Dead Optimizers Society

Julia para Investigación de Operaciones

Una gentil introducción

Eduardo Salazar Treviño

Facultad de Ingeniería Mecánica y Eléctrica, UANL

Noviembre 5, 2021

Contenidos

Orígenes de Julia

julia Download Documentation Blog Community Learn Research JSoC Sponsor

Why We Created Julia

14 February 2012 | **Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman**

Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman
In short, because we are greedy.

We are power Matlab users. Some of us are Lisp hackers. Some are Pythonistas, others Rubyists, still others Perl hackers. There are those of us who used Mathematica before we could grow facial hair. There are those who still can't grow facial hair. We've generated more R plots than any sane person should. C is our desert island programming language.

We love all of these languages; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

We are greedy: we want more.

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

En pocas palabras

Why we created Julia(2012)
A programming language to heal the planet together: Julia(2020)
Julia: A Fresh Approach to Numerical Computing(2017)

Desde 2009, distinto personal del MIT trabaja en Julia para crear un lenguaje que se vea como Python, se sienta como **Lisp** y sea tan rápido como C.
Su versión 1.0 salió en 2018 y recientemente en 2021 se publicó la versión 1.6.0, LTS. Tiene influencias de distintos lenguajes tales como Ruby, Fortran, Matlab y Scheme; desarrollado por un equipo de matemáticos y computólogos para resolver el problema de los dos lenguajes en la computación científica.

Razón de ser

Conforme más avanza el tiempo, es más común encontrar que estudios e investigaciones de múltiples campos involucran trabajo de cómputo de alto rendimiento, con investigadores que no son programadores. Tenemos economistas, biólogos, sociólogos cuyo primer acercamiento con la programación es software crítico para su labor.

El problema de los dos lenguajes se resume así:

1. Lenguajes accesibles para cualquier usuario (R, Python, Matlab) adecuados para el prototipado de modelos, pero tienen limitaciones inherentes a su diseño
2. Lenguajes de alto rendimiento (C, C++, Fortran) útiles para el desempeño y las partes críticas, pero difíciles de leer y escribir

Es ineficiente tener dos lenguajes distintos dentro de un mismo proyecto en el que constantemente se tienen que reescribir partes de un lenguaje a otro, introduciendo error humano y pérdida de tiempo.

Casos de uso

- Project Celeste (2017): Catalogar 188 millones objetos astronómicos en 15 minutos utilizando una supercomputadora, con 1.54 petaflops de rendimiento.
- Modelar y pronosticar demanda eléctrica en Francia (2021): Électricité de France modela la demanda de electricidad de Francia para pronosticar demandas futuras y tomar mejores decisiones.
- CliMA (2020): Crear un modelo global climático para entender mejor los impactos del cambio climático.
- Optimización de Infraestructura Crítica (2017): Los Alamos National Laboratory forma parte del Departamento de Energía de los Estados Unidos y son responsables de predecir y mitigar el impacto de desastres naturales en redes de infraestructura tales como gas natural, energía y agua, así como resolver la GO Competition.

Mecanismos internos de Julia

Julia es un lenguaje con compilación (JIT), es decir, se siente como un lenguaje interpretado pero con los beneficios de la compilación.
Es open source, dinámico, multiparadigma, diseñado para ser interoperable con C y Fortran (la interoperabilidad con Python y R también es posible mediante bibliotecas), con una stdlib especializada en matemáticas y concurrencia.
Tiene el modo de REPL, compilador de programas enteros, el mejor manejador de paquetes que personalmente he manejado y una documentación excelente.

Sintaxis Básica

```
using Dates

function saludar(f::Function)::Nothing
  nombre = f()
  println("Buenos días $nombre, hoy es $(today())")
end

function aleatorio()
  nombres = ["Beatriz", "Citlali", "Diana", "Neto", "Rodolfo", "Roger", "Uriel"]
  nombre = nombres[rand(1:end)]
  return nombre
end

for x ∈ 1:3
  saludar(aleatorio)
end
```

Multiple Dispatch

El paradigma central de programación de Julia es el Multiple Dispatch. En otros lenguajes (C++) podemos sobrecargar operadores existentes para que funcionen con nuestros tipos definidos, así como sobrecargar funciones para intentar reutilizar código. Sin embargo, Multiple Dispatch es una mejor versión de ambas sobrecargas y es lo que le permite a Julia ser rápido pero dinámico.
The Unreasonable Effectiveness of Multiple Dispatch(2019)

