GoCxx: a tool to easily leverage C++ legacy code for multicore-friendly Go libraries and frameworks

Sébastien Binet
Laboratoire de l’Accélérateur Linéaire, Université Paris-Sud XI, 91898, Orsay, FR
E-mail: binet@cern.ch

Abstract. Current HENP libraries and frameworks were written before multicore systems became widely deployed and used. From this environment, a ‘single-thread’ processing model naturally emerged but the implicit assumptions it encouraged are greatly impairing our abilities to scale in a multicore/manycore world.

Writing scalable code in C++ for multicore architectures, while doable, is no panacea. Sure, C++11 will improve on the current situation (by standardizing on std::thread, introducing lambda functions and defining a memory model) but it will do so at the price of complicating further an already quite sophisticated language. This level of sophistication has probably already strongly motivated analysis groups to migrate to CPython, hoping for its current limitations with respect to multicore scalability to be either lifted (Grand Interpreter Lock removal) or for the advent of a new Python VM better tailored for this kind of environment (PyPy, Jython,...)

Could HENP migrate to a language with none of the deficiencies of C++ (build time, deployment, low level tools for concurrency) and with the fast turn-around time, simplicity and ease of coding of Python? This paper will try to make the case for Go - a young open source language with built-in facilities to easily express and expose concurrency - being such a language.

We introduce GoCxx, a tool leveraging gcc-xml’s output to automatize the tedious work of creating Go wrappers for foreign languages, a critical task for any language wishing to leverage legacy and field-tested code.

We will conclude with the first results of applying GoCxx to real C++ code.

1. Introduction
The “Free Lunch” is over: Moore’s law [1] can not be as easily leveraged as in the past, computer scientists and software writers have now to be familiar with Amdahl’s law [2]. Indeed, computers are no longer ticking faster: instead, they are growing more and more CPUs, each of which is no faster (frequency-wise) than the previous generation.

This increase in the number of cores evidently calls for more parallelism in HENP software. Fortunately, typical HENP applications (event reconstruction, event selection,...) are usually embarrassingly parallel, at least at the coarse-grained level: one “just” needs to call in parallel the portion of code which massages the events retrieved from the detector (the event loop) while still executing sequentially all the code processing each event.

However, this ‘one event per core’ strategy may not scale up to hundreds or thousands of cores, thus requiring library implementers and code writers at large to deal with sub-event parallelism. Parallel programming in C++ has been done and is doable, but even if many libraries such as Threading Building Blocks [6] or OpenMP [7] exist to help the burden, such
an endeavour is still quite tedious and error prone. Note that C++11 with lambda functions, std::thread, std::future and std::promise will likely improve the situation but at the price of sprinkling templates everywhere in the code, slowing down further the edit-compile-run cycle, and complicating further an already not so simple language. The recent gain in popularity of python [4] in HENP is probably a consequence of these two C++ defects, even if python is probably not the best language to tackle parallel high performance computing. At this point, it would seem reasonable to ask if using a new language more capable at leveraging multithreaded environments would be a better alternative [17].

This paper explores such a path. We first introduce some of Go most relevant features with regard to concurrency and how its module system addresses the build scalability and dependency hell issues. We then describe the process of leveraging the many (wo)man-years put into scientific FORTRAN/C/C++ libraries and making them available to the Go ecosystem. Finally, we present GoCxx, a tool using GCC-XML’s output to generate bindings to C++ libraries, as well as the libraries which compose it.

2. New languages
Since HENP and C++ met to produce (among other projects) Gaudi [12] and ROOT [8], the language landscape greatly changed. Many new languages appeared or became “mainstream” and, while closely following the language trend was not achieved, some adaptations were performed. For example, many LHC experiments use python for the job configuration. Many physicists use this scripting language for the final steps of the analysis, consisting in creating and grouping plots or processing text files for collecting summary data. This has mainly been possible thanks to the PyROOT module, giving access to python scripts to the whole ROOT API. But python (or more precisely CPython) has well-known scalability issues in a multithreaded environment because of its Global Interpreter Lock (GIL) which serializes access to python objects 1. Moreover, even if this issue can be worked around by writing C extension modules, having an event loop in an interpreted language is not the best bet CPU-speed wise.

