Notes on the Julia Programming Language

Lutz Hendricks

UNC Chapel Hill

August 5, 2019

Abstract

This document summarizes my experience with the Julia language. Its main purpose is to document tips and tricks that are not covered in the official documentation.

1 My Setup (1.1)[1]

Since things are in flux, I find it useful to use the offical `JuliaPro` installation.

My startup file loads the packages `OhMyREPL` and `Revise`. `Revise` comes after packages from the standard libraries, so it does not track changes to those.

1.1 JuliaPro (1.1)

Sometimes JuliaPro gets slow and has trouble updating the REPL screen. Then restarting the computer is the only solution.

1.2 Julia + Editor (1.1)

It appears that the default editor is determined by the system wide file association. No need to set the `JULIA_EDITOR` environment variable.

[1] Each section is labeled with the Julia version for which it was last updated.

One drawback: Links in the terminal REPL are not clickable. A substantial drawback during debugging. So I end up using BBEdit as my main editor, but do some debugging in `Juno`. Not ideal.

2 Arrays (1.1)

2.1 Indexing

Extracting specific elements with indices given by vectors:

```
A = rand(4,3,5);
A[CartesianIndex.([1,2], [2,2]), 1] -> A[1,2,1] and A[2,2,1]
```

Similar to using sub2ind:

```
idxV = sub2ind(size(A), [1,2],[2,2],[1,1])
A[idxV]
```

To extract a "row" of a multidimensional matrix without hard-coding the dimensions, generate a `view` using `selectdim`.

3 Debugging

The `Juno` debugger stopped working in V.1.1 (invoking it hangs `Julia`). But the command line debugger may well be the better option. After `using Debugger` invoke `@enter foo(x)` to start a debugging session.

Particularly useful:

- `bp on error`
- `bp add func:line` with possible restrictions on particular argument types.

4 Formatting

The `Formatting` package seems to be the best bet. It uses `Python` like syntax and can format multiple arguments simultaneously (not well documented). Example:

```
fs = FormatExpr("{1:.2f} and {2:.3f}")
format(fs, 1.123, 2)
```

yields `"1.12 and 2.000"`.

5 Modules

5.1 Extending a function in another module (1.1)

The problem:

- Module `B` defines type `Tb` and function `foo(x :: Tb)`.
- Module `A` contains a generic function `bar(x)` that calls `foo()`. It should use the `foo()` that matches the type of `x`. That is, when called as `foo(x :: Tb)`, we want to call `B.foo`.

Solution:

- Module `A`:
  - Define the stub: `function foo end`
  - Call `foo(x)` from within `bar`.
- Module `B`:
  - Define `function foo(x :: Tb)`
  - `import A.foo`
- Now `A.bar(x)` knows about `B.foo()` and calls it when the type matches the signature.

See Duck typing when ‘quack‘ is not in ‘Base‘.

6 Packages

The intended workflow is:

1. Make a new package somewhere (in your project folder). push it to github.

2. If the package is used somewhere else (general purpose code), place it into the JULIA_PKG_DEVDIR folder (by default ~/.julia/dev/MyPackage). When a package is updated from a registry, this is where the code gets downloaded to. This is also where Julia tracks changes to the code when a packge is not registered.

3. Edit the code. Once something is in working order, push to github again with a new version number. This is a key point: if the version number does not change, code that gets downloaded from a registry will never know that the package has changed. In particular, if you deploy on a server, the server will not update.

6.1 Creating a package (1.1)

The easiest way is PkgSkeleton.jl. You need to set your github info (user.name etc) using

git config --global user.name YourName

This must be done insite a git directory. Then generate generates the directory structure and the required files (Project.toml etc).

Note: The package name should not end in ■.jl■ – this is automatically appended when the package is registered (?).

6.2 Unregistered packages (1.1)

Important point: Unregistered packages need to be added as dependencies "by hand." Pkg cannot track when other packages depend on them. This is a known issue 810.

Adding by hand involves] add <path to github repo> or by making a PackageSpec(name = MyPackage, url = ■https://github.com/user/MyPackage.git■) and issuing Pkg.add(pSpec).

Since you cannot register all sorts of small packages (that are frequently updated), most of your code will never be in registered packages. Unless you create your own registry.

