

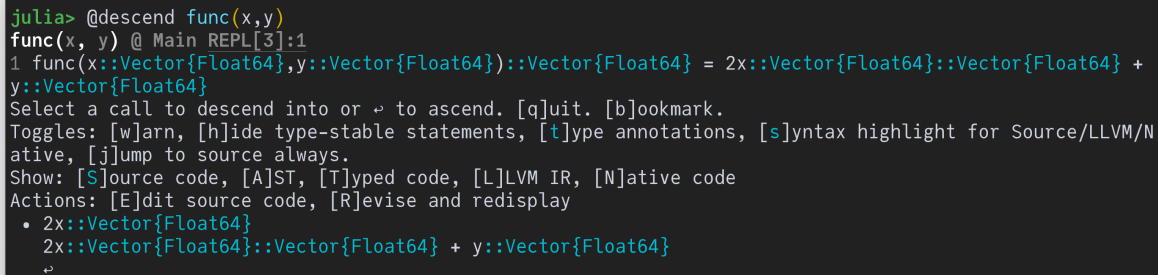
Activity 4 - Cthulhu (30 minutes)

This is based on “Julia for HPC course at TU Delft” by Carsten Bauer.

Cthulhu.jl

Cthulhu.jl is an interactive, more powerful generalization of the introspection macros. Among other things it has the following features:

- Allows easy switching between code representations (syntax, typed, native, ...).
- **Recursive application possible(!)** (i.e. introspecting a function that is called within a function within function ...).



```
julia> @descend func(x,y)
func(x, y) @ Main REPL[3]:1
1 func(x::Vector{Float64},y::Vector{Float64})::Vector{Float64} = 2x::Vector{Float64}::Vector{Float64} +
y::Vector{Float64}
Select a call to descend into or ↵ to ascend. [q]uit. [b]ookmark.
Toggles: [w]arn, [h]ide type-stable statements, [t]ype annotations, [s]yntax highlight for Source/LLVM/N
ative, [j]ump to source always.
Show: [S]ource code, [A]ST, [T]yped code, [L]LVM IR, [N]ative code
Actions: [E]dit source code, [R]eview and redisplay
• 2x::Vector{Float64}
  2x::Vector{Float64}::Vector{Float64} + y::Vector{Float64}
  ↵
```

Exercise

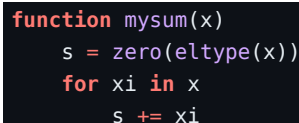
In this exercise, you will make yourself familiar with the basics of Cthulhu.jl, a tool to interactively introspect Julia code.

Note:

- Cthulhu only works in the REPL and not in Jupyter! Hence, you must do this exercise in a Terminal!
- Please run `julia --project` to start Julia REPL!

Part I: Introspection basics

Consider the following Julia code:



```
function mysum(x)
    s = zero(eltype(x))
    for xi in x
        s += xi
    end
end
```

```

end
return s
end

function entryfunc(x)
    if all(i->i>0, x)
        return mysum(x)
    else
        return mysum(abs.(x))
    end
end
end

```

1. Load Cthulhu (using `Cthulhu`) and use `@descend entryfunc(rand(3))` to start the analysis.

The output that you see is structured as follows:

- First, it shows the name of the current function and its source code location (`REPL[i]:j` if the function is defined in the REPL instead of a file).
- Second, it displays the source code of the function which, by default, is augmented with type information (from Julia's type inference).
- Third, it shows you modes/options that you can toggle.
- Lastly, it shows you a list of possible function calls within the current function.

For the last point above, it is important to note: **Cthulhu doesn't run your code and therefore has no runtime information but only static (type) information.** It doesn't know if the given `x` fulfills the if-condition or not. (Contrast this to a regular debugger.)

2. Press `t` on your keyboard to toggle the type annotations in the source code.
3. Let's now switch between different code representations. Type `N` (i.e. press `SHIFT+n`) to switch to native code. Afterwards, type `L` (i.e. `SHIFT+l`) to get the LLVM IR. Finally, type `S` (i.e. `SHIFT+s`) to get back to the source code level.
4. Assume you actually want to investigate the native code for the `mysum` function in the first branch (where all elements of `x` are positive). Use the up-/down-arrow keys to select `mysum(x)` (or `mysum(x::Vector{Float64})` if type annotations are active) in the list at the bottom (the dot on the left indicates the currently selected call) and press `ENTER` to step one level down. On the new level, switch to native code representation.
5. Optional/Bonus: See how the native code changes for integer input, i.e. `rand{Int, 3}`. Can you relate this to SIMD vectorization (if you already know what that means)?

If you're done with this part, you can exit Cthulhu by pressing `q`.

Part II: Code path traversal

We've seen above that, for a given entry point (i.e. function call) Cthulhu allows us to interactively traverse the underlying code paths. This is particularly useful to understand and navigate unfamiliar code bases.

Example 1: `all`

1. For the example code given above (Part I), navigate into the `all` function call.

What you see might look strange: The source code seems incomplete. However, note that Cthulhu gives us a hint (Info note) what we should do:

[Info: This method only fills in default arguments; descend into the body method to see the full source.

2. Do as Cthulhu says and navigate down one level (into `#all#...`).

Now we see complete source code but it is not very informative as it only redirects to another function `_all(...)`.

3. Step down further (i.e. into `_all(...)`).

We finally see the actual meaningful implementation of the `all` operation.

Example 2: BLAS/LAPACK calls

For most linear algebra operations Julia uses BLAS/LAPACK under the hood. However, don't take my word for it but figure it out yourself 😊

1. Use Cthulhu to descend into a matrix-matrix multiplication (i.e. `@descend rand(5,5) * rand(5,5)`) and try to traverse down to the underlying BLAS/LAPACK call. (Hint: focus on `mul!` calls)
 - Additional: Does Julia always call BLAS/LAPACK or can you see a branch in the source code that leads to a pure-Julia implementation? If so, where is it defined?

Part III: Recursive `@code_warntype` (Bonus)

Cthulhu is a very helpful tool to locate type inference issues, especially if they happen deep inside of your code rather than at the top-level. Essentially, you can use it like `@code_warntype` but interactively and recursively.

Consider this variation of the example above:

```
function myfunc(x)
    s = zero(eltype(x))
    for xi in x
        tmp = rand(["hello", xi, false])
        if tmp isa String
            s += 0
        else
            s += tmp
        end
    end
    return s::Float64
end

function entryfunc2(x)
    if all(i->i>0, x)
        return myfunc(x)
    else
        return myfunc(abs.(x))
    end
end
```

1. Use Cthulhu to figure out if `entryfunc2(rand(3))` has type inference issues.
 - Hint: Make sure to activate type annotations in source code mode (`t`) or switch to the typed code representation (`T`). Then, activate the warn option (`w`) to highlight problematic occurrences in red/orange.
2. Next, step one level down into the `myfunc` function. Does this function have type inference issues?