A linguagem

e sua comunidade no Rio

Felipe Noronha *felipenoris*

O que é?

- Linguagem de programação com foco em Computação Científica
- Atingiu versão 1.0 em 2018
- Visa resolver o "**Two-Language-Problem**"
  - sintaxe visando Produtividade, como o **Python**
  - alta performance, como o **C**

Linha do Tempo

1957 - Fortran, LISP

1972 - C

1983 - C++

1994 - Python 1.0

1995 - Java

2000 - Python 2.0 / R 1.0

2006 - NumPy 1.0

2018 - Julia 1.0

Julia combina atributos de outras linguagens

C = velocidade

Matlab = notação matemática óbvia

Ruby = dinamismo

Lisp = macros

Python = programação geral com REPL

R = estatística

Perl = processamento de strings

Shell = glue

Principais Características

- Open-Source com licença permissiva (MIT License)
- Sintaxe simples
- Dynamically Typed
- Compilação Just-In-Time (JIT)
- Multiple-Dispatch
- Macros (metaprogramação = código que gera código)
- Garbage-Collected

Principais Características

- Open-Source com licença permissiva (MIT License)
- Sintaxe simples
- Dynamically Typed
- Compilação Just-In-Time (JIT)
- Multiple-Dispatch
- Macros (metaprogramação = código que gera código)
- Garbage-Collected

Produtividade **Performance**

- interatividade
- prototipação
- domain-specific background

- tunning de código de máquina
- controle do uso de memória
- computer-science background

- Python
- R
- Matlab

- C
- C++
- Fortran

JIT

Somando elementos de um vetor com Python

In [6]:
```
# solução baseada em NumPy
import numpy as np
x = np.random.rand(1_000_000)
x.sum()
```

Out[6]:
```
500316.50954851654
```

In [7]:
```
# Python puro
def my_sum(x):
    r = 0.0
    for i in x:
        r += i
    return r
```

In [8]:
```
my_sum(x)
```

Out[8]:
```
500316.5095485121
```

Somando elementos de um vetor com Python

In [6]:
```
# solução baseada em NumPy
import numpy as np
x = np.random.rand(1_000_000)
x.sum()
```

Out[6]:
```
500316.50954851654
```

<<-- 2 ms

In [7]:
```
# Python puro
def my_sum(x):
    r = 0.0
    for i in x:
        r += i
    return r
```

<<-- 200 ms

In [8]:
```
my_sum(x)
```

Out[8]:
```
500316.5095485121
```

numpy / numpy

C 52.8% Python 45.8% C++ 1.1% JavaScript 0.1% Fortran 0.1% Shell 0.1%

Tree: 9d0225b800 ▾ numpy / numpy / core / src / multiarray / calculation.c

```
518  */
519 NPY_NO_EXPORT PyObject *
520 PyArray_Sum(PyArrayObject *self, int axis, int rtype, PyArrayObject *out)
521 {
522     PyObject *arr, *ret;
523
524     arr = PyArray_CheckAxis(self, &axis, 0);
525     if (arr == NULL) {
526         return NULL;
527     }
528     ret = PyArray_GenericReduceFunction((PyArrayObject *)arr, n_ops.add, axis,
529                                         rtype, out);
530     Py_DECREF(arr);
531     return ret;
532 }
```

A taste of Julia's JIT in action...

In [2]:
```
x = rand(Float64, 1_000_000)
sum(x)
```

Out[2]: 499791.5665808752

<<-- 0.6 ms

In [12]:
```
function my_sum(x)
    result = zero(eltype(x))
    @simd for i in x
        result += i
    end
    return result
end

my_sum(x)
```

Out[12]: 499791.5665808754

<<-- 0.6 ms

A taste of Julia's JIT in action...

