract GOOL would no longer be able to represent both `Java` input language table accesses and `C++` input language table accesses as just a `ArrayAccess` node : it would need to differentiate guarded access from the unguarded, so that the appropriate concrete output be generated. Furthermore, even partial semantics preservation attempt would jeopardize output code readability, thereby preventing the user to manually fix the concrete output for his own purpose, and hence endangering, in practice, the semantics of the final translation. We are obliged to conclude that the theoretically sound approach, according to which each language has its own semantics, whom the translator should map into one another in a correct manner, fails to be fruitful in this real life context.

Aiming at minimal correctness. Instead of phrasing correctness in terms of the concrete input and concrete output languages, one could take an abstract-GOOL-centric view on this matter. The abstract GOOL language is limited to minimal, generic OOP language syntax, just like the WHILE language [18] is limited to your typical abstract, minimal, generic imperative language syntax. Recall that the WHILE language is the standard example when it comes to providing a rigorous semantics for an imperative language, and that it has been extended to the Object Oriented paradigm for instance in the COOL language [19], and others [10, 20]. Therefore, what could be done is to attribute, to the abstract GOOL language, the semantics of COOL [19], say. Next, we could demand that:

- The fragment of input language which corresponds COOL language, be accurately captured in abstract GOOL;
- That abstract GOOL be accurately mapped into the fragment of the output language which corresponds to COOL language.

The current version of the GOOL system has been written in this spirit, but thorough theoretical analysis and development effort would be required before any such notion of minimal correctness can really be claimed. For instance, a proper handling of issues such as single-inheritance of `Java` versus the multiple-inheritance of `C++` would need the back up of unifying semantical theories on the matter, such as [21]. The issue of memory handling, freeing objects, etc. raises similarly deep concerns.

Dialects and ideograms. The above discussion on ensuring minimal correctness is evocative of a common situation in natural languages, whereby a main language (French, Spanish. . .) has several dialects, all of them sharing a wide common base. This has the advantage that the common base is understood by the majority, but not at the cost of restricting the language, which the dialects extend and adapt to different needs. It would certainly help code migration and cross-platform compilation if new OOP languages were to respect previous standards, i.e. be built as dialects of their predecessors. The GOOL system could be seen as a practical contribution in this direction, as it recognizes the common base of the existing OOP languages. Yet, their syntaxes are so different already, that the role of the GOOL language might be more akin to that of the ideograms for the Chinese languages, i.e. an abstract representation of their common base. Moreover, this role, for now, is syntax-oriented rather than semantics-oriented: again at this stage the GOOL language cannot claim to be some WHILE language [18] of OOP, because the translation performed by the GOOL system, whilst natural and readable, remains semantically loose.

Pivotal languages. The idea of a pivotal language for translating between languages $(L_i)_{i=1\dots n}$ is not new. In natural languages, English frequently takes the role of the intermediate language between non-native English speakers. Volapuk and Esperanto were other attempts. In Computer Science, standards such as `Java ByteCode` (resp. `MS CIL`) play this role to some extent, as an intermediates between `Java` (resp. `C#`) and many platforms (or processors), whether at run-time or compile-time. Conversely, a compiler like `gcc` takes in a variety of input languages: `C, C++, Objective-C, Fortran, Java, Ada, Go`, with some degree of common intermediate representation, cf. `GIMPLE`. The GOOL system takes this idea at the level of OOP languages, for both the input and the output sides.

6. Outlooks

Extra features. We are looking towards adding more input languages to the GOOL system (consolidating `ObjC`, adding `C#`, . . .) and satisfying frequent demands such as the provision of a `Javascript` output. More libraries need be handled, especially those regarding networks, dynamical class loading, marshalling, etc.

Cross-platform compilation. In this paper, the GOOL system has been presented mainly as providing code migration between commonly used OOP languages. It may happen, however, that an application be written with cross-platform compilation in mind right from the start. This could be in the context of smartphone apps development, of distributed computing development, or that of the creation of the kernel of GOOL, see Section 4. In order to do so, it would suffice that the application be expressed in the abstract GOOL language. For instance, some process algebra could be compiled to the abstract GOOL language, and then cross-compiled via the GOOL system. Alternatively, a programmer may want to develop directly in the GOOL language: since the GOOL language is abstract, he could do so simply by keeping to the fragment of `Java` which is recognized by GOOL.

