Conversation between a Julian and non-Julians

2.9.2024 OpenMendel meeting

Some personal encounters

“How do I use your [Julia] software in R/Python?”

[My collaborator have some C++ code (wrapped by R/Rcpp) that I want to use.] “How do I call it from Julia?”

“Most of genetics and bioinformatics rely on binaries. Can you get a binary from Julia?”

Today's agenda

I want to share my experience in doing the following

- Writing Python/R package that internally calls Julia code
- Writing C++ wrapper to be called by Julia
- Creating "binary executables" from Julia package

Today's agenda

I want to share my experience in doing the following

- Writing Python/R package that internally calls Julia code

  This is easy (2-3 days of effort)

- Writing C++ wrapper to be called by Julia

  This is hard (1-2 month effort) unless you are C++ expert

- Creating "binary executables" from Julia package

This is easy (1-2 days effort) if you are willing to accept >100MB "binaries"

Write a python/R wrapper for your Julia package

- The packages JuliaCall (juliacall) calls Julia from R (python)
- Our strategy: write a thin python/R package that:
  1. Installs Julia (using JuliaCall)
  2. Tell Julia to install your Julia package
  3. Call into Julia code from R/Python via JuliaCall
- This strategy is used by SciML for DifferentialEquations.jl, I copied them for the Knockoffs.jl package
- For a regular Python/R user, the fact that most code is written in Julia becomes an implementation detail
- Effort required: writing the wrapper is rather quick, but re-writing documentations/unit tests/CI are time consuming. You need documentation to put an R package on CRAN (but PyPI will take anything).

Example: knockoffsr and knockoffspy for Knockoffs.jl

- These are thin wrapper packages with only 62 (91) lines of source code
- They are regular python/R packages and can be installed in the regular way
  - In python: `pip install knockoffspy`
  - In R:

```
library(devtools)
install_github("biona001/knockoffsr")
```

- Look at diffeqr and diffeqpy for full examples setting up unit test/CI/docs

Example Usage

- In R

```
> library(knockoffsr)
> ko <- knockoffsr::knockoff_setup()
> result <- ko$modelX_gaussian_group_knockoffs(
                X, “maxent”) # assumes X is available
> S <- result$S
```

- In python

```
>>> from knockoffspy import ko
>>> ko.install()
>>> result = ko.modelX_gaussian_group_knockoffs(
                X, "maxent") # assumes X available
>>> S = result.S
```

```
struct GaussianGroupKnockoff{T}
    X::Matrix{T}
    Xko::Matrix{T}
    groups::Vector{Int}
    S::Matrix{T}
    gammas::Vector{T}
    m::Int
    Sigma::Matrix{T}
    method::Symbol
    obj::T
end
```

In Julia, the function modelX_gaussian_group_knockoffs() returns a struct

Ben's recommendations for writing Julia code (to make Python/R wrapping easier)

- Use regular English letters for structs, i.e. no Greek letters, latex annotations, or emojis (you cannot type them in R)
- Functions should accept strings rather than symbols (no "symbol" in R/python)
- Functions should not require Julia-specific types, e.g. ::Symmetric would fail. Rather, it should accept AbstractMatrix and you write a separate routine to check symmetry, e.g. isapprox(X, X', atol=1e-8)

More than 90% people use R/python. If it is easy to provide an R/python wrapper, why not?

Writing Julia wrapper for C++ code

- Overall strategy (red = step that was hard for me):
  1. Clone target repository containing C++ code
  2. Write wrapper code (in C++) via CxxWrap.jl, upload code to forked repo
     - We will compile it into a shared library (.so) file
  3. Use BinaryBuilder.jl to compile wrapper code into a jll package
     - Compilation for different OS and CPU architectures is possible (the build script is written by you!)
  4. Deploy jll package to your personal github, then to Yggdrasil when it is fully ready
  5. To expose functionality, write regular Julia package that calls jll package internally
- In this setup,
  - Users run Julia code the standard Julian way. The fact that your package has C++ dependency becomes an implementation detail
  - Users never compile anything: compiled library (i.e. provided by your jll package) is **downloaded** as an artifact
    - In contrast, R/RCpp make users run a build script to compile the C++ code during package installation, and this step fails very often
  - Extremely small overhead calling C++ code (not humanly noticeable)

