Julia
Uma análise sobre a linguagem

Autores: Gustavo Cayres

Leonardo Benicio

Pedro Marcondes

Instituto de Matemática e Estatística, 4 de julho de 2017.

Sumário

1 Resumo

Em todas as áreas da ciência, desde cálculos matemáticos em simulações físicas à análise de dados estatísticos em experimentos sociais, a computação tem um papel cada vez mais decisivo. Esse crescimento para outras áreas do conhecimento torna necessária a criação de ferramentas capazes de tratar um grande leque de problemas. Nessa pesquisa, pretendemos abordar a linguagem de programação Julia, que está se mostrando promissora como uma dessas novas ferramentas. Pretendemos tratar do surgimento da linguagem e dos paradigmas que seus criadores tentaram seguir durante seu desenvolvimento.

Trataremos, ainda, de suas principais características, incluindo sua relevância na computação paralela e distribuída e as funcionalidades que a melhor diferenciam de outras linguagens de programação. Por fim, destacaremos alguns artigos e projetos relevantes realizados em Julia, além de alguns problemas que observamos durante nossa experiência com a linguagem.

2 História

Julia teve seu desenvolvimento iniciado em 2009 por 4 cientistas da computação:

- **Alan Stuart Edelman.** Professor de Matemática Aplicada em *Massachusetts Institute of Technology*[1].
- **Jeffrey Werner Bezanson.** Formado em Ciência da Computação por *Harvard* e tornou-se PhD no *MIT* sob orientação de Alan Edelman[2].
- **Stefan Karpinski.** Formado em Matemática por *Harvard*.
- **Viral B. Shah.** Pesquisador indiano doutor em Ciência da Computação por *University of California*, Envolvido no projeto indiano *Aadhaar*[3].

A divulgação da linguagem começou em 2012, momento a partir do qual a popularidade da linguagem se manteve em crescimento constante. Seus criadores tinham como objetivo obter uma linguagem de uso geral que desfrutasse de qualidades de outras linguagens mais específicas. Idealmente, Julia deveria ser:

- De uso geral, como Python.
- Rápida como C.
- Dinâmica como Ruby.
- Com macros, como Lisp.
- Com uma boa notação matemática e poderosa para álgebra linear, como Matlab.
- Fácil para cálculos estatísticos, como R.
- Natural para o processamento de strings, como Perl.

Em 2015, os criadores da linguagem fundaram a *Julia Computing Inc.*, uma empresa responsável por desenvolver produtos de suporte para Julia e seus usuários.

3 Características

Nesta seção, apontaremos algumas propriedades importantes da linguagem que tentam tingir os objetivos de seus criadores.

3.1 Performance

Julia apresenta uma performance surpreendente para sua versatilidade, permitindo que ela compita tanto com linguagens científicas já estabelecidas como Matlab e R, quanto com linguagens de mais baixo nível como C e Fortran (figura 1).