We investigated Go as a possible alternative to C++.

3. Elements of Go
Go [5] is a new open source language from Google, first released in November 2009, and recently reached a new milestone: Go-1, the first stable release in March 2012. It is a compiled language with a garbage collector and built-in support for reflection, first-class functions, closures and object-oriented programming. The obligatory “hello world” program can be found in figure 1.

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello, world")
}
```

**Figure 1.** The obligatory “hello world” program, in Go.

Go is lauded to bring the best of both dynamic and static worlds:

* the feel of a dynamic language, thanks to its limited verbosity, its type inference system and its fast compile-edit-run cycle,

1 Other python implementations (JPython, IronPython,...) do not present this limitation.
• the safety of a static type system,
• the speed of a machine compiled language.  

Moreover, Go support for interfaces resembles the *duck-typing* motto of python. Finally and more importantly, Go has language support for concurrency, following the *Communicating Sequential Processes (CSP)* [13] model: prefixing a method or function call with the keyword `go` will spawn off a goroutine; the function will be executed concurrently to other codepaths. goroutines are multiplexed onto multiple OS threads so blocked goroutines because of a non-finished I/O operation will not halt the execution of the others. Furthermore, goroutines are lightweight thanks to their variable stack size, starting small and growing as needed.

In Go, the typesafe mechanism to exchange data between goroutines, is a *channel*. Sending or receiving data on a channel is atomic and can thus be used as a synchronization mechanism. It should be noted that as of Go-1, Go is lacking a few features which would probably make the implementation of typical HENP frameworks a bit easier, such as dynamic libraries and dynamic code loading. Another set of missing features more important for efficient scientific code is the lack of generics 3 and, for mathematical code clarity, the lack of operator overloading. Go programs are constructed by linking together packages, whose properties allow efficient management of dependencies. Each package may in turn use facilities provided by other packages. From these inter-package dependencies, forming an acyclic dependency graph, a correct compilation order can be inferred. Within a package, all global variables, functions and types defined in that package are visible. From the outside, only the exported ones are available. In Go, the rule about visibility of information is simple: if a name (of a top-level type, function, method, constant or variable, or of a structure field or method) is capitalized, users of the package may see it. Otherwise, the name and hence the thing being named is visible only inside the package in which it is declared. This is more than a convention: the rule is enforced by the compiler. Each compiled package file imports transitive dependency informations. If `A.go` depends on `B.go` which itself depends on `C.go`, compiling the package `A` will result in first compiling `C.go`, `B.go` and then `A.go`. And to recompile `A.go`, the compiler will only have to read the object file `B.o` but not `C.o`. At scale, this can result in huge speedups: the improvements become exponential.

Third-party Go libraries and programs are also easily distributed via *go get*, an executable providing automatic package installation with dependencies’ tracking. *go get* is the *de facto* standard to check out, compile and distribute Go code. An author wanting to distribute Go code just needs to publish it on some repository (*bitbucket, launchpad, github, ...*) and clients will be able to check out, compile and install it in one go, without special permissions, like in the shell session of figure 2.

```shell
shell> go get -x -v bitbucket.org/binet/go-cxxdict/pkg/cxxtypes/gccxml
fmt: skipping standard library

[...]
bitbucket.org/binet/go-cxxdict/pkg/cxxtypes
[...]
bitbucket.org/binet/go-cxxdict/pkg/cxxtypes/gccxml
```

**Figure 2.** Installing libraries or applications with *go get*. The gccxml Go package is being go-installed. Note how *go get* automatically installed the cxxtypes package, a dependency of gccxml.

2 the aim of the Go authors is to eventually bring the performances of a Go binary within 10% of C.
3 also called templates in C++.
Then, clients can import these packages like any other, with the correct import path, as shown in figure 3.