In [7]:
```
using BenchmarkTools # @btime
@btime sum(x)
```

```
  558.322 μs (1 allocation: 16 bytes)
```

In [11]:
```
@btime my_sum(x)
```

```
  553.459 μs (1 allocation: 16 bytes)
```

JuliaLang / julia

Watch 919 | Unstar 21,904 | Fork 3,315

Code | Issues 2,549 | Pull requests 736 | Insights

The Julia Language: A fresh approach to technical computing. https://julialang.org/

julia-language | programming-language | scientific-computing | high-performance-computing | numerical-computation | machine-learning

JIT Overhead

In [2]:
```
function my_sum(x)
    result = zero(eltype(x))
    @simd for i in x
        result += i
    end
    return result
end

@time my_sum(x)
```

```
  0.030082 seconds (30.23 k allocations: 1.688 MiB)
```

Out[2]:
```
500024.9857636496
```

In [3]:
```
@time my_sum(x)
```

```
  0.000965 seconds (5 allocations: 176 bytes)
```

Out[3]:
```
500024.9857636496
```

Por que é diferente de PyPy ou Numba?

- Suporte nativo da linguagem
- Compilador tem mais oportunidades para otimizar:
  - inline de funções
  - propagação de informação sobre tipos e valores de constantes
  - otimizações inter-procedurais
  - oportunidades especializar código para o processador do usuário (ex.: SIMD)

Multiple Dispatch

Orientação a Objetos = "Single-Dispatch"

- Seja "obj" uma instância da classe "**Obj**":
  - obj.**metodo**(x::**Int**, y::**Int**) :: **Int**
  - **metodo**(obj::**Obj**, x::**Int**, y::**Int**) :: **Int**
- A seleção da implementação de "metodo" depende do tipo do primeiro argumento.
- Não é possível definir:
  - obj.**metodo**(x::**String**, y::**String**) :: **Int**
  - obj.**metodo**()

Multiple-Dispatch: exemplo clássico

- os métodos podem ser especializados para os tipos de todos os argumentos.

```
function colide_com( x :: Asteroide, y :: Asteroide )
    # trata colisão Asteroide-Asteroide
end
function colide_com( x :: Asteroide, y :: Espaconave )
   # trata colisão Asteroide-Espaconave
end
function colide_com( x :: Espaconave, y :: Asteroide )
   # trata colisão Espaconave-Asteroide
end
function colide_com( x :: Espaconave, y :: Espaconave )
   # trata colisão Espaconave-Espaconavee
end
```

Fonte: Wikipedia

Multiple-Dispatch: efeitos colaterais

Multiplicação	Python	R	Julia
Escalares	a * b	a * b	a * b
Matricial	np.matmul(A, B)	A %*% B	A * B
element-wize com matrizes	A * B	A * B	A .* B

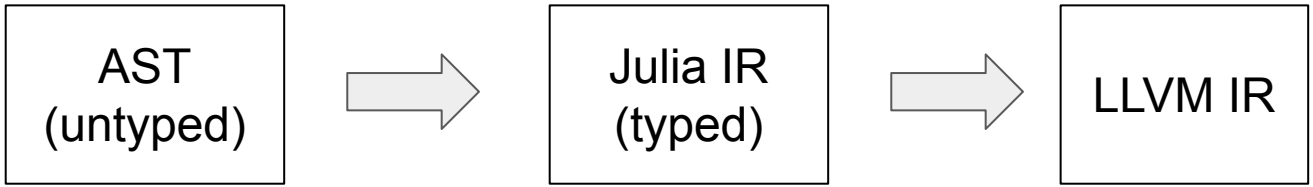
broadcast operator

Metaprogramação

Metaprogramação: o que é e para que serve?