Tracking changes in unregistered packages is difficult. For me, the following do not work

- `Pkg.update`
- Edit source in `JULIA_PKG_DEVDIR`
- `Pkg.rm` followed by `Pkg.add`

What works:

- Restart the `REPL`. This recompiles the **local** version of the code (not downloading from `github`, unless perhaps the `github` version is newer?).
- `Pkg.develop`. This downloads a copy of the code to `JULIA_PKG_DEVDIR`. But only if the code does not already exist (or is older, I suppose). Then recompiles. Implicitly, this changes the `url` in `Manifest.toml` to the local path where the code resides.
- Unfortunately, `Pkg.free` does not work, even if a `UUID` is given ("no versions left" error). To free a package and to fix a specific `github` commit that is tracked, the package needs to be removed and then added again.
- Editing locally (not in JULIA_PKG_DEVDIR) and then issuing `Pkg.develop` does not work. Because Julia does not see a new version on `github`. This is true even if the version number is increased in `Project.toml`.

So the workflow is:

- Set JULIA_PKG_DEVDIR to point to where you edit the package.
- Edit and test. This works fine as long as no dependencies change. `test` always uses a freshly compiled version of the package.
- For dependency packages: `Pkg.develop(PackageSpec)` (issued in the main project's environment). Edits are now tracked in the main project. `develop` is a setting that persists even after restarting the REPL.

- When the code is supposed to be frozen (e.g. at paper submission), remove and re-add all unregistered packages (to fix their versions to specific `github` commits).

Unclear: How to deploy the code to another computer?

6.3 Creating a package registry (1.1)

Any registry that lives in `~/.julia/registries` is automatically used by `Pkg`.

In principle, it is easy to create your own registry (see discourse for a guide). But there are problems:

1. Entries must be added to the registry by hand. Each package gets an entry line in the registry and a subdirectory in the registry directory with `Versions.toml, Deps.toml, Compat.toml, Package.toml`.

2. Any inconsistencies in the entries will cause the registry to be ignored by `Pkg`. So this approach is fragile.

3. Each time a package is changed or updated, the registry needs to be augmented by hand, including all dependencies.

Fredrik Ekre has an example in his github repo. At this time, the approach is not really workable.

6.4 Multiple Modules in one Package (1.1)

The cleanest approach is sub-modules. One can still `import Foo.Bar` to only use the sub-module (especially for testing). In the test function, non-exported functions can be called as `Bar.f()`.

6.5 Testing a package (1.1)

Activate the package by issuing `activate .` in the package's directory (not in `src`). Then type `test`.

Placing test code inside a module:

- This can be useful when the test code defines `structs` that one would like to be able to modify without having to restart `Julia` all the time. Note that objects defined in tests are no longer visible once `Pkg` is exited.

- Place the module definition into `test`. Add `push(LOAD_PATH, @__DIR__)`. This has to be done in each module. Not elegant.

7 Performance

The compiler does not optimize out `if false` statements. Hence, defining a constant that switches self-testing code on and off does not result in no-ops. Of course, the overhead is quite small.

7.1 Profiling

The output generated by the built-in profiler is hard to read. `ProfileView` does not compile (1.1).

`StatProfilerHTML` is a good alternative (1.1). It provides a flame graph with clickable links that show which lines in a function take up most time.

7.2 Type stability

One can automate checking for type stability using the `code_warntype()` function. Example:

- For function `foo(x)`, call `code_warntype(stdout, foo, (Int,1))`.

- This can be written to a file by changing the `IO` argument.

- It generates output even if no issues are found.

- The amount of output generated is overwhelming. Signs of trouble are `Union` types, especially return types (at `Body:`).

8 Remote Clusters

How to get your code to run on a typical Linux cluster?

- Get started by writing a simple test script (`Test3.jl`) so we can test running from the command line.
- Add the Julia binary to the `PATH` using (on Macos, editing `~/.bash_profile`):

  ```
  PATH="/Applications/Julia-1.1.app/Contents/Resources/julia/bin:$PATH"
  ```

- Then make sure you can run the test script with
  `julia ■full/path/to/Test3.jl■`

Now copy `Test3.jl` to a directory on the cluster and repeat the same.

