CMPT 383

Assignment 2

Yifan Liu

301297560

yla455@sfu.ca

Introduction

With the stagnation of *Python*, a new hot competitor has emerged which is *Julia*. The popularity of Python is still firmly supported by computing scientists, data scientists, and experts in the field of artificial intelligence. But *Python* is slow and needs many efforts to debug. Also, there will still be many errors until run time despite the previous tests done. These weaknesses are enough to irritate. Therefore, more and more programmers are adopting other languages. *Julia* becomes the biggest winner since it is very suitable for mathematical and technical tasks in these field.

When people create a new programming language, their purpose is to retain the good features of the old language and fix its bad features. In this sense, the purpose of creating *Python* in the late 1980s is to improve ABC (Abstract Base Class). *Python* still retains the good features of ABC, for example, readability, simplicity, and friendliness for beginners. ABC paves the way for *Python*, and *Python* also paves the way for *Julia*. [1]

Julia is a high-level, high-performance, general-purpose, and dynamic programming language which can be used to write any kinds of application. It is originally designed to meet the needs of high-performance numerical analysis and computational science. *Julia* can also be used for client-side and server-side web application, low-level system programming, and protocol language. [2]

Type System

The type systems are traditionally divided into two different categories including static typing and dynamic typing. Static typed programming languages are those in which every program expression must have a computable type before the execution of the program. They require extra steps for explicit declaration and initialization of variables before they are used. Every variable has a unique type which cannot be changed within its scope. A variable with a certain type can only refer to the value with that type. On the other hand, dynamic type systems do not declaration before every expression where nothing is known about the type until the values manipulated by the instructions are available. A variable has not type, but the value it refers to do, therefore, the variable can be reallocated actual values with other types. [3]

Julia is dynamically typed but gains some benefits of statically typed languages in which a variable can be defined as a certain type. The uniqueness of the design of *Julia* includes a type system with parametric polymorphism in a dynamic programming language and its multi-dispatch core programming paradigm. If no type is explicitly declared, the variable will be an *Any* type that is an abstract supertype of all types. *Julia* is polymorphic with the ability to write the code that can operate on different types. [4]

The "::" operator is used to combine with type names to declare the type of variables as shown in the following. These will create an integer and a character type variable with initialized value.

(1) number::Int = 3
(2) character::Char = `y`

In *Julia*, there are plentiful built-in types available, such as *Int, UInt* (unsigned integer), *Float, Char, String, Bool, Array, Tuple, Dictionary, Set,* and *Any.* The first six types are similar with other languages. And *Any* typed variables can be any type in *Julia.* [5]

An array is a container storing several variables with a certain type. To declare an array, the type, dimensional or number of values, and initialized values for every entry should be available. The first line of code constructs an initialized N-dimensional and N-by-N array containing elements of type T. The other is all the same but leads to a n-by-m array. [6]

```
(1) arr = Array{T}(undef, N)
(2) arr = Array{T, N}(undef, n, m)
```

Tuples in *Julia* are an immutable collection of distinct values of same or different datatype reside within a pair of parentheses and separated by commas. Tuples are more like arrays in *Julia.* The difference is that the array can only store values of similar datatypes. Since the tuple is immutable, the value of the tuple cannot be modified. As a dynamically typed language, a variable can be redefined with another values. Meanwhile, a named tuple works like a dictionary. Accessing the value associated with a name in a named tuple can be done using field access syntax, e.g. `tup3.a` or `tup3[:a]`. [7]

```
(1) tup1 = (1, 2, 3, 4)                    # tuple with the same type
(2) tup2 = (1, 2.0, 3, "Hello")            # tuple with different types
(3) tup3 = (a = ("A", 10), b = ("B", 20))  # named tuple
```

Dictionary in *Julia* is a collection of key-value pairs where every key is associated with one value and unique. The *Any* typed key and value are not necessarily be the same. This implies that a String typed key can hold a value of any type such as Int, String, and Float. [8]

```
(1) dict = Dict("pi" => 3.14, "e" => "2.718", 1 => 1)
(2) delete!(dict, "pi")          # delete the pair with key "pi"
(3) dict["golden"] = 0.618       # add a pair into the dictionary
```

Sets is a collection of *Any* typed values without duplicated values where all element in a set is unique. Between sets, it is possible to get the union, intersect, and difference by calling the built-in functions. [9]

```
(1) set = Set(["Jim", "Pam", "Jim", 1])
(2) union, intersect, setdiff(set1, set2)
```

Besides, *Julia* support some composite types which are called records, structs, or objects in other languages. A composite type (user-defined type) is a collection of named fields that contains infinite number of variables of built-in types and are treated as a single type. Composite types are introduced with the *struct* keyword followed by its member variable declarations with types defined or not. Since a composite type is immutable, value cannot be changed whenever it is initialized. [5]

```
struct Customer
    name::String
    id::Int
```

```
    balance::Float64
end
```