- código que gera código
- Opera sobre as expressões, logo após o "*parsing*"
- Útil para:
  - code templates
  - antecipar processamento para o tempo de compilação
  - extensões da linguagem
  - passar "*hints*" para o compilador (**@inline**, **@simd**)
  - *staged programming*

3 estágios de IR



macros

generated
functions

llvmcall

Macros

```
v = [1,2,3]
@time sum(v)
```

```
  0.000003 seconds (4 allocations: 160 bytes)

6
```

Macros

```
macro time(ex)
    quote
        local stats = gc_num()
        local elapsedtime = time_ns()
        local val = $(esc(ex))
        elapsedtime = time_ns() - elapsedtime
        local diff = GC_Diff(gc_num(), stats)
        time_print(elapsedtime, diff.allocd, diff.total_time,
                   gc_alloc_count(diff))
        println()
        val
    end
end
```

Macros

```
@macroexpand @time sum(v)
```

```
quote
    local #24#stats = (Base.gc_num)()
    local #26#elapsedtime = (Base.time_ns)()
    local #25#val = sum(v)
    #26#elapsedtime = (Base.time_ns)() - #26#elapsedtime
    local #27#diff = (Base.GC_Diff)((Base.gc_num)(), #24#stats)
    (Base.time_print)(
        #26#elapsedtime,
        (#27#diff).allocd, (#27#diff).total_time,
            (Base.gc_alloc_count)(#27#diff))
    (Base.println)()
    #25#val
end
```

Hacking Julia's compiler

parse -> @macroexpand -> @code_lowered -> @code_typed -> @code_llvm -> @code_native

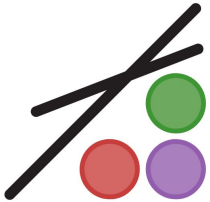
https://www.youtube.com/watch?v=osdeT-tWjzk

Batteries Included

stdlib rica para aplicações científicas

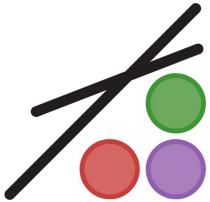
- Linear Algebra
- Statistics
- Pkg
- Collections and Data Structures
- Dates
- DelimitedFiles
- Dynamic Linker
- Distributed, Multi-Threading
- Shared Arrays
- Random Numbers
- Sockets
- Sparse Arrays
- Unit Testing
- Profiling

Bibliotecas "Notáveis"



JuMP.jl

- Linguagem de domínio específico para Otimização Matemática.
- Problemas de Otimização são especificados com sintaxe familiar.
- Suporta vários solvers como *backends* (Cbc, Clp, Gurobi, CPLEX, ...).
- Tipos de Otimização: linear, convexa quadrática, não linear
- Patrocinado pela NUMFOCUS OPEN CODE = BETTER SCIENCE



JuMP.jl

Problema:

max 5x + 3y
 st 1x + 5y <= 3
    0 <= x <= 2
    0 <= y <= 30

```
model = Model(with_optimizer(GLPK.Optimizer))

@variable(model, 0 <= x <= 2)
@variable(model, 0 <= y <= 30)

@objective(model, Max, 5x + 3y)
@constraint(model, 1x + 5y <= 3.0)

JuMP.optimize!(model)

obj_value = JuMP.objective_value(model)
x_value = JuMP.value(x)
y_value = JuMP.value(y)
```

ForwardDiff.jl

- Forward-mode automatic differentiation.
- derivadas, gradientes, hessianas, jacobianas.
- Baseado em Dual Numbers.
- Suporte da comunidade **JuliaDiff**.

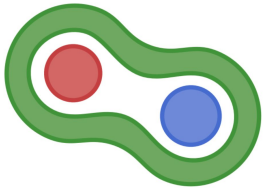
https://www.youtube.com/watch?v=rZS2LGiurKY

```
using ForwardDiff

f(x) = 3*x[1]^2 + 1 # f'(x) = 6x
∇f = x -> ForwardDiff.gradient(f, x)

println( ∇f([0.]) )
println( ∇f([1.]) )
println( ∇f([2.]) )
```

```
[0.0]
[6.0]
[12.0]
```



Zygote.jl

- Diferenciação automática
- Utiliza técnicas de metaprogramação para transformar o problema de calcular derivadas em problema de compilação.
- Resultado é similar a escrever o código fonte das derivadas à mão.
- Ainda em estágio experimental.
- Autor: **Mike Innes** (JuliaComputing)

```
using Zygote
f(x) = 5x + 3

f'(1)
```

```
5
```

```
@code_llvm f'(1)
```

```
;  @ /Users/noronha/.julia/packages/
1:50 within `#36'
define i64 @"julia_#36_13827"(i64) {
top:
  ret i64 5
}
```

Como temos usado

Infra

- Servidor Linux virtualizado
- Imagem no Docker
- https://github.com/felipenoris/math-server-docker
- Jupyter + RStudio
- Compartilhamento de arquivos via NFS

Projetos

- Linguagem para modelagem de contratos financeiros inspirado em **LexiFi** (linguagem Haskell), e **Miletus.jl** (JuliaComputing).