A Full Example

I am providing everything here, even though they are surely not good example, because I had trouble finding a **full** example online

- Basic C++ wrapper
  - https://github.com/biona001/ghostbasil/blob/master/julia/ghostbasil_wrap.cpp
- Basic build_tarballs.jl (from BinaryBuilder.jl):
  - https://github.com/biona001/Ghostbasil.jl/blob/main/src/build_tarballs.jl
  - Run `julia build_tarballs.jl --deploy="GithubUSERID/SITE_jll.jl"` to get jll package
- Basic jll package
  - https://github.com/biona001/ghostbasil_jll.jl
- Basic upper level package:
  - https://github.com/biona001/Ghostbasil.jl

Everything + explanations:
https://github.com/biona001/Ghostbasil.jl/blob/main/test/400phenotypes_binary_executable.ipynb

Ben's recommendation for writing C++ wrappers

- Finding a full example is the most important imo. Somehow, this was difficult
  - If a python/R wrapper exist for the C++ code, study them carefully
- Use CxxWrap.jl instead of Cxx.jl
  - Bart Janssens (@barche) did many helpful CxxWrap.jl tutorial/workshop at JuliaCon. The actual documentation of CxxWrap.jl was not very helpful for me
  - jluna.jl seems promising and well maintained but I have not tried it
- wrapit is C++ package which automatically generates C++ wrapper for Julia, based on CxxWrap.jl. You should try it first, but it never worked for me.
- Specifying compat entry in build_tarballs.jl for libcxxwrap_julia_jll is extremely important, e.g.

```
Dependency("libcxxwrap_julia_jll"; compat = "0.11.2")
```

  CxxWrap.jl internally depends on this binary, but it **often** release breaking changes
- The step calling BinaryBuilder.jl can be complicated. Fortunately, thousands of example `build_tarballs.jl` files can be found at Yggdrasil. Unfortunately, most of them copy each other.

Many R/Python packages are internally calling C++ code, but as of 2024 it is not easy to write Julia wrappers for them, let alone automate it

Building binaries from Julia package

- Many users prefer binaries:
  - No required installation/compilation (just download a pre-compiled program)
  - Clearly defined inputs/outputs
  - E.g. PLINK, bcftools, ADMIXTURE, … etc
- Strategy: use `create_app` function of PackageCompiler.jl
  - StaticCompiler.jl is another option, but it seems less flexible (I have not tried it)
- PackageCompiler.jl produces **a folder** which includes the desired executable + all the necessary files to run it
  - When zipped, it is 100-400 MB (seems like this number can be optimized)
- The entire folder can be sent to another machine directly, e.g. one without Julia installed
  - The target machine **must have the same operating system**
  - Unlike BinaryBuilder.jl, one needs to compile for Mac, Linux, Windows separately (e.g. via AWS?)

Strategy

1. Write a regular Julia package
2. Define a function `julia_main()` which operates on `ARGS` and returns
   - 0 (if program successfully ran) or
   - 1 (if an error occurred)

```
function julia_main()::Cint
    try
        # do something based on ARGS?
    catch
        Base.invokelatest(Base.display_error, Base.catch_stack())
        return 1
    end
    return 0 # if things finished successfully
end
```

3. Call `PackageCompiler.create_app`
   - src = directory of your package
   - des = output folder
   - The result is an executable whose usage is
     PKG_NAME ARGS[1] ARGS[2] …
   - `include_lazy_artifacts` keyword is required if your package has external (artifact) dependencies, e.g. one based on a jll package

```
using PackageCompiler, GhostKnockoffGWAS
src = normpath(pathof(GhostKnockoffGWAS), "../..")
des = normpath(pathof(GhostKnockoffGWAS), "../../linux_64")
@time create_app(src, des,
    include_lazy_artifacts=true,
    force=true,
)
```

