Umeå University
Department of Computing Science

February 24, 2017

Programing Languages VT17

A brief overview of Julia the programming language

Olle Larsson (`c14oln@cs.umu.se`)
Johannes Umander (`c14jur@cs.umu.se`)
Niklas Bäckström (`c14nbm@cs.umu.se`)
Robin Lindgren (`c11rln@cs.umu.se`)

Version 1

julia

Supervisors
Filip Allberg
Jan-Erik Moström

Contents

1 Background

Julia is a high-level, open source, high-performance, dynamic, multi-paradigm programming language for numerical and scientific computing. Julia is a relatively new programming language that has been developed by the MIT students Jeff Bezanson, Stefan Karpinski, Viral Shah and Alan Edelman. They introduced the first version of Julia to the public in February 2012 which had at that point been in development for 2.5 years.
The students were all involved in technical computing, using languages such as MATLAB, Python, Ruby, Lisp and Perl. Multiple programming languages had to be used because each of the languages were good at solving one set of problems but worse at solving others. With Julia they wanted a single language that could replace the others.
They wanted a language with mathematical notation like MATLAB, one that could solve general problems like Python, one with the dynamism of Ruby, one with macros like Lisp and the ease of string processing found in Perl.

"We are greedy: we want more."

Their demands did not end there, they also wanted Julia to perform as fast as C with extensive support for parallel computing. In the end they basically wanted a language that could do it all while still being simple and clean to program[1].

The current stable version of Julia is `v0.5.0` as of February 24, 2017.[2] The official website for Julia (`http://julialang.org`), lists these as the primary features of Julia:

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Dynamic type system: types for documentation, optimization, and dispatch
- Good performance, approaching that of statically-compiled languages like C
- Built-in package manager
- Lisp-like macros and other meta-programming facilities
- Call Python functions: use the PyCall package
- Call C functions directly: no wrappers or special APIs
- Powerful shell-like capabilities for managing other processes
- Designed for parallelism and distributed computation
- Co-routines: lightweight "green" threading
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- MIT licensed: free and open source

As stated above Julia is open source, and all the source code is available at https://github.com/JuliaLang/julia.

Julia's base library is largely written in Julia itself, but it also incorporates open source C and FORTRAN libraries. The Julia developer community can contribute with external packages through Julia's built in package manager.[3]

In Julia all values are objects, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a function to use by using something called multiple dispatch. Memory management is taken care of by a garbage collector.

Julia uses a LLVM based JIT compiler. This gives Julia the ability to reach the performance levels of compiled languages such as C, while also having some of the advantages of interpreted languages. It also makes Julia's programs platform-independent.

Julia comes with a full featured command-line REPL, Read-Eval-Print-Loop, built on-top of the executable "julia" The REPL allows for quick evaluation of Julia statements. The REPL is started by simply calling `julia`.

2 Design

Julia is a multi-paradigm language which combines features of imperative, functional, and object-oriented programming languages. This section is an overview of some some basics parts along with core parts of Julia which makes it unique and different from other high performance languages.

2.1 Syntax

Julia borrows a lot of its basic syntax from one of its main competitor, MATLAB, which is arguably one of the most readable programming languages.
The flow control in Julia is the same as most other commonly used programming language, but a difference is that the switch-case is not supported.
Like MATLAB, Julia's is prioritizing mathematical concepts in it's syntax such as linear algebra, for example; in Julia, array indexing starts at 1.
The following code is a simple function in Julia that generates an array with the first N Fibonacci numbers.

```
function fibonacci(N)
  fibo = zeros(Integer, (N, 1))
  for i=1:N
    if i == 1 || i == 2
      fibo[i] = 1
    else
      fibo[i] = fibo[i-2] + fibo[i-1]
    end
  end
  return fibo
end
```

A major difference between MATLAB and Julia is how functions work. The programmer can only create one function with a given name in MATLAB, and if more behaviors are needed, the programmer has to parse the arguments in the function. Multiple functions can have the same name in Julia, and Julia's powerful multiple dispatch system is used to invoke the correct behavior based on the types specified. This feature makes it easier to write code, since functions like `+` can be used for many different types.