Finally, static or dynamic type and strong or weak type are two different concepts. *Julia* is a strongly typed language in which a variable cannot change its type to suit the current situation automatically. In this case, "2" will always be treated as a string. Without operator overloaded or manual conversion and intervention, a string is not allowed to add with an integer. [10]

```
X = 1
Y = "2"
Z = X + Y
```

Memory Management

Memory management is an important function of an abstract machines' interpreter to deal with the memory allocated for the data programs using. There are both static and dynamic memory management. Memory allocation can be divided into three different parts, static, stack and heap. Static memory is generated during compile time and stores strings or global variables as usual. Stack memory happens when calling a function. Whenever a function is called, the data will be stored on contiguous blocks of memory, and the number of blocks to be allocated is known by compiler. And whenever the function is ended, the stack memory will be deallocated automatically and freely to be used. Heap allocation is a dynamic allocated memory during execution of instructions of dynamically allocated objects such as malloc in C, new in C++, and all objects in Python. Memory leak might happen if the allocated memory is not freed and handled well. [11]

In *Julia*, every value of fundamental types, like Int, Char, and Float is stored in the static memory. There is no pointer in *Julia.* The memory allocated in static and stack memory can be deallocated automatically and completely without causing memory leak in heap memory. In addition, some tools of garbage collector are available for heap memory.

Other Features

An interesting feature is partial application for functions in Julia. A function can be returned as a lambda function with some arguments fixed and initialize. Then, the rest of arguments will be available from further inputs. [12]

```
function makeMultiplier(num)
   return function(x) return x * num end
end
mult3 = makeMultiplier(3)
println(mult3(6))
```

In addition, Julia has a multi-dispatch core programming paradigm, that is a function can have the same name and same numbers of argument with others but the types of all arguments.

```
describe(n::Integer, m::Integer) = "integers n=$n and m=$m"
describe(n, m::Integer) = "only m=$m is an integer"
describe(n::Integer, m) = "only n=$n is an integer"
```

This is more like pattern-matching in Haskell or called “type-matching”. The three different versions of describe have n and m with different pairs of type. The program will match the pair of type of n and m in the order. If only one of n and m is in Integer type, then the first function do not meet the requirements and the last two function will be check and execute the corresponding one later. [13]

Comparison with Python

As I mentioned above, Python is widely used by the experts in the field of artificial intelligent, data science and the area of mathematical analysis and computation. Julia is designed to meet the needs of high-performance numerical analysis and computational science. There are several differences between them, including speed, versatility, community, and libraries.

Julia is as fast as C. It is compiled at Just-In-Time or runtime using the LLVM framework. Julia is excellent for numerical computing and takes much lesser time to compile with big and complex codes than Python. Also, python has easy readability and a code friendly syntax. Meanwhile, Python has a large community for developers to share ideals and resolve problems and doubts. As a new growing language, the community for Julia is at a very nascent stage. Finally, the packages of Julia are not well maintained but it can interface with libraries in C directly. On the other hand, Python has a bunch of open-source libraries which makes every task work easily with some lines of code. [14] [15]

Conclusion

Julia is a language used for numerical analysis and data computation. Python pave the way for Julia. With multiple advantages and features, Julia’s recent popularity is well explained. However, it still has some drawbacks against python. Mixture between Julia and Python will make work better.

References

[1] https://reinout.vanrees.org/weblog/2018/04/25/origin-of-python-abc.html

[2] https://www.infoworld.com/article/3284380/what-is-julia-a-fresh-approach-to-numerical-computing.html

[3] https://dev.to/jiangh/type-systems-dynamic-versus-static-strong-versus-weak-b6c

[4]https://nullprogram.com/blog/2014/03/06/#:~:text=The%20defining%20characteristic%20of%20static,abstract%20supertype%20of%20all%20types.

[5] https://docs.julialang.org/en/v1/manual/types/

[6] https://docs.julialang.org/en/v1/base/arrays/

[7] https://www.geeksforgeeks.org/tuples-in-julia/

[8] https://www.geeksforgeeks.org/julia-dictionary/

[9] https://www.geeksforgeeks.org/sets-in-julia/

[10] https://medium.com/@cpave3/understanding-types-static-vs-dynamic-strong-vs-weak-88a4e1f0ed5f

[11] https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

[12] https://discourse.julialang.org/t/partial-functions/2206

[13] https://riptutorial.com/julia-lang/example/10467/introduction-to-dispatch

[14] https://docs.julialang.org/en/v1/manual/noteworthy-differences/index.html

[15] https://www.techaheadcorp.com/blog/julia-vs-python/