A Random Walk through the Julia Language

Avik Sengupta

Julia Computing

19 Nov, 2019

Julia is a fast language

.. but lets not talk about performance ... yet

One Minute introduction to syntax

We will talk a lot about types, but types are optional

In [1]:
```
a = 10
```

Out[1]:
```
10
```

In [2]:
```
f(x) = x^2
```

Out[2]:
```
f (generic function with 1 method)
```

In [3]:
```
f(a)
```

Out[3]:
```
100
```

Duck typing (or generic functions)

In [5]:
```
f(5.5)
```

Out[5]:
```
30.25
```

In [7]:
```
f([1  2; 3  4])
```

Out[7]:
```
2×2 Array{Int64,2}:
  7  10
 15  22
```

In [4]:
```
f("Julia is great! ")
```

Out[4]:
```
"Julia is great! Julia is great! "
```

But types are close when you need them

In [8]:
```
addByCounting(x::Int, y::Int) = repeat("∘", x) * repeat("∘", y)
```

Out[8]:
```
addByCounting (generic function with 1 method)
```

In [9]:
```
addByCounting(2, 3)
```

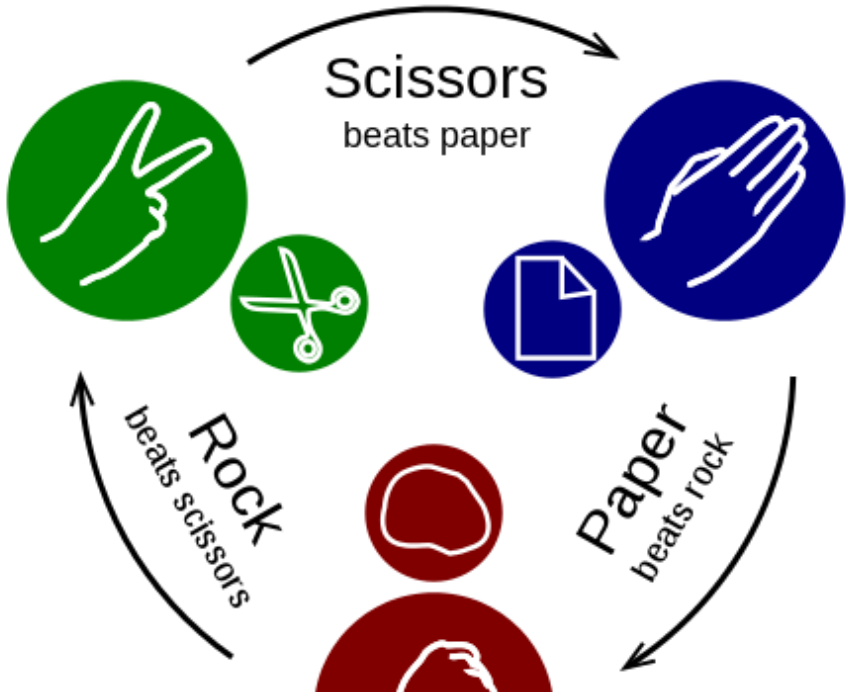
Out[9]:
```
"∘∘∘∘∘"
```

In [10]:
```
addByCounting(2.5, 3.5)
```

```
MethodError: no method matching addByCounting(::Float64, ::Float64)

Stacktrace:
 [1] top-level scope at In[10]:1
```

Rock Paper Scissors



In [1]:

```
abstract type Shape end
struct Rock     <: Shape end

struct Paper    <: Shape end
struct Scissors <: Shape end
```

In [2]:
```
play(x::Paper, y::Rock)     = "Paper wins"
play(x::Paper, y::Scissors) = "Scissors wins"
play(x::Rock,  y::Scissors) = "Rock wins"
```

Out[2]:
```
play (generic function with 3 methods)
```

In [3]:
```
play(x::T, y::T) where {T<: Shape} = "Tie, try again"
play(x::Shape, y::Shape) = play(y, x)
```