A variable in Julia is simply a name associated, or bound, to a value. Variable names are case sensitive. Julia allows for redefining predefined constants such as the mathematical constant π.[4]

2.2 Functions

Functions are a core part in most languages, and that holds true for Julia as well. In Julia functions are somewhat different from many other imperative, object-oriented languages. A function in Julia is an object that maps a tuple of argument values to a return value. Functions can alter the state, and be affected by the global state of the program, which means that they are not purely mathematical functions.

Arguments which are passed to a function are not copied, which means that function arguments themselves acts as new variable bindings, and the value that they refer to is identical to the value passed. Alterations to mutable values such as Arrays within a function will be visible to the caller, which is the same behavior found in many other dynamic languages such as Python, Perl and Lisp.

Unlike MATLAB, and many other languages, Julia has taken a different approach on how to return multiple values from a function. Remember that a function maps a tuple of arguments to a return *value*, and not return *values*. Julia simulates multiple return values by letting the singular return value be a tuple, if needed. You can construct or deconstruct a tuple without using parentheses, which might provide the illusion that multiple values are actually returned. If the `return` keyword is omitted from a function, then the value of the last expression in the function body will be returned. Of course the only time it is really useful to omit `return` is in purely linear functions. Anywhere where there are some type of flow control, the `return` keyword is paramount.[5]

In Julia one cannot declare what type of a value a function will return, the reason why declaring a return type is not possible can very simplified be put as: One can either have a dynamically typed language, or one can have static return types, because once one does not have static return types, getting the semantics right becomes much harder.[6]

In most cases operators are just functions, with the exceptions being special evaluation semantics such as `&&` and `||`. They simply cannot be functions because `short-circuit-evaluation` requires that their operands are not evaluated before the evaluation of the operator.

In Julia Unicode characters can be used for function and variables names.

A simple function in Julia can be written as follows:

```
function mult_and_add(x,y)
    x*y, x+y
end
```

Alternatively it can also be written as:

```
mult_and_add(x,y) = x*y, x+y
```

The function `mult_and_add` takes two arguments, and returns both the sum, and multiplication of the two arguments as a tuple if the function is called in the following way:

```
a = mult_and_add(1,2)
```

`a` will be bound to the tuple: `(2,3)`.
The function can also be called in the following way:

```
mult, sum = mult_and_add(1,2)
```

`mult` will be bound to the value 2, and `sum` will be bound to the value 3.

2.3 Type system

Julia's type system was, and is, designed to be powerful and expressive while still being intuitive and clear. The type system which Julia is using is a dynamic one. Although dynamically typed Julia also gains some of the advantages of statically typed systems, by actually letting you be able to indicate that a certain value are of a certain type. One can write Julia code without explicitly using types, however some kinds of programming becomes clearer, faster and more robust, if types are declared. It is important to know that only values have types and not variables, remember that variables are simply names bound to a value.[7]

Traditionally dynamic languages has a disparity between built in types and user defined types, where built in ones are much faster then the user defined ones. However this is not the case in Julia since there is no meaningful distinction between a user defined types and built in types.[8]

The `::` operator is useful when it comes to handling and checking types in Julia.
The `::` operator can have multiple different meanings depending on where it is located.

When `::` is appended to an expression, then it is read as "is an instance of" and can be used anywhere to ensure that the value of an expression on the left is of the correct type.
For example, `x::T`, if `x` is of type `T` then the expression evaluates to `T`, else an error is raised.

When `::` is appended to a variable, then it declares the variable to always have the specified type. Every value assigned will be converted to the declared type. This can be compared to a type declaration in a statically typed language such as C.
For example, `x::Int32 = 10`, makes the value of `x` be of type `Int32`.

When `::` is appended to a function, that means that the output of that function basically will be converted to the annotated type.
For example, `foo(x)::Int32 = x`, converts the output `x` to `Int32` if possible, otherwise an error is raised and the expression returns no value.

Finally the `::` operator can be used when defining a function, to constrain the types of parameters it is applicable to.
For example, `foo(x::T) = x`, `foo` only applicable to values which are of type `T`.

