Notes on the Julia Programming Language

Lutz Hendricks

UNC Chapel Hill

July 6, 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.

2 Arrays (1.1)

2.1 Indexing

Extracting specific elements with indices given by vectors:

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

```
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]
```

3 Modules

3.1 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‘.

4 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.

4.1 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.`

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`.

5 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`).

6 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.
