Compilation and Language Design in Julia

JIT/AOT LLVM Metaprogramming and More

CIS561 Graduate Report

Alexandre Chen

1 Introduction

Julia is a newly developed programming language that specializes in Computational Science and Numerical Algorithms. It was designed to solve the famous two-language problem: Developers or scientists would use high-level easy-to-use languages such as MATLAB/Python to test the prototype of an algorithm and then re-implement it via languages that have better performance such as C/C++/FORTRAN for actual deployments. The two-language problem significantly lengthens the development cycle. Also due to the inconsistency usage of prototyping language and implementing language, other issues would caused which further increases the maintenance costs.

Ideally, researchers who develop numerical algorithms to solve problems arising from science and engineering would prefer a language that is easy to use as MATLAB, open as Python, and fast as C/C++/FORTRAN. And Julia is designed to satisfy these requirements. Julia adopts Just-in-time (JIT) compilation, although not exactly the same with JIT compiler in Java, to make it faster than interpreted languages such as Python. Julia is also compatible with LLVM for code generation and optimization. Aside from being fast and easy to use, Julia also supports functional programming paradigm,the heritage of LISP which turns out to be very handy in computational science where mathematical function evaluations are heavily used.

2 Multiple Dispatch

As a dynamic programming language, Julia uses multiple dispatch, allowing extensions of functionality by overloading. Each function can consist of a large number of methods. (E.g. "+" method in Julia has 180 methods) The types of variables that each method can handle are declared. Julia will dispatch to the most specific method by for a given function call. Julia doesn't use class and class functions as other programming language such as Java or C++ where dynamic dispatch is adopted. The Struct in Julia is similar to the struct used in C.

```
Struct Dual{T} // T is type of the variable
    s1::T
    s2::T
end

function Base.:(+)(a::Dual{T},b::Dual{T}) where T
    Dual{T}(a.s1 + b.s1, a.s2 + b.s2)
end

function Base.:(*)(a::Dual{T},b::Dual{T}) where T
    Dual{T}(a.s1 * b.s1, a.s2 * b.s2)
end
```

Suppose we then have function f(a,b) = a*b + a. For given variable a = 1, b = 2, when we call f(a,b). It will load operators "+" and "*" for Int type, with will return output Int(3). However,

we can also have input variables given in the form of our self-defined Dual Struct. c = Dual(1.0, 1.0), d = Dual(2.0,2.0). If we call function f(a,b), by overloading arithmetic operators (+) and (*) that we defined for Dual Struct, it will return output Dual(3.0, 3.0).

Dispatching on a function for a call with argument type T in our example consists in picking a method m from all the available methods of function f. The selection filters out methods whose type are not a super type of T, and it takes the method whose Type T' is the most specific of the left ones. Every position in the tuples T and T' have the same role, there is no single receiver position that takes precedence over another. This is different from single dispatch that is adopted in C++ and Java.

3 JIT in Julia

The internal levels of code generation follows the processes below. From high-level structure going all the way down.

- Method Definition → methods, metaprogramming
- code_lowered → generated functions, simplified code structure
- code_typed
  - code_typed → precompiled modules (.jl) global inference
  - local optimization
  - code_warntype → dynamic behavior annotations
- code_llvm → external codegen, llvmcall-2.0 julep
  - Intermediate representation for low-level optimization
- code_native → static system image (.so/.dll/.dylib) Machine Code Representation

Just-in-time compilation (or dynamic translation or run-time compilation) is a way of executing computer code that involves compilation during the execution of a program at run-time rather than prior to execution, which is in contrast of ahead-of-time (AOT) compilation. [1] It is a combination of AOT and interpretation. Roughly the JIT compilation combines the speed of compiled code while being flexible like interpretation. In theory, since JIT is a dynamic compilation and allows adaptive optimizations including dynamic recompilation and micro-architecture-specific speedups, it can be faster than static compilation. The earliest version of JIT can be attributed to LISP by John McCarthy in 1960. JIT compilers are mostly known for Java and Lua.

Julia's LLVM-based just-in-time (JIT) compiler, combined with the its own language design allows it to have similar level of performance of C. However, the JIT used in Julia is not exactly the same as the JIT in Lua or Java. For example, LuaJIT is a Just-In-Time compiler for Lua programming language, but it offers APIs so i can be used as a general-purpose standalone language for different platforms. There is existing approach by applying LuaJIT to interpreted languages such as Python for better performance.

However, Julia used a different approach towards JIT implementation. Other JIT setups for scripting languages use something known as tracing JIT. It adopts probabilistic method to determine the frequencies of the parts of the code that are repetitively run and compiled these code. This tracing JIT approach is different from traditional JIT that works on a per-method basis. Julia, by comparison, is almost static. When running any Julia function, it will first take function calls and

then auto-specialize it down to concrete data types. So f(x::Number) = x ^ 3 will be specialized to Float64 when it is called as f(1.0). This type information propagates along type-inference as far as it can go. If you do a x^3 internally, it will replace that expression with ^(::Float64,3) (Here 3 is the constant that we used), and then since it can determine all the variable types it will use, it will push this all the way down to the generate LLVM IR code. And from this level, machine code can be generated. Notice that we only generated the code for function f(x::Number) with Floating point input variables. If we are going to recall the function f(x::Number) with input value 2.0, We can use the same compiled code. But If we want to use input value 1 which is an Integer, Julia will compile a separate version for Int.