2.3.1 Type hierarchy

In Julia there are abstract types, and concrete types. An abstract type cannot be instantiated. They hold particular value in that they allow the construction for a hierarchy of types, where the hierarchy is of a tree structure. One can think that abstract types indicates relationships between concrete types, and can be thought of as similar to Interfaces in Java. The root of the type tree structure is the abstract type `Any`. `Any`'s children are subtypes to `Any`. The leaves in the tree structure are called concrete types. Concrete types can be instantiated, and cannot have subtypes. In Julia concrete types are objects, which abstract type are not since they cannot be instantiated.[9]

The following is a small illustration of the type hierarchy, from the root `Any` to the concrete type `Int64`:

```
* Any →
*         Number →
*                Real →
*                       Integer →
*                              Signed →
*                                     Int64
```

The type tree in Julia is huge, with well over 500 types. Do not let the number of types scare you away from trying Julia, since a large portion of the types are tailored for advanced usage and people who know their mathematics.

The benefit of this hierarchy is that it allows for construction of generalized code, instead of restricting it to a single type.
A concrete example of this would be the following function:

```
function foo(x::Number)
    return x + 1
end
```

The type of the parameter `x`, is the abstract type `Number`, which means that `foo` is applicable for any subtype to `Number`. So one might call this function with a value of type `Int32` or `UInt8` etc.

The `<:` operator can be used to check if a type is a subtype of another. For example, `Number <: Any`, which returns true if `Number` is a subtype of `Any` which it is.

Concrete types and abstract types are internally represented as the type `DataType`. So, for example if one were to check the type of `Int64` or `Number`, using the `typeof()` function, one would see that both of these are of type `DataType`.

2.3.2 Parametric types

A powerful and key feature in Julia's type system is that it is parametric. This simply means that types can take parameters, and this in return can introduce a whole new family of new types. One type for each possible combination of parameters.

The following code declares a new parametric composite(similar to a `struct` in C) type, `rectangle{T}`:

```
type rectangle{T}
    height::T
    width::T
end
```

`T`, is the parameter which can be of any type, like `Float64, Int64` etc. So there are an infinite amount of possible types of `rectangles`.

To instantiate a composite type a constructor is used. All types has got a default constructor, but of course there is the possibility to create your own constructor(s) as well. So to create a `rectangle`, one can write: `rectangle{Int64}(2, 3)`, which will create a rectangle in which height will have a value 2 and width `3`, and their value are of type `Int64`.

The `<:` operator can be used to constrain which type a parameter is applicable to. For example `{T<:Number}`, means that the type parameter `T` must be a subtype of `Number`.
However, abstract type parameters can not be sub-typed, which is a drawback. For e.g. `Array{Integer}<:Array{Number}` is false while `Array{Integer}<:Array` and `Integer<:Number` is true.

2.4 Multiple dispatch

Multiple dispatch is one of Julia's most central feature. To be correct the full terminology would actually be `Dynamic` multiple dispatch, although from now on it will only be referred to simply as multiple dispatch. In order to grasp what multiple dispatch is, you first need to know what a method is. A method is a definition of a behavior for a function. And so a function may have multiple methods, where each method has different argument types and, or different arguments.

The following are examples of methods for the function `foo`:

```
foo() = "Empty input"
foo(x::Int) = x
foo(s::String) = length(s)
```

The choice of which method to execute when a function is applied is called *dispatch*. In multiple dispatch, the dispatch process chooses which of a function's method to call based on the number of arguments given, and on the type of all of the arguments, during run-time. This is where multiple dispatch differs from traditional single dispatch in object-oriented languages, such as Java, where dispatch occurs based only on the first argument, which is often treated specially. This first argument is often indicated syntactically, for example in Java this is denoted by typing the `"dot"` operator after the special argument.[10]

Too illustrate, in Java one might write something like:

```
rectangle.area(height, width)
```

Whereas in Julia the dispatched method is the one which matches all of the arguments and argument types, which might look something like:

```
area(rectangle, height, width)
```

From an object-oriented view, one can say that methods are defined on combinations of data types (classes), instead of encapsulating methods inside classes.[11] Figure 1 shows how methods belong to objects, whereas Figure 2 shows how methods are applicable on multiple objects.