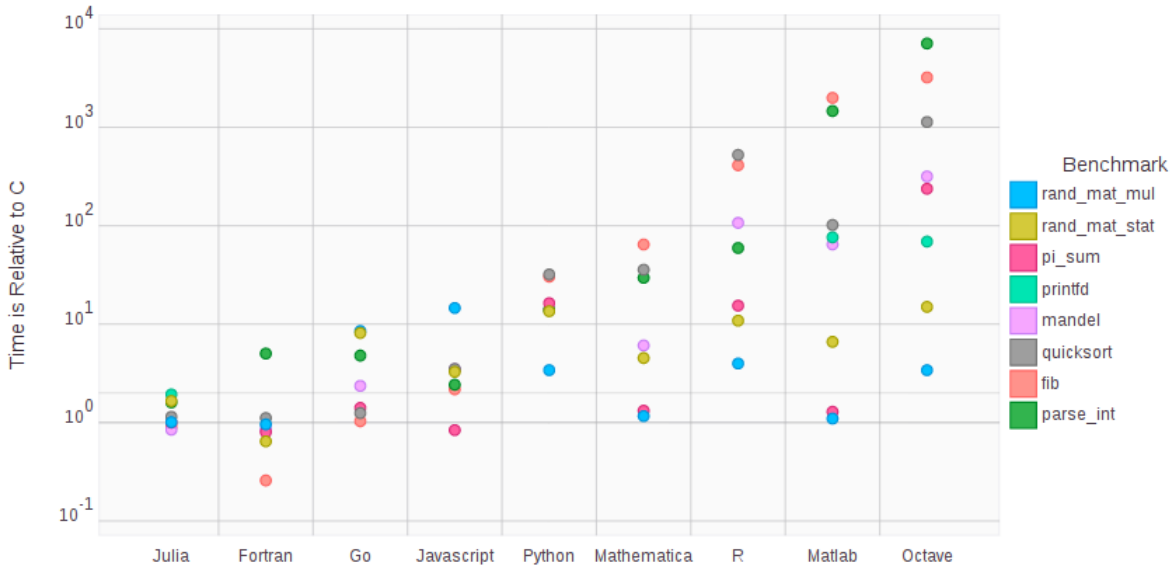

Figura 1: Comparação do tempo de execução de alguns algoritmos notáveis entre Julia e outras linguagens, em relação ao tempo em C (tempo em C = 10^0) [4].

3.2 Tipos

Julia permite o uso de tipagem dinâmica, facilitando o aproveitando das flexibilidade das funções da linguagem (ver seção 3.3); por outro lado, é possível utilizar variáveis com tipos a fim de garantir a robustez do código e melhorar a performance dos programas (figura 2). Notavelmente, é possível melhorar consideravelmente o tempo de execução seguindo o conceito de *type stability*: quando todas as operações de um trecho que código envolve os mesmos tipos de variáveis, o compilador é capaz de gerar código especializado para o tipo em questão, levando a uma grande melhora de eficiência. O exemplo da figura 3 ilustra essa situação; na primeira função, a variável s começa como um inteiro e é convertida em um *float* durante a execução da função, o que impede o compilador de assumir informações sobre o conteúdo da variável, levando à queda de eficiência mostrada na figura 4.

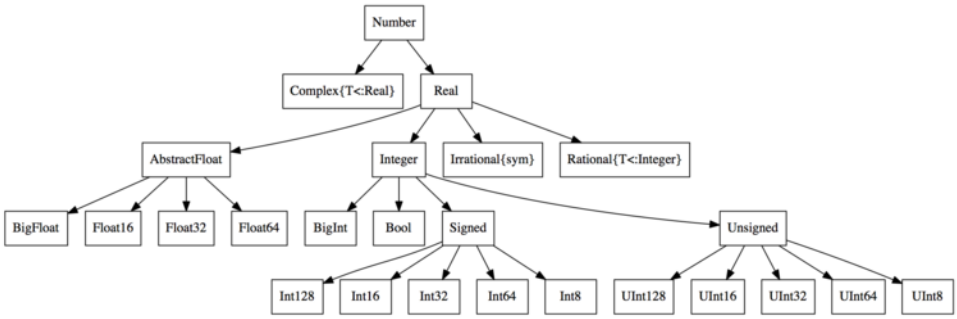

Figura 2: Árvore de tipos númericos. Julia organiza os tipos númericos de forma análoga à matemática formal. É importante notar, ainda, que variáveis só podem assumir os tipos nas folhas (tipos concretos), enquanto os tipos nos outros nós (abstratos) são utilizados para especificar argumentos de funções, garantindo que todos os subtipos daquele tipo serão aceitos no argumento [5].

```
julia> function t1(n)
         s  = 0
         for i in 1:n
            s += s/i
         end
       end
t1 (generic function with 1 method)

julia> function t2(n)
         s  = 0.0
         for i in 1:n
            s += s/i
         end
       end
t2 (generic function with 1 method)
```

Figura 3: Funções *t1* (sem estabilidade de tipos) e *t2* (com estabilidade de tipos).

```
julia> @time t1(10000000)
  0.679342 seconds (30.00 M
allocations: 457.756 MB, 4.99% gc
time)

julia>  @time t2(10000000)
  0.000002 seconds (4 allocations:
160 bytes)
```

Figura 4: Tempos de execução das funções. Note o desempenho muito pior devido à falta de estabilidade.

3.3 *Multiple Dispatch*

Frequentemente no tratamento de problemas matemáticos, uma mesma operação pode ter comportamentos diferentes conforme os operandos nos quais é aplicada. Julia lida com essa dificuldade de forma elegante através do sistema de *multiple dispatch*, permitindo que cada função assuma diversos comportamentos pré-configurados, dependendo dos argumentos recebidos.