```
package mypackage

import "bitbucket.org/binet/go-croot/pkg/croot"
// croot can now be used like any package:
// file := croot.OpenFile("")
```

**Figure 3.** Importing and using third-party packages, using the correct import path.

Finally, a categorized list of all `go get`-able packages and commands is kept on the godashboard [14], providing the last piece for an effective CPAN-like facility.

4. **Wrapping foreign language libraries**

To paraphrase Tim Mattson, "successful new languages build on existing languages and where possible, support legacy software. C++ grew out of C. Java grew out of C++." Go is no exception and provides a standard way to access C libraries: cgo.

4.1. **Wrapping C libraries**

cgo allows to access and use entities defined in a C library with limited effort, as shown in figure 4.

```
package myclib

// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func foo(s string) {
    c_str := C.CString(s) // create a C string from a Go one
    C.fputs(c_str, C.stdout)
    C.free(unsafe.Pointer(c_str))
}
```

**Figure 4.** Wrapping C `stdio` with cgo.

Wrapping bigger libraries, such as C-BLAS, is no different albeit quite repetitive and mechanical a task, which should probably be automatized, using tools like SWIG [15].

4.2. **Wrapping FORTRAN libraries**

Wrapping FORTRAN 2003 code providing an ISO C interface is similar to the simple C library case. However, FORTRAN 77 libraries are more common in HENP and are dealt with the usual way:

- write a set of C wrappers to access the needed functionalities from the FORTRAN library,
- write cgo code to wrap the C wrappers.
4.3 Wrapping C++ libraries

Wrapping C++ is a bit more involved a task. cgo only supports C. The current solution is to use SWIG and a SWIG interface file to generate cgo code. The Go backend of SWIG is sophisticated enough to handle and map C++ constructs not existing in Go, back to idiomatic Go code. For example, overloaded functions are dispatched from a common Go variadic function to each cgo-wrapped function; while multiple inheritance is handled by defining multiple Go interfaces; and virtual methods are implemented by methods on interfaces. And thanks to SWIG machinery, it is even possible to implement a C++ abstract class with a Go struct implementing the according interface.

However, SWIG does not use a proper C++ compiler to parse the C++ header files and thus can not handle all of C++03 constructs. Attempts at wrapping ROOT’s TObject class failed, even after trying numerous tricks like presenting a preprocessed TObject.h file to SWIG or injecting #ifdef SWIG preprocessor directives. This motivated the development of GoCxx.

5. Architecture of GoCxx

GoCxx tries to address the shortcomings of cgo - wrapping only C - as well as those of SWIG - not being able to parse all C++ constructs. The latter is achieved by using a C++ compiler (GCC) and analyzing GCC-XML’s output to extract informations about types and identifiers of a compilation unit. The first version of GoCxx was directly modifying genreflex 4 python sources but the result even if functional, was not completely satisfactory due to the organical structure of genreflex. A rewrite in Go was believed to be a good educational purpose.

GoCxx follows the UNIX precept: a suite of simple tools doing only one simple thing, but can be easily combined. Thus, GoCxx provides:

- go-gencxxinfos, a tool to generate Go serialized objects describing C/C++ libraries,
- go-gencxxwrapper, a tool to generate bindings for the C/C++ libraries.

These binaries rely in turn on a few packages:

- cxxtypes models C++ types and identifiers, providing the same level of informations and query capabilities than Reflex and the additional ability to easily locate and load identifiers (e.g. enums and functions 5)
- wrapper provides the infrastructure and logic to load the C/C++ types and identifiers informations. It also schedules the plugins to generate the bindings. The plugin interface has been modeled after the Protocol Buffer [16] one.

A diagram representing the relations between these binaries and packages as well as the overall workflow is given in figure 5. The first step of this workflow is to run GCC-XML on the set of headers holding the API to be wrapped. This will produce an XML file which will be fed to and analyzed by the go-gencxxinfos binary, as shown in figure 6. Go already ships with a capable XML library: extracting the informations from the XML file boils down to writing Go-structs with a few annotations to associate XML tags and hierarchical layouts with Go-structs’ members. The rest -reading and parsing the file and then deserializing into the user provided structures- is performed by the standard library in a single API call. This task was facilitated by the built-in reflection capabilities of the Go language. Most of the implementation and testing work was thus devoted to create a Go package to model all the intricacies of a C++ type system. The representation of the compilation unit (classes, members, methods, functions, enums, . . . ) is then serialized into a Gob-file, the Go equivalent of the pickle files of python, for later use. This repository of C++ constructs can be queried to figure out which library provides which definition.

4 Reflex is a CERN library to enhance C++ with reflection capabilities [9].
5 This is not possible with Reflex at this time.
Figure 5. The overall workflow of GoCxx with its different components.