How to write a financial contract

S.L. Peyton Jones and J-M. Eber

This chapter is based closely on "Composing contracts: an adventure in financial engineering", Proceedings International Conference on Functional Programming, Montreal, 2000, with permission from ACM, New York.

1 Introduction

The finance and insurance industry manipulates increasingly complex contracts. Here is an example: the contract gives the holder the right to choose on 30 June 2000 between

D_1 Both of:

D_{11} Receive £100 on 29 Jan 2001.

D_{12} Pay £105 on 1 Feb 2002.

D_2 An option exercisable on 15 Dec 2000 to choose one of:

D_{21} Both of:

D_{211} Receive £100 on 29 Jan 2001.

D_{212} Pay £106 on 1 Feb 2002.

D_{22} Both of:

D_{221} Receive £100 on 29 Jan 2001.

D_{222} Pay £112 on 1 Feb 2003.

Composing Contracts: An Adventure in Financial Engineering

Functional pearl

Simon Peyton Jones, Microsoft Research, Cambridge, simonpj@microsoft.com

Jean-Marc Eber, LexiFi Technologies, Paris, jeanmarc.eber@lexifi.com

Julian Seward, University of Glasgow, v-sewardj@microsoft.com

Abstract

Financial and insurance contracts do not sound like promising territory for functional programming and formal semantics, but in fact we have discovered that insights from programming languages bear directly on the complex subject of describing and valuing a large class of contracts.

We introduce a combinator library that allows us to describe such contracts precisely, and a compositional denotational semantics that says what such contracts are worth. We sketch an implementation of our combinator library in Haskell. Interestingly, lazy evaluation plays a crucial role.

1 Introduction

Consider the following financial contract, C_1: the right to choose on 30 June 2000 between

D_1 Both of:

D_{11} Receive £100 on 29 Jan 2001.

D_{12} Pay £105 on 1 Feb 2002.

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library". And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon for typical combinations of financial contracts (swaps, futures, caps, floors, swaptions, spreads, straddles, captions, European options, American options, ... the list goes on). Treating each of these individually is like having a large catalogue of prefabricated components. The trouble is that someone will soon want a contract that is not in the catalogue.

If, instead, we could define each of these contracts using a fixed, precisely-specified set of combinators, we would be in a much better position than having a fixed catalogue. For a start, it becomes much easier to describe new, unforeseen, contracts. Beyond that, we can systematically analyse, and perform computations over these new contracts, because they are described in terms of a fixed set of primitives.

The major thrust of this paper is to draw insights from the study of functional programming to illuminate the world of financial contracts. More specifically, our contributions are the following:

Exemplo: contrato a termo de moeda estrangeira

$$payoff = S_T - K$$

O contrato pode ser modelado por:

```
c_future_usd = Risk.Core.Future(Date(2019, 1, 2), 1USD, 2.9 * 0.9 / 0.7 * BRL)
```

```
WhenAt
 ├─2019-01-02
 └─Both
    ├─Unit
    │  └─SpotCurrency
    │     └─USD
    └─Give
       └─Scale
          ├─3.7285714285714286
          └─Unit
             └─SpotCurrency
                └─BRL
```

