T Julia: M ,

Abstract

E ;

Description of the Julia programming language

Julia is a high level, multi-paradigm, dynamically typed, programming language. It is aimed at fields such as numerical analysis, computational science, while also being well suited for general purpose programming.

Focusing on providing Ruby's dynamic types, the syntactic simplicity of Python and C-like performance, Julia could prove to be the leading language in high performance computing, and an indispensable tool for research in scientific and engineering fields.

Julia is released under the MIT license, therefore is free and open source.

Feature overview

Julia is JIT-compiled and garbage collected and uses multiple dispatch. It is designed with high performance in mind, being comparable to much lower level languages, such as C. In addition, parallel execution and distributed computing are first class citizens. Other key components, include macros and metaprogramming support, a built in package manager, seamless interop with C and Fortran and a highly sophisticated compiler, able to generate specialized code, depending on argument types.

History

Julia was designed by Viral B. Shah, Jeff Bezanson, Stefan Karpinski and Alan Edelman. Released in 2015, after first being revealed on Valentine's Day of 2012. Its user base has been growing exponentially, while its popularity landed it at the top 50 of the TIOBE Programming Community Index (www.tiobe.com).

The two languages problem

Julia's design, came as an answer to the two languages problem, faced by modern data scientists; Writing a code prototype in a dynamically typed language, to verify a working solution, but then having to rewrite a whole new implementation in another, statically typed language in order to achieve acceptable performance.

One can easily implement some algorithm or conceptual solution in Julia. Its great advantage in comparison to languages like Python, is that the very same code, can achieve the performance of highly optimized, machine specific code (thanks to LLVM), only by introducing very minor changes, in the form of type declaration for a method's arguments. This makes the code extremely easy to optimize, even for users with little understanding of low level architecture.

This is achieved, thanks to Julia's versatile and advanced compiler, that can produce LLVM IR (intermediate representation), specialized on the types of the parameters of each calculation. If the types are not known in advance, the generated assembly may not make any assumptions about the arguments' memory representation, and while perfectly working, it is sub optimal. In case that constraints are enforced on the types, the compiler is smart enough, to take advantage of them, and generate assembly similar to that of the statically typed C. As a result, you have all of the benefits of a statically typed language, both in type safety and performance, as an opt in feature, allowing the liberty and ease of use of a dynamic type system wherever speed is not a concern.

Platforms

Julia is JIT-compiled with an LLVM backend. It can generate native code for all of the major modern platforms:

- Windows
- Linux
- Mac OS
- FreeBSD

The main architectures supported are x86-64 (AMD64) and x86, while there is experimental support for ARM, AARCH64, and POWER (little-endian).

Language Features in depth

Type System

Julia features a dynamic, nominative and parametric type system. Every type is abstract, except concrete types, which are final. This means that the only types that can be instantiated, are unable to have subtypes, while their supertypes cannot be instantiated. That distinction from other object oriented languages' type systems, implies that inheritable types are only used to infer common behavior, not implementation details such as data (much like interfaces in other OO programming languages); Only the types which are at the bottom of the type tree, can have implementations and specific memory representation.

As the type system is dynamic, usually it is not necessary to explicitly declare types. Variables are just a way to name (or reference) data, and do not have types, only the values that they can hold have types. As a result of those two

points, most programs do not need to declare any type information, leading to adaptable code, that can behave robustly, no matter the input.

Multiple Dispatch

Short introduction to multiple dispatch

Possibly the most attractive feature of Julia, is multiple dispatch, the ability to dispatch a function, based on all of its parameters.

Multiple dispatch, is a generalization of single-dispatch polymorphism, present in most object oriented programming languages.

Commonly, object oriented languages, have a feature called subtyping (usually implemented through inheritance), where one parameter type, can be substituted by its subtypes, types that are more specific and lower in the type hierarchy tree. In every function call, the specific function that will be executed, is selected based on the type of one of the arguments (most commonly the first one):

```
objectOfTypeA.method()
```

The above, in languages like C++, C#, Java, etc., assumes the existence of a function named `method`, which implies an argument of some type, named `objectOfTypeA`. The same identifier, `method`, can also exist in other types, or even subtypes of the aforementioned object's class. Thus, in order to differentiate them, the type of the object "calling" the method, is of significance; this is usually implemented as the first argument of a function like so:

```
method(A parameter1)
```

(where A is the type of the first parameter), in C like code. That is how a function is dispatched (its code is selected for execution), in single dispatch languages.

In contrast to that, multiple dispatch languages, do not have any significant parameters, whose types are treated differently; They are dispatched based on the derived type of every argument. For example:

```
method (T_1 parameter1, T_2 parameter2)
```

```
method (T_1 parameter1, T_3 parameter2)
```

Those two methods, are different, even though they have the same name, as the type of their second parameter is T_2 and T_3 respectively. If one would call those methods in the following way:

```
method (object1, object2)
```

The type of object 2 must be determined in order to decide which method to route this statement to. If object 2 is of type T_2, the first method will

be dispatched. In the special case where T_3 is a subtype of T_2, if the object is of type T_3, the second method will be selected, as it is more specific. This is something that can only occur for a special argument in single dispatch languages, while in multiple dispatch, any number of arguments enjoys the same polymorphism.

Method calls in Julia

Given the following call:

```
method(argument1, argument2)
```

The existence of a method named `method` is assumed, whose return type is `T`, and takes two arguments, of types `T1` and `T2`, named `argument1` and `argument2` respectively.

According to the Julia spec, every function, is a member of a certain type; This is the return type of the function, in our example `T`. Type T, has a method table, which contains all the functions associated with it.

When the JIT compiler encounters that statement, it will dispatch a method through the following algorithm:

1. Determine the associated type with `method`, as `T`
2. Lookup in the function table of `T`, for a method with that name (`method` in the example)
3. Select the one who has two parameters of types `T1` and `T2`

The compiler, will generate specific code, only for those three aforementioned types, if it doesn't already exist. In case that we call the same method with a new argument:

```
method(argument1, argument3::T3)
```

of type `T3`, unrelated to `T2`, the previous method, is not applicable, therefore a new one will be dispatched (and compiled if needed).

In the case where the argument types are not specified -as Julia is a dynamically typed language and does not require type declaration-, and no assertion can be inferred about the runtime types, multiple dispatch cannot be utilized, because only one method can be generated, that can make no assumptions about its arguments. The generated assembly is much less efficient, as many precautions must be taken for memory access, since the types are uncertain, and their size is unknown, as well as them being values or references to values.

Parallel and Distributed Computation

Linear Algebra

Optimizing Julia Code

Profiling