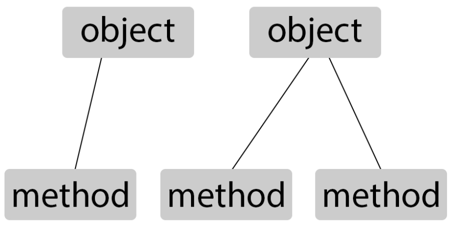

Figure 1: Illustrates how methods are defined in objects, in languages such as Java.

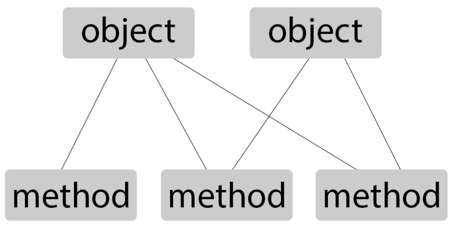

Figure 2: Illustrates how methods are defined on objects.

One might see the usefulness of multiple dispatch in mathematical code, where it pretty much makes no sense to say that an operation belongs to one argument more than any of the others, since the implementation of a mathematical operator generally depends on all of the types of all of its arguments.
For example, the addition(+) operation in `x + y`, it does not make much sense for the operation to belong to `x` any more than it does to `y`. As of February 24, 2017, the mathematical operation(function) + in Julia has 163 methods, which all are applicable on different argument types.

Multiple dispatch is also a powerful paradigm when it comes to structuring and organizing programs.

Although it seems a simple concept, multiple dispatch is perhaps the single most powerful and central feature of the Julia language! Multiple dispatch along with a parametric type system, gives one the ability to abstractly express high-level algorithms which are decoupled from implementation details in Julia.

2.5 Embedding other languages

Julia has the ability to run code from other languages such as C and Fortran. It does this in a way that is both simple to program and efficient. C code can be executed by Julia if it has been compiled as a part of a shared library. The code can then be executed using the function `ccall` with the library and function as parameters. When calling C code from Julia, Julia's JIT will run the same machine instructions as a native C call would. This results in that Julia will be as fast as native C when executing C code contained in a library.[12]

Code from Interpreted languages such as Python can also be embedded using the **PyCall** package.

The capability to run code from other languages is very useful for a new language such as Julia, as it allows a programmer to access a mature, high-quality library even though there may not exist such a library built in Julia yet. It is also possible to embed Julia code in other languages if the programmer wants to use Julia but does not want to rewrite the rest of the project. In C this is accomplished with the Julia API.

2.6 Parallelism and Concurrency

Parallel computing is an important part of a programming language. This is getting increasingly important as we are getting CPU:s with more and more threads, but the gains we see in single threaded performance year over year is diminishing. Programming a program that can run on multiple threads is usually difficult and has many pitfalls in most programming languages. The programming language has to support parallelism in a way that is both simple to program and mitigates potential problems, and also efficiently scales with the number of threads available. Since Julia is primarily focused towards technical/numerical computing, in addition to running a program on multiple threads it also has the goal to run a computation across multiple computers for solving complex problems faster.

Julia supports the use of both processes and threads, but it was only recently that support for threads were added, and they are still in the experimental phase. Therefore, processes are the primary means for parallel programming in Julia. Compared to a Thread, a process does not have a common address space. This means that two processes can not communicate via shared memory. Instead, communication in Julia is based on message passing. By not using shared memory, Julia avoids the common concurrency problem of synchronized use of shared memory, but at the cost of more resource usage and overhead.[13]

Julia's Concurrency model is based on the actor model. The actor model is a conceptual model on how to deal with concurrency computations. In the actor model, actors are the primitive units of computation. Actors can perform calculations, send messages to any other actors and they can also create more actors. Actors are implemented as processes in Julia. In Julia, processes typically communicate on a higher level than using messages. Programmers instead uses functions for communication, where one process invokes a function on another process. The idea is very similar to what we have in object-oriented languages: An object receives a message (a method call) and do something depending on which message it receives (which method we are calling).[14] Calling a function in another process is done with the function `remotecall`. `remotecall` returns immediately with a `Future` object when it is called. This object can then be used to retrieve the result of `remotecall` when the result is needed.