```
shell> gccxml lib.h -fxml=lib.xml
shell> go-gencxxinfos -fname lib.xml -o repo.db
```

Figure 6. First step of the GoCxx workflow: run GCC-XML to parse the C++ headers and then run go-gencxxinfos to model types and identifiers. This representation is then serialized for later use.

The last step of the GoCxx workflow is performed by go-gencxxwrapper which will load a (set of) repository file(s) and generate the bindings, using the wrapper package to do so. Multiple target languages are possible for the bindings as go-gencxxwrapper exposes a plugin API which could be implemented by third-party tools. GoCxx currently only provides a plugin to generate cgo code to wrap C/C++ but a plugin to generate Reflex stubs functions is planned.

```
shell> go-gencxxwrapper -fname repo.db -sel sel.xml \n   -pkg foo -o outdir
shell> ls outdir
foo.h foo-impl.cxx foo.go
```

Figure 7. Last step of the GoCxx workflow: load the repository of C++ informations and generate the wrapping code as a package named foo.

6. Conclusions

We presented the work on investigating Go as a viable C++ replacement for the multi-core era. Go has many built-in capabilities to ease the burden of exposing and leveraging concurrency in HENP applications as previously shown in [17]. The short development cycle together with the code distribution platform and tools provide Go with a short ‘time-to-market’ length.

Wrapping foreign language libraries, and especially C libraries, is made easy thanks to cgo, a tool shipped with the Go distribution. Wrapping C++ is possible - albeit cumbersome - with SWIG but is subject to SWIG’s limitations and does not seem to be applicable to large libraries such as ROOT. GoCxx, an automatized solution built on top of GCC-XML has been presented. While
package atlasevent

import "fmt"

// the foo package generated earlier
"foo"

func use_foo() {
    vec := foo.P4EEtaPhiM(20, 10, 10, 10)
    fmt.Printf("vec.E= %v\n", vec.E())
}

Figure 8. Using a C++ class from the ATLAS event data model.

some the generation of the wrapping code for some C++ constructs is not complete no technical showstopper was found.

Future work on GoCxx will focus on providing a CLang-based plugin to extract C++ informations as GCC-XML is now essentially unmaintained. Consolidation of the feature set as well as integration with the 'go get' binary are planned.

References
[1] Moore E, "Cramming more components onto integrated circuits", Electronics Magazine, 1965
[2] Amdahl G, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967
[3] The C++ programming language, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf
[4] The python programming language. http://python.org
[5] The Go programming language. http://golang.org/
[6] Intel Threading Building Blocks, http://threadingbuildingblocks.org
[7] OpenMP (Open Multi-Processing), http://openmp.org
[8] The ROOT framework, http://root.cern.ch
[9] S. Roiser, "The SEAL C++ Reflection System", CHEP-2004
[10] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage", ACM Trans. Math. Soft., 5 (1979), pp. 308–323., http://www.netlib.org
[11] G. Corcella, I.G. Knowles, G. Marchesini, S. Moretti, K. Odagiri, P. Richardson, M.H. Seymour and B.R. Webber, "HERWIG 6.5", JHEP 0101 (2001) 010 [hep-ph/0011363]; hep-ph/0210213
[12] Mato P 1998 Gaudi-architecture design document Tech. Rep. LHCb-98-064 Geneva
[13] CSP, http://en.wikipedia.org/wiki/Communicating_sequential_processes
[14] The Go dashboard, http://godashboard.appspot.com/project
[15] SWIG, http://swig.org
[16] Protobuf, http://code.google.com/p/protobuf/
[17] S. Binet, "ng: what next-generation languages can teach us about HENP frameworks in the manycore era", CHEP-2010
[18] https://bitbucket.org/binet/go-cxxdict

6 Support for dispatching method call to the proper method does not work yet in the case of multiple virtual inheritance.