GOOL introspection. For now the GOOL system entirely relies upon its external, concrete input language to abstract input language parsers to perform name and type analysis. There could of course be an interest for a further step of analysis within the abstract GOOL representation, so as to enforce correctness according to intrinsic criteria. More importantly, perhaps, the GOOL system could become a convenient way to implement Cross-Platform analysis. Indeed, there are countless papers which provide static analysis methods, checking for Information flow, Security, Safety properties of object-oriented programs [22, 23]. Generally, these methods are not bound to one OOP language in particular, and yet the authors have to choose one of them in order to implement, test, and benchmark their methods — but could equally have chosen another. By choosing the abstract GOOL language, their static analytical tools would be readily available on each of the input languages to the GOOL system.

[1] Instead, the GOOL system opts for the following direct translation without test out of bounds (in `C++` for example) :

```
cout << myArray[i] << endl;
```

Acknowledgments

We would like to thank Alexander Concha who worked on the early version of GOOL, Philippe Bidinger for his many constructive comments, and the Master 1 Compiler students at Université Joseph Fourier.

References

[1] Versant Developer Community. *"Sharpen, an Eclipse plugin to convert Java into C#"*.

[2] Eclipse Labs. *"J2C, Java to C++ converter (Eclipse plugin)"*.

[3] Chris Laffra. *"C2J, a C++ to Java translator"*.

[4] Frank Buddrus and Jörg Schödel. 1998. *"Cappuccino – A C++ to Java translator"*. In Proceedings of the 1998 ACM symposium on Applied Computing (SAC '98). ACM, New York, NY, USA, 660-665. DOI=10.1145/330560.331015

[5] Scott Malabarba, Premkumar Devanbu, and Aaron Stearns. 1999. *"MoHCA-Java: a tool for C++ to Java conversion support"*. In Proceedings of the 21st international conference on Software engineering (ICSE '99). ACM, New York, NY, USA, 650-653. DOI=10.1145/302405.302918

[6] Tangible Software Solutions Inc. *"Source code conversion tools and source code conversion projects"*.

[7] Varycode Inc. *"Programming code conversions between different programming languages"*.

[8] Remotesoft Inc. *"Octopus .NET Translator"*.

[9] Microsoft Inc. *"Java Language Conversion Assistant 2.0 "*.

[10] Atsushi Igarashi, Benjamin C. Pierce, Philip Wadler, *"Featherweight Java: A Minimal Core Calculus for Java and GJ"*, ACM Trans. Program. Lang. Syst. **23**(3) 396–450, (2001).

[11] V. David, A. Demaille, R. Durlin and O. Gournet. 2005. *"C/C++ Disambiguation Using Attribute Grammars"*. Project Transformers, Utrecht University, Netherland.

[12] J. Gosling, B. Joy, and G. Steele. *"The Java Language Specification"*.

[13] Michael Lee Scott. *"Programming Language Pragmatics"*.

[14] PIATOV, Danila, JANES, Andrea, SILLITTI, Alberto, et al.*" Using the Eclipse C/C++ Development Tooling as a Robust, Fully Functional, Actively Maintained, Open Source C++ Parser"*. OSS, 2012, vol. 378, p. 399.

[15] M. Kucera, IBM Toronto Software Lab. *"Parsing and Analyzing C/C++ code in Eclipse"*.

[16] The openjdk project page. *"http://openjdk.java.net"*.

[17] The GOOL project page. *"https://github.com/librecoop/GOOL"*.

[18] KLEIN, Gerwin, LOETZBEYER, Heiko, NIPKOW, Tobias, et al. *"IMP–A WHILE-language and its Semantics"*. 2011.

[19] Alex Aiken. *"The Cool Reference Manual"*. University of Stanford.

[20] G. Rosu and T. F. Serbanuta. *"KOOL – Untyped"*. University of Illinois at Urbana-Champaign.

[21] Roland Ducournau, Jean Privat, *"Metamodeling semantics of multiple inheritance"*, Science of Computer Programming, **76**(7), 555–586, (2011).

[22] Jens Palsberg and Michael I. Schwartzbach. 1991. *"Object-oriented type inference"*. SIGPLAN Not. 26, 11 (November 1991), 146-161. DOI=10.1145/118014.117965

[23] GRAVER, Justin Owen. *"Type-checking and type-inference for object-oriented programming languages."* 1989. Thse de doctorat. University of Illinois.

[24] KONTOGIANNIS, Kostas, MARTIN, Johannes, WONG, Kenny, et al. *"Code migration through transformations: An experience report"*. In : CASCON First Decade High Impact Papers. IBM Corp., 2010. p. 201-213.

[25] TEREKHOV, Andrey A. et VERHOEF, Chris. *"The realities of language conversion"*s. IEEE Software, 2000, vol. 17, no 6, p. 111-124.