Comparando Julia y C++

```
abstract type Animal end
struct Lagarto <: Animal nom :: String end
struct Conejo <: Animal nom :: String end

carrera(l::Lagarto, c::Conejo) = "$(l.nom) nada más rápido"
carrera(c::Conejo, l::Lagarto) = "$(c.nom) corre más rápido"
carrera(a::T, b::T) where T <: Animal = "$(a.nom) y $(b.nom) se duermen"

function encontrar(a::Animal, b::Animal)
  println("$(a.nom) se encuentra a $(b.nom) y hacen una carrera")
  println("Resultado: $(carrera(a,b))")
end

godz = Lagarto("Godzilla")
tamb = Conejo("Tambor")
encontrar(godz, tamb)
encontrar(tamb, godz)
encontrar(godz, godz)
```

Julia

```
> julia .\md.jl
Godzilla se encuentra a Tambor y hacen una carrera
Resultado: Godzilla nada más rápido
Tambor se encuentra a Godzilla y hacen una carrera
Resultado: Tambor corre más rápido
Godzilla se encuentra a Godzilla y hacen una carrera
Resultado: Godzilla y Godzilla se duermen
```

Código de C++

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
    public:
        string nom;
};
string carrera(Animal a, Animal b) { return a.nom + " y " + b.nom + " se duermen"; }
void encontrar(Animal a, Animal b) {
   cout << a.nom << " se encuentra a " << b.nom << endl;
   cout << "Resultado: " << carrera(a, b) << endl;
}
class Lagarto : public Animal {};
class Conejo : public Animal {};

string carrera(Lagarto l, Conejo c) { return l.nom + " nada mas rapido"; }
string carrera(Conejo c, Lagarto l) { return c.nom + " corre mas rapido"; }

int main() {
   Lagarto godz; godz.nom = "Godzilla";
   Conejo tamb; tamb.nom = "Tambor";
   encontrar(godz, tamb);
   encontrar(tamb, godz);
   encontrar(godz, godz);
}
```

Ejecución de C++

```
$ g++ -o md md.cpp && ./md.exe
Godzilla se encuentra a Tambor
Resultado: Godzilla y Tambor se duermen
Tambor se encuentra a Godzilla
Resultado: Tambor y Godzilla se duermen
Godzilla se encuentra a Godzilla
Resultado: Godzilla y Godzilla se duermen
```

Metaprogramación

El legado más fuerte de Lisp en Julia es la metaprogramación. En pocas palabras, metaprogramar hace referencia al hecho de poder manipular el programa actual usando el mismo lenguaje.
Metaprogramar permite editar y ejecutar código ya procesado por el lenguaje durante la ejecución para añadir dinamismo y capacidades adicionales que un lenguaje tradicional no cuenta con.
In what sense are languages like Elixir and Julia homoiconic?(2019)

Metaprogramando un do *n* veces

```
macro dotimes(n, cuerpo)
  quote
      for i = 1:$(esc(n))
          $(esc(cuerpo))
      end
  end
end

@dotimes 3 println("Hola mundo")

@dotimes 2 begin
  for i ∈ 1:10
    j = 1:i |> sum |> println
  end
 @info "Terminado"
end
```

Resultado del do *n*

```
> julia .\meta.jl
Hola mundo
Hola mundo
Hola mundo
1
3
6
10
15
21
28
36
45
55
[ Info: Terminado
1
3
6
10
15
21
28
36
45
55
[ Info: Terminado
```

Ejemplo: Heurística Inserción más cercana

Jupyter Notebook

JuMP: Investigación de Operaciones

JuMP (Julia Mathematical Programming) es un lenguaje de modelación algebraico (como GAMS o AMPL) embebido dentro del lenguaje Julia como una biblioteca externa que le ofrece al usuario todas las ventajas de usar un lenguaje de programación tales como extensibilidad e implementación de modelos más diversos para después hacer interfaz con un solver.
Su análogo más similar en otros lenguajes es Pyomo, pero al estar escrito en Python tiene un desempeño lento.
Hoy en día podemos usar JuMP para problemas de LP, MIP, MILP, programas cuadráticos, cónicos, no lineales con restricciones, etcétera. Problemas de optimización convexa se pueden resolver con Convex.jl, que forma parte del mismo ecosistema.

Características de JuMP

- Velocidad: JuMP es tan rápido como soluciones comerciales y es por mucho el mejor AML open source en esa métrica.
- Extensibilidad: Diseñado para funcionar como un DSL dentro de Julia, JuMP puede usarse para más aplicaciones que IO y colabora con otros paquetes
- Agnóstico de solver: A través de otro paquete (MathOptInterface), JuMP puede trabajar con varios solvers sin tener que modificar el modelo.
- Ecosistema: JuMP está considerado como uno de los **mejores** paquetes de Julia y gran cantidad de otros paquetes se desarrollan alrededor.
  JuMP: A Modeling Language for Mathematical Optimization(2017)

Solvers compatibles con MOI

- Comerciales: CPLEX, Gurobi, Knitro, BARON, Xpress, Mosek
- Open Source: CBC, CLP, GLPK, Ipopt, NLopt, SCIP
- Solvers en Julia: Alpine, COSMO, EAGO

Adicionalmente, existen paquetes para hacer interfaz con solvers que sólo usan GAMS o AMPL como lenguajes de modelado.

Knapsack