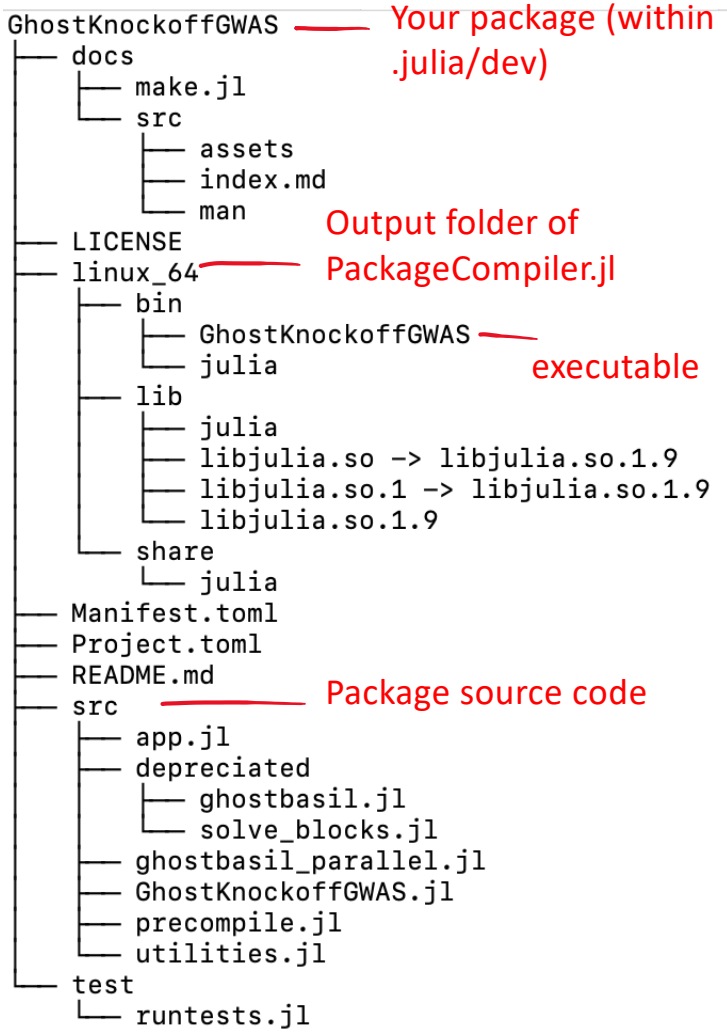
Example

- `create_app()` will run for 1-2h and then generate a folder containing the executable + bundle of other files →→
- The executable can be invoked in the terminal directly, e.g.

```
$ GhostKnockoffGWAS example_zfile.txt EUR 506200 38 example_output
```

- For more command-line-program-like usage, use ArgParse.jl to parse `ARGS`:

```
$ GhostKnockoffGWAS \
    --zfile example_zfile.txt \
    --LD-files EUR \
    --N 506200 \
    --genome-build 38 \
    --out example_output
```



Ben's recommendation for using PackageCompiler.jl

- Compiling Julia code into an app (binary? Executable?) is pretty easy
  - The result is not a "true binary", whatever that means
- Beware of accidentally depending on large binaries (e.g. MKL_jll.jl is 600+MB)
- Underlying Julia code could still be **JIT compiled**, if you are not careful with precompilation statements
  - E.g. My "time to help statement" now takes 0.5 sec (still slow) but an improvement from 10s

```
$ time GhostKnockoffGWAS –h
  real    0m0.574s
  user    0m0.343s
  sys     0m0.159s
```

- Still not easy to compile for every platform (e.g. Windows/Mac/linux), not sure why it can't work like BinaryBuilder.jl
- Still not clear how to distribute the resulting (large) binaries

Summary

- Writing Python/R package that internally calls Julia code?

  This is easy (2-3 days of effort), but re-writing documentation/tests/CI is a lot more effort

- Writing C++ wrapper to be called by Julia?

  This is hard (1-2 month effort) unless you are C++ expert. It is probably easier if you re-implement everything in Julia

- Creating “binary executables” from Julia package?

  This is easy (1-2 days effort) if you are willing to accept >100MB “binaries”. You might encounter small inconveniences but I’m satisfied overall