Ao definirmos uma função pela primeira vez, estamos associando a ela seu primeiro método:

julia ⟩ f(x::Number, y::Number) = 2x + y
f (generic function with 1 method)

Podemos testar a função utilizando como argumentos qualquer subtipo de *Number*:

julia ⟩ f(2.0, 3.0)
7.0
julia ⟩ f(2, 3.0)
7.0
julia ⟩ f(2.0, 3)
7.0
julia ⟩ f(2, 3)
7

Entretanto, podemos querer comportamentos diferentes para a f em diferentes situações. É nesse caso que *multiple dispatch* se torna útil:

julia ⟩ f(x::Float64, y::Number) = 2x - y
f (generic function with 2 methods)

A segunda "definição" da f na verdade adiciona um novo método para a função (nesse exemplo, um que adiciona um comportamento específico quando operando sobre um real de 64bits e um número qualquer. Se listarmos os métodos da função agora, temos:

julia ⟩ methods(f)
2 methods for generic function "f":
f(x::Number, y::Number) in Main at none:1
f(x::Float64, y::Number) in Main at none:1

Refazendo as operações anteriores, obtemos:

julia ⟩ f(2.0, 3.0)

1.0

julia ⟩ f(2, 3.0)

7.0

julia ⟩ f(2.0, 3)

1.0

julia ⟩ f(2, 3)

7

Ou seja, nas primeira e terceira operações, o método 2 foi invocado, enquanto nas segunda e quarta operações, o método 1 foi invocado.

3.4 Paralelismo

A maioria dos computadores modernos possuem mais de uma CPU e vários computadores podem ser combinados para formar um "cluster"que, por sua vez, possui um poder de processamento muito maior que uma única CPU. Um dos grandes gargalos da programação paralela é comunição entre os processos, o que pode acarretar em um programa mais lento. Julia fornece um ambiente de multiprocessamento baseado em troca de mensagem que permite múltiplos processos rodarem em um domínio de memória separado.

O ambiente de troca de mensagem da Julia é diferente de outros mais conhecidos, como por exemplo o MPI. Em Julia a comunicação de processos é "one-sided", isso quer dizer que o programador só precisa lidar com um processo em uma operação que depende de dois processos.

A programação paralela nessa linguagem de programação foi construída com base em duas primitivas "Remote References"e "Remote Calls". Uma "Remote Reference"é um objeto que pode ser usado por qualquer processo que referencia um outro objeto guardado em um processo particular. Já uma "Remote Call"é uma requisição feita por um processo para que outro processo execute uma função com certos argumentos.

3.4.1 Remote References

Uma "Remote Reference"é feita de dois modos, através de um "Future"ou de "Remote Channel".

- Uma "Remote Call"retorna umediatamente um "Future"como resultado, enquanto o processo que fez a "Remote call"procede para sua proxima operação enquanto a função é executada em outro processo.

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, Nullable{Any}())

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, Nullable{Any}())

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

Figura 5: Exemplo de Remote Call e Future

- Um "Remote Channel"são canais de comunição que podem ser reescritos. Como por exemplo, vários processos podem se coordenar referindo-se ao mesmo "Remote Channel"

3.4.2 Movimentação de Dados

Como dito anteriormente, enviar dados é um dos grandes gargalos da programação paralela, a velocidade no qual o dado que você precisa está disponível no processo correto. Logo, reduzir o tanto de dados enviados entre os processos é crucial para poder escalar um projeto. Ao observamos o método 1 da imagem 2, vemos

Method 1:

```
julia> A = rand(1000,1000);

julia> Bref = @spawn A^2;

[...]

julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawn rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

Figura 6: Exemplo "remote call"

que foi gerado uma matriz A localmente e ela foi elevada ao quadrado em um segundo processo. Já no método 2, a matriz é gerada e elevada ao quadrado por um segundo processo. A diferença entre os dois métodos parece trivial porém é muito importante para a programação paralela e concorrente.

Esses dois métodos mostram de forma simples como organizar o fluxo de dados em uma aplicação paralela. Nesse exemplo, o método 2 envia muito menos dados entre os processos o que acarreta em uma velocidade maior para a aplicação.

3.4.3 Loops e Maps Paralelos

Alternativamente a seção anterior, existem muitos algoritmos que não requerem movimentação de dados. Um exemplo comum é a simulação de Monte Carlo.

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

Figura 7: função que conta n jogadas de uma moeda