- You may need to add the Julia binary to the path. On Longleaf (editing `~/.bash_profile`):
  `export PATH="/nas/longleaf/apps/julia/1.1.0/bin:$PATH"`
- Then run `julia "julia/shared/Test3.jl"`

Now run the test script via batch file:

```
sbatch -p general -N 1 -J "test_job" -t 3-00 --mem 16384 -n 1 --
mail-type=end --mail-user=lhendri@email.unc.edu -o "test1.out" -
-wrap="julia julia/shared/Test3.jl"
```

8.1 Generate an ssh key

This allows log on without password. Instructions on the web.

Now you can use the terminal to log in with `ssh user@longleaf.unc.edu`.

8.2 Rsync File Transfer

A reliable command line transfer option is `rsync`. The command would be something like

```
rsync -atuzv "/someDirectory/sourceDir/" "username@longleaf.unc.edu:someDirectory"
```

Note: The source dir should end in “/”; the target dir should not.

8.3 Git File Transfer

The following, described here does not work for me. Run once:

`git remote add server ssh://lhendri@longleaf.unc.edu/nas/longleaf/home/lhendri/jul`

After

`git init --bare` (on the server side)

`git push server master` (in the repo dir)

I have a directory full of files the meaning of which I don't understand.

8.4 Running code on the cluster

`longleaf` uses slurm. This is equivalent to running a `julia file.jl` command from the terminal.

To run a package, the file should contain:

```
using Pkg
cd("directory/where/package")
Pkg.activate(".")
using MyPackage
MyPackage.run_things()
```

Note: It helps to have all directories hang off the same base directory on both machines. Then, in the code, data files can be located with paths that are relative to `@__DIR__`.

9 Types (1.1)

I find it easiest to write model specific code NOT using parametric types. Instead, I define `type aliases` for the types used in custom types (e.g., `Double=Float64`). Then I hardwire the use of `Double` everywhere. This removes two problems:

1. Possible type instability as the compiler tries to figure out the types of the custom type fields.
2. It becomes possible to call constructors with, say, integers of all kinds without raising method errors.

9.1 Constructors (1.1)

Constructing objects with many fields:

- Define an inner constructor that leaves the object (partially) uninitialized. It is legal to have new(x) even if the object contains additional fields.

9.2 Loading and saving (1.1)

`using FileIO` and extension `.jld2` automatically saves in `jld2` format. This can save used defined types.

Loading user defined types is more complicated. All modules needed to construct the loaded types need to be known in the loading module and in `Main`. See Issue 134. It is not possible to use `Core.eval(Main, :(using Module))` for unclear reasons.

Implications:

1. Each user defined type needs its own `load` function.
2. All dependencies need to imported into Main `■by hand■ for each loaded object.`

An alternative is `BSON.jl`. It has the same limitation.

One could save the `ParamVectors` in each object and reconstruct the object from those (recursively). This, of course, only works for objects that can be constructed from `ParamVectors`. Each `ParamVector` could be stored as a `Dict{Symbol, Any}`. But even easier: store the `ParamVectors` directly. Constructing them after loading only requires `modelLH`. The approach would then be:

1. Collect the `ParamVectors` from all model objects into a `Dict{Symbol,ParamVector}`. The symbol identifies the associated model object.
2. Save the `Dict`.
3. In `Main:` using modelLH, so that loading works.
4. Function that loads the model:

(a) Construct the model object with arbitrary default values.

(b) Load the `ParamVector`s.

(c) Sync each `ParamVector`'s parameters into the correct model object. Essentially, the model object needs a constructor that accepts a `ParamVector`.

10 Unit Testing (1.1)

All codes should be in `modules` because code in `Main` runs slower, pollutes `Main`, and it harder to `revise`. This also applies to test code.

However, placing the `@test` or `@testset` portions into the test module causes them not to run sometimes (why?). It also implies that `using` the test module runs all tests, which is generally unwanted. I therefore place the `@test` code into a separate file (not inside a `module`).

Errors in the code to be tested (but not caught by `@test`) cause the entire test run to crash. Preventing this requires all tests to be enclosed in a `@testset`. A sequence of `@testset` does not do the trick. An error in one prevents all others from being run. Nested `@testset`s produce nested error reports (nice).

`@test` statements can be placed inside functions. To preserve result reporting, the function should contain a @testset and return its result.

11 Workflow (1.1)

`Revise` is key. It is now possible to simply use `using` on any `module` once. `Revise` then automatically keeps track of changes. Using `includet` creates problems for me.