Exemplo: contrato a termo de moeda estrangeira

Calculando o preço inicial do contrato:

```
dt_referencia = Date(2018, 5, 29)

currency_to_curves_map = Dict( "onshore" => Dict( :BRL => :PRE, :USD => :cpUSD ))
model = Risk.Core.StaticHedgingModel(BRL, currency_to_curves_map)

scenario = Risk.Core.FixedScenario()
scenario[Risk.Core.SpotCurrency(USD)] = 2.9BRL # cotação do dolar
scenario[Risk.Core.DiscountFactor(:PRE, Date(2019, 1, 2))] = 0.7 # PU da curva PRE
scenario[Risk.Core.DiscountFactor(:cpUSD, Date(2019, 1, 2))] = 0.9 # PU da curva de cup

attributes = Risk.Core.ContractAttributes(:riskfree_curves => "onshore", :carry_type =>
pricer_future_usd = Risk.Core.Pricer(dt_referencia, model, c_future_usd, attributes)
Risk.Core.price(pricer_future_usd, scenario)
```

```
0.0
```

Exemplo: contrato a termo de moeda estrangeira

E o mapeamento em fatores de risco:

```
Risk.Core.exposures_report(Risk.Core.exposures(pricer_future_usd, scenario))
```

```
3×2 DataFrames.DataFrame
| Row | RISK_FACTOR                          | EXPOSURES |
|-----|--------------------------------------|-----------|
| 1   | SpotCurrency(USD)                    | 2.61      |
| 2   | DiscountFactor(PRE, 2019-01-02)      | -2.61     |
| 3   | DiscountFactor(cpUSD, 2019-01-02)    | 2.61      |
```

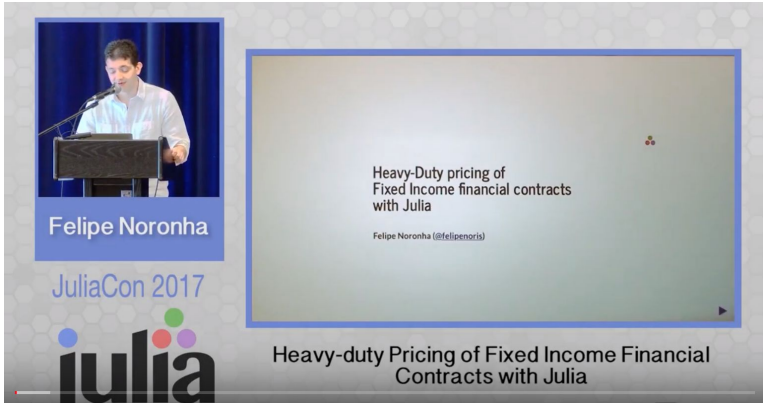
A Comunidade no Rio

Dificuldades

- dispersão geográfica
- Julia é o tipo de linguagem favorita para "não computeiros"
- Julia não é o tipo de linguagem que gera dúvidas
- domínios variados
- technical **people** are not **people people**
- Where is everybody?

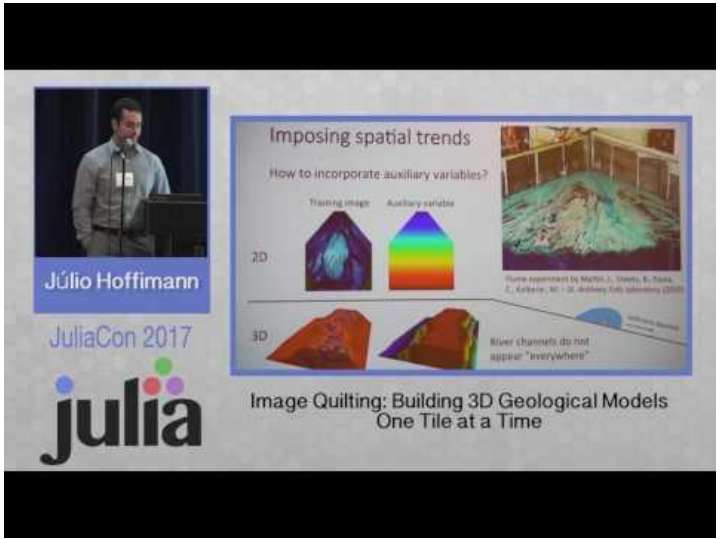
@felipenoris

- XLSX.jl
- Mongoc.jl
- Oracle.jl
- JuliaPackageWithRustDep.jl
- JuliaFinance/BusinessDays.jl
- InterestRates.jl



