Julia, C++, and ROOT

Philippe Gras

CEA/IRFU — Saclay

May 9, 22

Introduction

Definitions

- **Julia**: a great programming language that allies performance with ease of programming: may supersede the current Python $\oplus$ C++ HEP paradigm.
- **C++**: a high-performance language we all know,
- **ROOT**: a great and wide-used NHEP data analysis framework based on C++ and we all use.

This talk

I will address the interface between the three.

Introduction to Julia

Solving the two-language problem

Fast/easy coding		**Fast running**
Python	$\Leftrightarrow$	C/C++

$\Rightarrow$ Effect: mix of languages and going back-and-forth between them

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah tackled the problem in 2009

- Birth of Julia, release 0.1 in **2013**, a language that provides both fast coding and running
- Breakthrough recognised by 2 awards: James H. Wilkinson Prize in Numerical Analysis and Scientific in 2019, IEEE Computer Society Sidney Fernbach Award in 2019

Julia groups advantages of Python and C++ in a single programming language

Introduction to Julia: use in NHEP

- KM3Net high-level software has a Julia environment in development in addition to the Python one (reported here)
- The LEGEND $0\nu\beta\beta$ experiment uses two parallel stacks, the primary in Python and the secondary (for validation and experimentation) in Julia. C++ is used for Geant-4 simulation software (reported here)
- LHCb analysis that leads to the first observation of the $\Omega_b^- \to \Xi_c^+ K^- \pi^-$ decay (10.1103/PhysRevD.104.L091102) uses Julia: see Julia for data analysis in High Energy Physics, Mikael Mikhasenko. M. Mikkhasenko has used Julia also for a JPAC analysis (doi:10.1103/PhysRevD.98.096021) and a COMPAS analysis (doi:10.1103/PhysRevLett.127.082501) as reported here.
- Julia HEP organisation on Github
- Paper demonstrating the competitiveness of Julia for HEP published in last April Performance of Julia for High Energy Physics Analyses, Marcel Stanitzki and Jan Strube
- Julia-in-HEP session of PyHEP 2021 workshop, HSF Julia for HEP Mini-worksop

Interest of Julia in NHEP is growing.

Simple Julia code example

Open Data CMS dimuon spectrum analysis
Uses UnROOT.jl to read a CMS data ROOT file.

```
function analyze_tree(t, maxevents = -1)
    bins = 30_000 # Number of bins in the histogram
    low = 0.25 # Lower edge of the histogram
    up = 300.0 # Upper edge of the histogram
    h = H1{Float64}(Axis(bins, low, up))
    for (ievt, evt) in enumerate(t)
        maxevents >= 0 && ievt > maxevents && break
        evt.nMuon ==2 || continue
        evt.Muon_charge[1] != evt.Muon_charge[2] || continue
        dimuon_mass = m(ptetaphim(evt.Muon_pt[1], evt.Muon_eta[1], evt.Muon_phi[1],
                                  evt.Muon_mass[1])
                      + ptetaphim(evt.Muon_pt[2], evt.Muon_eta[2],
                                  evt.Muon_phi[2], evt.Muon_mass[2]))
        hfill!(h, dimuon_mass)
    end
    h
end

t = LazyTree(ROOTFile(fname),"Events")
h = analyze_tree(t);
```

Support for ROOT file format

Currently two options available

- UnROOT.jl written in Julia. Limited to supported class and tree contents, but development plans to go beyond. No write support.
- Use of Python uproot library via the Julia-Python transparent interface. Read and write access. Limited to supported class and tree contents. See also UpROOT.jl.

Interporability with other languages

Direct call from Julia

- Python. Very well integrated in both directions, included in Jupyter notebooks. `PyCall` and `IJulia`.
- Fortran and C: built-in `ccall` function, no overhead.
  - call of c-function `double f(double)`:
    ```
    @ccall f(x.::Cdouble)::Cdouble
    ```
  - Writing wrapper in Julia needed for a good integration.
  - Julia code can also be called from C and C++ code (with a time overhead).

Requiring wrapper code

- C++. `CxxWrap.jl`: glueing code to write in C++. Inspired from `Boost.Python` like `PyBind11` is.

Note: Direct C++ code call possible with old versions of Julia (1.1.x–1.3.x) with the `Cxx.jl` package. Equivalent to `cppyy`.

CxxWrap

How to add Julia bindings to a C++ library with CxxWrap?

- Compile a shared library, where we register the types and functions to bind

  1 or 2 line per class + 1 per function;

Behind the scene

- The shared library wraps the C++ functions/methods in C functions in order to use the Julia built-in C call.
- The package generates (at runtime) Julia wrappers

Features

- Perfect integration, no need to write Julia wrappers;

  `a->f(x)` maps to Julia style `f(a, x)`
- Mapping of single class inheritance;
- Beside the class method mapping, support mapping of pure structs to Julia.

Writing the c++ glue: examples

Example

```
//TH1 and TH1F registration with
//inheritance specification
th1  = types.add_type<TH1>("TH1F",
                           jlcxx::julia_base_type<TNamed>());
th1f = types.add_type<TH1F>("TH1F",
                            jlcxx::julia_base_type<TH1>());

//TH1::Fill method registration
th1.method("Fill",
  static_cast<Int_t (TH1::*)(Double_t) >(&TH1::Fill));
```

CxxWrap.jl used by the Fastjet and LCIO Julia interfaces developed by Jan Strube.

Writing the c++ glue: automatisation

Writing the type and function registration lines is:

- Simple
- but it can be cumbersome to cover all classes and methods of a large library
  - → Covering all the ROOT classes and methods would be a huge work
- Needs to be updated when the library API is changed.