```
julia> @everywhere include("count_heads.jl")

julia> a = @spawn count_heads(100000000)
Future(2, 1, 6, Nullable{Any}())

julia> b = @spawn count_heads(100000000)
Future(3, 1, 7, Nullable{Any}())

julia> fetch(a)+fetch(b)
100001564
```

Figura 8: função que conta n jogadas de uma moeda

A imagem 4 mostra como executar 200000000 ”jogadas”e contar quantas foram caras em 2 processos. Aqui as iterações ocorrem de forma independente nos processos e seu resultado é combinado usando alguma função (nesse caso, a função +).

```
nheads = @parallel (+) for i = 1:200000000
    Int(rand(Bool))
end
```

Figura 9: função que conta n jogadas de uma moeda

Ainda, se observamos a imagem 4, percebemos que estamos citando explicitamente quantos processo irão executar a nossa função ”count heads”, porém podemos generalizar, como mostra exemplo da imagem 5,

para que a função seja executada em quantos processos tivermos disponíveis em nosso "cluster".

3.4.4 Channels

Um *Channel* pode ser visualizado como um pipe. Eles podem ser usados para passar dados entre duas tarefas que estão em execução, em particular as que envolvem *I/O*.

- Múltiplos *writers* em diferentes tarefas podem escrever no mesmo *channel* concorrentemente via chamadas de *put!()*.
- Múltiplos leitores em diferentes tarefas podem ler um mesmo *channel* concorrentemente via *take!()*.
- *Channels* são criados via o construtor *Channel{T}(sz)*. Ele só vai ter objetos do tipo *T*, se o tipo não for especificado, ele poderá ter objetos de todos os tipos.Já o *sz* se refere ao número máximo de elementos que pode ter em determinado momento. Por exemplo *channel(32)* cria um *channel* que pode ter um máximo de 32 objetos ao mesmo tempo.
- Se um *channel* estiver vazio, as chamadas *take!()* serão bloqueadas até algum dado estiver disponível.
- Se um *channel* estiver cheio, as chamadas *put!()* serão bloqueadas até espaço estiver disponível.
- *isready()* testa pra presença de algum objeto no *channel*, enquanto *wait()* espera por um objeto ficar disponível.

3.5 Gerenciador de Pacotes

Julia possui um gerenciador de pacotes integrado par adicionar "add-on"escritos em Julia e também para adicionar bibliotecas externas que podem ser adicionadas pelo sistema padrão do sistema operacional ou compiladas do código fonte.

O gerenciador de pacotes do Julia é um pouco diferente dos convencionais. Isso quer dizer que no gerenciador do Julia, o usuário diz o que ele quer e o próprio gerenciador determina qual versão deve instalar ou remover de cada pacote para satisfazer os requerimentos de modo ótimo. Logo, ao invés do usuário instalar um pacote, ele apenas adicionar o mesmo à lista de requerimentos e o próprio gerenciador decide o que deve ser instalado.

Julia possui um repositório git padrão no qual se encontra os pacotes registrados porém o gerenciador de pacotes do Julia funciona sob os protocolos de um repositório git e desde que um repositório tenha código Julia seguindo o layout padrão para repositório de pacotes, o usuário pode adicionar repositórios não oficiais

de pacotes. Graças ao sistema de git, Julia permite que o usuário desenvolva seus próprios pacotes e edite e melhore pacotes já existentes.

3.6 Compilação

Julia dispõe de um compilador *just-in-time*, ou seja, o código é ao mesmo tempo interpretado, analisado em busca de segmentos que seriam acelerados pela compilação e compilado.

3.7 Licença

Julia é distribuída sob licença MIT, livre e com código aberto.

Referências

[1] Alan Stuart Edelman. Alan stuart edelman - mathematics genealogy project, jun 2017.

[2] Julia computing - about us, jun 2017.

[3] Aadhaar project, jun 2017.

[4] Julia language benchmarks, jun 2017.

[5] Julia types, jun 2017.