At first sight, this might seem highly inefficient because essentially you are going to compile different versions of the same function for different input variable types separately. But the advantage of this process is that it's clear and deterministic. Also since Julia is widely used in Computational Science, in many cases, functions are handling numerical data as input. Usually based on different purposes, from accuracy requirements to stability requirements the precision levels for certain types of variables are fixed before the program is written and compiled. Specifying the types of compiled functions doesn't suffer from efficiency too much as it might seem.

Adopting deterministic rules would also help developers know what's going behind the code generation and how to optimize for efficiency, this transparency is lacked in MATLAB, which is a traditional commercial software widely used in numerical algorithms and engineering.

However, there is certain drawback in this approach. In order this process can work well, types of variables inside a function should be "type-stable". Consider the following example:

```
function g(v,n)
    x = 0
    for i in 1:n
        x += v
    end
    return x
end
```

If we call this function with input variable v=1.0, n=5. It is weird that this function is not as fast as we would expect. The reason is because the type of x starts as Int. But in step x += v, Int + Float64 produces a Float64 type so the type of x gets changed here. In static programming language like C, you can't change the type of variables once declared. Julia is a dynamic typed language that can compile x in a way that is dynamically typed. The accompanying cost of the increase in agility is the decrease in efficiency, and this is why this code is slower than expected.

Even though Julia is a dynamic language, programmers are often advised to avoid data type conversions for the sake of efficiency. One way to achieve this is that Julia supports method overloading. We can define g(v,n) function for different input variables. In different versions of function g(v,n), we use instance variables matching the types of input variables. So Julia would know exactly which g(v,n) we would need to compile when given input v=1.0, n=5. However, doing this will certainly make duplication repeatedly and is not the most efficiently way. this is a task This can be avoided by using the same type with input variable v for instance variable x. Julia provides built-in functions such as zero(), ones(), similar() that take input variables and return variables of the same type used as instance variables very often in initialization processes. For example, we can replace our previous function with a generic type-stable one

```
function g(v,n)
```

```
    x = zero(v)
    for i in 1:n
        x += v
    end
    return x
end
```

Compared with tracing JIT compilation, this semi-static approach for JIT implementation doesn't not require statistical data from run-time information, and is easier to introspect and diagnose the code generation process, especially when things are going wrong. This feature is extremely important for many computational scientists who don't have compiler knowledge but need to develop a package consists of tens of separate code files and make it compatible with other implemented packages. There are other packages in Python adopts similar approach with Julia, such as Numba. The difference is that this feature is a language-level feature of Julia, users have better direct handling of data types than Python + Numba, which allows Julia to scale well.

4 LLVM in Julia

LLVM is a compiler framework created by Chris Lattner and Vikram Adve at University of Illinois at Urbana-Champaign for the support of *transparent, lifelong program analysis and transformation* for arbitrary programs. [2] LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form. It has several novel features which includes a simple, language-independent type system, an instruction for typed address arithmetic, and a simple exception handling mechanism for high-level languages. LLVM currently supports many languages. The Clang is a part of LLVM framework that uses LLVM as its back end. It has become a popular alternative to GCC for better error message handling and is much faster and uses far less memory. Julia dynamically links to LLVM by default. The LLVM C Backend also allows translating Julia Code to C code so the algorithm developed in Julia can also be easily redeployed on servers that don't support C. Also there is high-level LLVM package in Julia that allows users to directly access LLVM IR code for profiling and diagnosis.

5 Metaprogramming in Julia

Julia provides various features for defining functions at compile-time and run-time. Most significantly, it provides Macros to generate code and reduce the need to call built-in evaluation function eval(). Macro maps a tuple of arguments to an expression that has been already compiled. By using macros, many duplicate or similar functions can be avoided. Many built-in Julia functions for performance analysis, code profiling have been implemented in macros, which simplifies the process for code bench marking and optimization. Metaprogramming also enables efficient use of already implemented functions without writing APIs to work with other functions. This has improved the potential collaborations among Julia users significantly, especially those from other engineering and science fields who lack background knowledge of programming languages and software methods.

6 Conclusions

Julia is a dynamic programming language specifically designed for numerical algorithms that has some similar properties with static ones to avoid run-time uncertainty for efficiency. It was built on

top of modern compiler framework and absorbs many language designs from other programming language such as JIT compilation, Metaprogramming etc for the purpose of being fast and easy-to-use in computational science. It is an evolving languages with many new features and has a booming Julia ecosystem. Many new features including garbage collection are gradually added to Julia and getting improved. The metaprogramming property allows the low-level Julia source code to be written in native Julia, which is easy for Julia users to read and understand low-level functions that are used in their code compilation. It's also to easy to make different Julia packages compatible with others. Actually many Julia native functions are coming from merging user-generated packages. Although with certain limitations coming from the language design, Julia is becoming a popular and promising programming language in computational science.

References

[1] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky. Julia: Dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA):120:1–120:23, Oct. 2018.

[2] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.