Automatic generation of glue

- If the code is simple, then it can be automatically generated
- Automatic generation investigated with the WrapIt!
  proof-of-concept project

WrapIt!

Principle

- Produces the glueing code from the library header files
- Minimal configuration

Challenges

- Interpreting content written in sophisticated language (C++20 standard: 1853 pages!)
- Header files $\neq$ API definition

Design choices

- Written in C++
- Use of LLVM/Clang
  - mainly libclang: stable C API of clang libraries;
  - few calls to Clang AST C++ library for few missing features of libclang.

WrapIt! features

- Takes a list of header files containing the classes and functions to wrap.
- Adds automatically to the list all the types needed to use the wrapped functions (argument and return types).
- Selection of classes and functions to wrap can be fine-tuned by providing an exclusion list.
- Maps of inheritance: max. one parent class. Selection of parent class configurable.
- Supports generation of accessors for class fields.

WrapIt! demo

Based on ex002-ROOT example

We will do the following from Julia:

- Book a TH1D histogram;
- Fill the histogram with random numbers;
- Fit the histogram;
- Display the histogram on Screen;
- Save in a `.root` file the histogram and the TCanvas;
- Save the Figure as an image file.

Towards a ROOT-Julia interface

What is missing to generate a complete ROOT framework Julia-binding?

- Split of generated code in several files to reduce resources to compile it;
- Needs to improve WrapIt! code: missing features discovered while adding more class to wrap;
- Graphic support in Jupyter to be added;
- Support to read/write TTree needs to be debugged. Not clear how much development this support will require.
- Support of Julia to write functions for RDataFrame::Filter() and RDataFrame::Define(). Not clear yet how this should be implemented.

Conclusions

- Julia is ideal for NHEP data analysis: combines fast coding and fast running;
- Julia binding to ROOT is on the way
  - Based on automatic code generation to ease the maintenance.
  - Well integrated interface out-of-the box, although we need to understand how it will be for TTree and RNtuple I/O.

Appendix

Programming with Julia is easy

- Code syntax and grammar is similar to Pythons. No *std::map<std:string, std::vector<MyClass»*..., no compilation step.
- Dynamic type system
- Easy to learn
- Syntactic sugars similar to Python for a concise code: list comprehension, a < b < c, 1_000_000, support of symbols for variables...

and more: e.g. a function call is "vectorized" (ala numpy) with a simple dot, `f.(x)`

- Interactive help, nice tools to debug, to optimise code, for introspection.

What makes Julia unique

Designed from the very beginning with the goal of conciliating high performance computing with easy coding

Just-in-time compilation

- Provides both fast execution and a good interactive experience

Support for **Ju**pyter notebook

- (**Ju** stands for Julia).

A dynamic type system

- Specification of variable and function argument types optional.

Its dynamic multiple dispatch paradigm

- Functions can be implemented once and support many argument types;
- Specialized implementations can be provided for a specific type (or group of types) of one argument or more.
- Achieve in a much simpler, more consistent and more complete manner what C++ provides with templates+overloading+overriding.

Programming in a community

- Internet search engine and stack overflow play is an essential ingredient in nowadays programming workflow.
- Julia is already widespread enough, to find all the information on the Internet.
- In addition to usual resource, Julia has dedicated fora on Discourse, Slack, and Zulip with an active and friendly community.

Go to `https://www.duckduckgo.com` or your preferred search engine and make a try.

A rich ecosystem

- Large set of libraries and active development
  - Julia is firstly used by scientific community $\Rightarrow$ oriented to our needs
- Machine Learning, GPU, Plotting, DataFrames, etc...
- I did the following exercise during the PyHEP2021 workshop: I've looked for a Julia equivalent each time a speaker mention a Python library (apart from HEP specific ones).
  - Found a Julia equivalent of 16 out of the 18 mentioned libraries: missing one was a binding to FreeCAD (which is in discussion) and the software testing library with a specific technique (Hypothesis).

Develoment tools for Julia

IDE

- Emacs and vim support
- Atom and VScode support. Many features. Code can be run and debugged with the IDE, with support for plots.

Notebooks

- **Ju**pyter
- Pluto. A new generation notebook with automatic update of cells.

Debugger

- Debugger, Rebugger, Juno debugger (for Atom IDE)

Code optimisation

- Integrates nice and easy-to-use tools to optimise code performance.

Package installation

Package installation

Python made it easy with Conda and pip. It's even easier in Julia

- A standard library part of the Julia installation
- Give instructions to the user, when he or she tries to import a missing package.

Comparison with Python (1/2)

The main practical difference with Python is that Julia code runs faster.

- Canfranc DQM code was originally written to Python.
- Python code needs to be written in a certain way to get the best performance (but still worst than Fortran/C/C++/Julia.)
  - $\Rightarrow$ Has required a substantial effort when writing the DQM code.
- The code was rewritten in C, because of performance issue in terms of running speed and memory usage, despite use of state-of-the art technics for performance (`Numba`, `Numpy`).

Comparison with Python (2/2)

Loops are not issue with Julia

```julia
#
# Julia
#
function f()
    a = 0.0
    for i in 1:1_000_000 # ✨ Note the underscores that improves legibility
       a = a + 1.0/i
    end
    return a
end
f()
@time b = f()
```

```
  0.001004 seconds (1 allocation: 16 bytes)
```

```
14.392726722864989
```

C/C++	Python	Julia
1.0ms	44ms	1.0ms

- As simple as Python, as fast as C/C++