It is very easy to have a program run on multiple computers in Julia. This is done by supplying a list of the addresses of the computers to the function `addprocs`. The programmer can then treat the computers as normal processes.[13]

2.7 Modules

In Julia, identifiers such as types, functions, macros and variables are called names. Inside a `module`, using the keyword `export`, one may define which of the names within that module are to be visible outwards (i.e. which names can be imported).
Modules are separate variable workspaces i.e. they introduce a new global scope. Modules allow you to create top-level definitions without worrying about name conflicts.[15]

One can use the keyword `importall` as in the following example, instead of
`import Numbers: two, three` or `import Numbers.two, Numbers.three`.

```
module Numbers
    export three, two

    three = 3

    function two()
        2 * one()
    end

    one() = 1
end

importall Numbers

println(two() + three)
```

Notice that, in this example the name `one` is only visible inside the module `Numbers`.

It is also possible to copy the contents of a file entirely with `include("my_file.jl")`, but keep in mind that, as to expect, any modules within that file still need to be imported if to be used.

Imported names are read-only but are extendible for function names. One can also use the keyword **using** in place of **import** to not allow function extension.

Namespace issues should be mentioned, although this is only a problem that arises on selective importing. One can always refer to a name within a module hierarchically as **Numbers.two**. But defining **two** outside the module will give a warning unless all names are imported as `[module name].[name]` with **import Numbers**.

Julia has some pre-defined modules that are always loaded. `Main` is the top-level module, and julia starts with `Main` set as the current module. Variables defined at the prompt go in `Main`.[15] `Main` uses the built in module `Base` which defines the standard library i.e. the contents of the folder `base/`. And in turn, `Base` uses the `Core` module which presents the most basic functionality such as `eval()` and `typeof()`.

3 Use cases

Julia is first and foremost a language designed for technical/numerical computing. So the primary use case for Julia is to solve technical and numerical problems.

Julia can be used in computations where arbitrarily precision is needed, because Julia has support for arbitrary large numbers using the **BigInt** type, and arbitrarily good floating-point precision using the **BigFloat** type.

Julia can be used in situations where MATLAB, FORTRAN and R are used.

Julia is useful when the programmer wants to generate fast code, as the programmer has the ability to write high-level code and then inspect the resulting Low-Level Virtual Machine (LLVM) optimized code, and Native Assembly code counterparts directly in the Julia environment, and optimize the code to make the it run faster.

Although Julia is primarily for technical/numerical computing there is nothing that restricts you from using it as a general purpose language, but since it is missing a lot of object-oriented concepts it can be difficult to program large complex programs in Julia. As of now there is no option to create a runnable executable that works on all operating systems.

4 Benchmarks

Since Julia is a a high performance language, it is nice to see were Julia stacks up against it main competitors in numerical computing. Finding a reliable benchmark which is objectively pure is rather difficult, since Julia is a relative new language. To write our own benchmarks which are consistent and actually proves something would require a deeper and more advanced knowledge of multiple languages, which we currently do not have.

Sadly the only benchmark we found are those which Julia themselves has provided on their web page, see Figure 3. To say how accurate they are is difficult, since the benchmark is more than likely biased towards Julia.