```
using JuMP,Gurobi,Test
function knapsack(; verbose = true)
    # maximizar Σi=1:n v_i x_i
    # s.a. Σi=1:n w_i x_i ≤ W, x_i ∈ {0,1}
    v = [5, 3, 2, 7, 4] # valores
    w = [2, 8, 4, 2, 5] # pesos
    W = 10 # capacidad
    model = Model(Gurobi.Optimizer)
    @variable(model, x[1:5], Bin)
    @objective(model, Max, v' * x) # la transpuesta de v por la funcion objetivo
    @constraint(model, w' * x <= W)
    optimize!(model)
    if verbose
        println("El valor de la función objetivo es : ", objective_value(model))
        println("La solución es:")
        for i in 1:5
            print("x[$i] = ", value(x[i]))
            println(", v[$i]/w[$i] = ", v[i] / w[i])
        end
    end
    @test termination_status(model) == MOI.OPTIMAL
    @test primal_status(model) == MOI.FEASIBLE_POINT
    @test objective_value(model) == 16.0
    return
end
knapsack()
```

Ejecución de Knapsack

```
❯ julia .\knapsack.jl
  Activating environment at `C:\Users\Salazar\Documents\AG-DIC21\opt\juliaIO\Project.toml`
Academic license - for non-commercial use only - expires 2021-11-18
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (win64)
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 1 rows, 5 columns and 5 nonzeros
Model fingerprint: 0x51d1e86e
Variable types: 0 continuous, 5 integer (5 binary)
Coefficient statistics:
  Matrix range     [2e+00, 8e+00]
  Objective range  [2e+00, 7e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+01, 1e+01]
Found heuristic solution: objective 8.0000000
Presolve removed 1 rows and 5 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds
Thread count was 1 (of 8 available processors)

Solution count 2: 16 8

Optimal solution found (tolerance 1.00e-04)
Best objective 1.600000000000e+01, best bound 1.600000000000e+01, gap 0.0000%

User-callback calls 37, time in user-callback 0.00 sec
El valor de la función objetivo es : 16.0
La solución es:
x[1] = 1.0, v[1]/w[1] = 2.5
x[2] = 0.0, v[2]/w[2] = 0.375
x[3] = 0.0, v[3]/w[3] = 0.5
x[4] = 1.0, v[4]/w[4] = 3.5
x[5] = 1.0, v[5]/w[5] = 0.8
```

Más ejemplos

JuMP
Julia Programming for Operations Research 2/e (2021)

Paquetes Interesantes

- DifferentialEquations.jl: De acuerdo a sus benchmarks, es el mejor solver de ecuaciones diferenciales existente
- Zygote.jl: El sistema de autodiferenciación más moderno y rápido de todos
- DataFrames.jl: El equivalente de Pandas y dplyr
- InfraestructureModels.jl: Optimizador de redes de infraestructura: Gas Natural, Petróleo, Transmisión y Distribución de Electricidad y Agua
- ModelingToolkit.jl: Sistema de modelado para física y procesos con machine learning científico
- Turing, Plots, PyCall, RCall, Makie, Javis, Flux

Desventajas de Julia

- Time to First Plot: El tiempo en descargar e instalar un paquete por primera vez es alto, pero incluso si ya se tiene instalado el tiempo para cargar el runtime sigue siendo largo.
- Paradigma distinto: Julia no soporta OOP y usa un paradigma que es teóricamente lo contrario
- Ecosistema temprano: Apenas se cumplirán 3 años de su primera versión estable por lo que el ecosistema es infinitamente más pequeño que comparado a ecosistemas de otros lenguajes.
- Sistema de Tipos unortodoxo: A veces el sistema de tipos puede meterse en el camino del programador y resultar confuso

Fin

Por su atención, gracias ☺

https://github.com/eduardosalaz/JuliaParaIO
https://julialang.org/
https://docs.julialang.org/en/v1/
https://discourse.julialang.org/