@juliohm

- GeoStats.jl
- ImageQuilting.jl

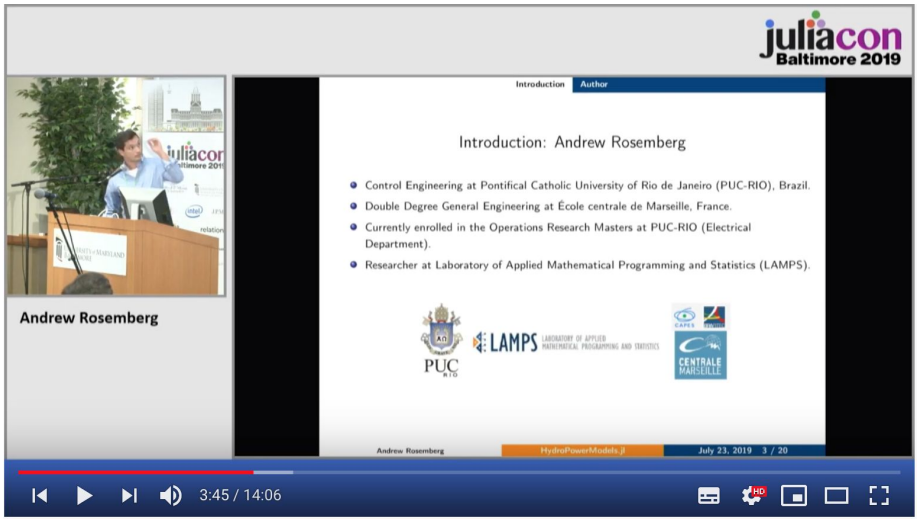


@raphaelsaavedra

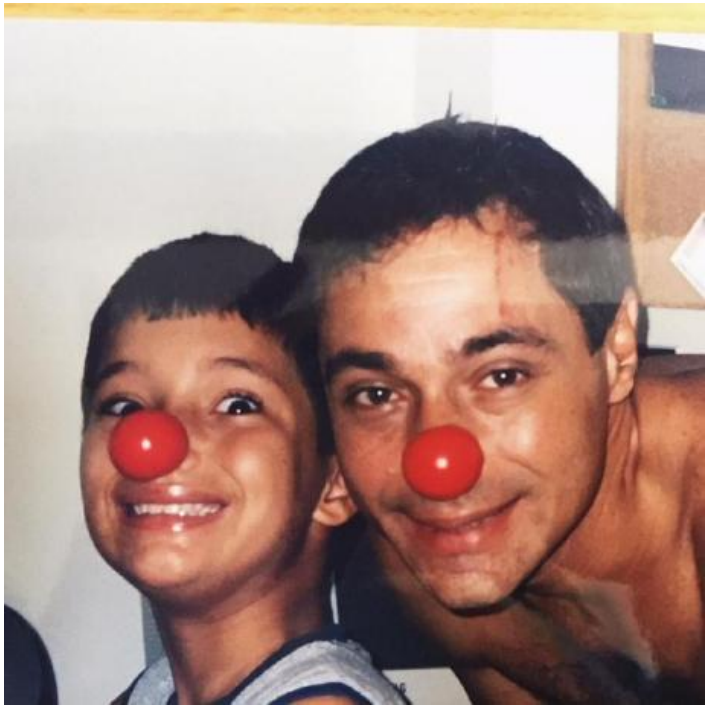
- StateSpaceModels.jl

@andrewrosemberg

- HydroPowerModels.jl



JuliaCon 2019 | HydroPowerModels.jl: A Package for Hydrothermal Economic Dispatch Optimization



@guilhermebodin

- JuliaOpt/Dualization.jl
- StateSpaceModels.jl

@filipebraida

- JuliaRecsys/Persa.jl