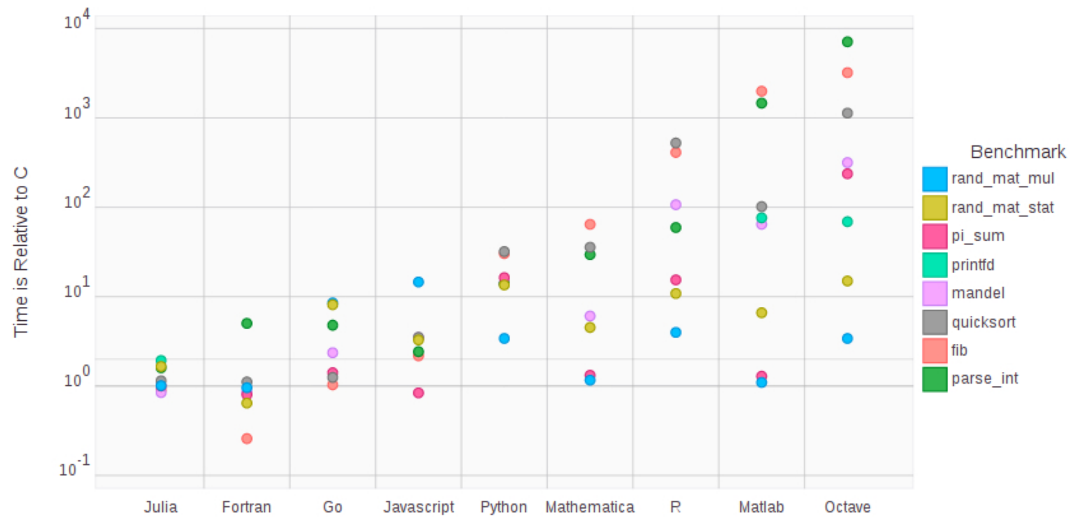

Figure 3: Julia's supplied benchmark times relative to C (smaller is better, C performance = 1.0)

From Figure 3 we can see that Julia seems to perform extremely well towards its competitors, but then again we do not know the accuracy of the numbers.

5 Discussion

Although Julia is a dynamic, high-level language that is pleasant to write code in, it is one of the highest performing languages. Even though Julia is focused on technical computing, there is nothing that stops you from using it as a general purpose language.
This unusual combination of traits combined with the fact Julia has not even got a 1.0 release yet makes Julia's future very promising.

`Tiobe` is an index which compares the popularity of programming languages. According to the `Tiobe` index, Julia placed 52 compared to last year when it was said to be the 73 most popular programming language.[16]

What might convey a MATLAB user from moving to Julia is its lackluster support for generating advanced plots. Although with time the support for advanced plots in Julia will more then likely be much improved, and might rival that of MATLAB's. One cannot forget however that Julia is free, while MATLAB is not, which speaks for why new users to technical/numerical computing might might want to start using Julia.

References

[1] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman *Why We Created Julia*, 2012. [Online]. Available http://julialang.org/blog/2012/02/why-we-created-julia (visited on Jan. 31, 2017)

[2] [Online]. Available http://julialang.org/downloads/ (visited on Jan 31, 2017)

[3] [Online]. Available http://julialang.org/ (visited on Feb 3, 2017)

[4] [Online]. Available http://docs.julialang.org/en/release-0.5/manual/variables/ (visited on Feb 3, 2017)

[5] [Online]. Available http://docs.julialang.org/en/release-0.5/manual/functions/ (visited on Feb 3, 2017)

[6] [Online]. Available https://github.com/JuliaLang/julia/issues/1090 (visited on Feb 10, 2017)

[7] [Online]. Available http://docs.julialang.org/en/stable/manual/types/ (visited on Feb 10, 2017)

[8] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman *Julia: A fresh approach to numerical computing*, 2015. [Online]. Available https://arxiv.org/pdf/1411.1607v4.pdf (visited on Feb. 10, 2017)

[9] [Online]. Available https://en.wikibooks.org/wiki/Introducing_Julia/Types (visited on Feb 10, 2017)

[10] [Online]. Available http://docs.julialang.org/en/stable/manual/methods/ (visited on Feb 10, 2017)

[11] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, Jiahao Chen *Array Operators Using Multiple Dispatch* [Online]. Available https://arxiv.org/pdf/1407.3845v1.pdf (visited on Feb 10, 2017)

[12] [Online]. Available http://docs.julialang.org/en/release-0.5/manual/calling-c-and-fortran-code/ (visited on Feb 8, 2017)

[13] [Online]. Available http://docs.julialang.org/en/stable/manual/parallel-computing/ (visited on Feb 16, 2017)

[14] [Online]. Available http://www.brianstorti.com/the-actor-model/ (visited on Feb 16, 2017)

[15] [Online]. Available http://docs.julialang.org/en/stable/manual/modules/ (visited on Feb 22, 2017)

[16] [Online]. Available http://www.tiobe.com/tiobe-index/ (visited on Jan 30, 2017)