Out[3]:
```
play (generic function with 5 methods)
```

In [4]:
```
play(Paper(), Rock())
```

Out[4]:
```
"Paper wins"
```

In [5]:
```
play(Rock(), Paper())
```

Out[5]:
```
"Paper wins"
```

In [6]:
```
play(Rock(), Rock())
```

Out[6]:
```
"Tie, try again"
```

In [9]:
```
play(Scissors(), Scissors())
```

Out[9]:
```
"Tie, try again"
```

In [10]:
```
subtypes(Shape)
```

Out[10]:
```
3-element Array{Any,1}:
 Paper
 Rock
 Scissors
```

In [11]:
```
rand(subtypes(Shape))
```

Out[11]:
```
Rock
```

In [12]:
```
rand(subtypes(Shape))()
```

Out[12]:
```
Rock()
```

In [13]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[13]:
```
"Tie, try again"
```

In [14]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[14]:
```
"Paper wins"
```

In [15]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[15]:
```
"Scissors wins"
```

Why Does this matter?

1. The Expression Problem

A. New objects

In [24]:
```
struct Lizard <: Shape end

play(x::Lizard, y::Rock)     = "Lizard wins"
play(x::Lizard, y::Scissors) = "Lizard wins"
play(x::Lizard, y::Paper)    = "Paper wins"
```

Out[24]:
```
play (generic function with 8 methods)
```

In [17]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[17]:
```
"Scissors wins"
```

In [18]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[18]:
```
"Paper wins"
```

In [19]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[19]:
```
"Paper wins"
```

In [26]:
```
play(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[26]:
```
"Lizard wins"
```

B. New operations

In [27]:
```
up(x::Rock, y::Paper) = Scissors()
up(x::Paper, y::Scissors) = Lizard()

up(x::Scissors, y::Lizard ) = Rock()
up(x::Lizard, y::Rock) = Paper()
up(x::T, y::T) where {T<: Shape} = T()
up(x::Shape, y::Shape) = error("Shapes incompatible for upgrade")
```

Out[27]:
```
up (generic function with 6 methods)
```

In [28]:
```
up(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[28]:
```
Rock()
```

In [29]:
```
up(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

```
Shapes incompatible for upgrade


Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] up(::Scissors, ::Rock) at ./In[27]:6
 [3] top-level scope at In[29]:1
```

In [30]:
```
up(rand(subtypes(Shape))(), rand(subtypes(Shape))())
```

Out[30]:
```
Rock()
```

Why does it matter?

2. Mathematics is heavily polymorphic

In [37]:
```
methods(+)
```

Out[37]: 191 methods for generic function **+**:

- +(x::**Bool**, z::**Complex{Bool}**) in Base at complex.jl:277
- +(x::**Bool**, y::**Bool**) in Base at bool.jl:104
- +(x::**Bool**) in Base at bool.jl:101
- +{T<:AbstractFloat}(x::**Bool**, y::**T**) in Base at bool.jl:112
- +(x::**Bool**, z::**Complex**) in Base at complex.jl:284
- +(a::**Float16**, b::**Float16**) in Base at float.jl:392
- +(x::**Float32**, y::**Float32**) in Base at float.jl:394
- +(x::**Float64**, y::**Float64**) in Base at float.jl:395
- +(z::**Complex{Bool}**, x::**Bool**) in Base at complex.jl:278
- +(z::**Complex{Bool}**, x::**Real**) in Base at complex.jl:292
- +(::**Missing**, ::**Missing**) in Base at missing.jl:96
- +(::**Missing**) in Base at missing.jl:83
- +(::**Missing**, ::**Number**) in Base at missing.jl:97
- +(level::**Base.CoreLogging.LogLevel**, inc::**Integer**) in Base.CoreLogging at logging
- +(c::**BigInt**, x::**BigFloat**) in Base.MPFR at mpfr.jl:406
- +(a::**BigInt**, b::**BigInt**, c::**BigInt**, d::**BigInt**, e::**BigInt**) in Base.GMP at gmp.jl:434
- +(a::**BigInt**, b::**BigInt**, c::**BigInt**, d::**BigInt**) in Base.GMP at gmp.jl:433
- +(a::**BigInt**, b::**BigInt**, c::**BigInt**) in Base.GMP at gmp.jl:432
- +(x::**BigInt**, y::**BigInt**) in Base.GMP at gmp.jl:403
- +(x::**BigInt**, c::**Union{UInt16, UInt32, UInt64, UInt8}**) in Base.GMP at gmp.jl:440
- +(x::**BigInt**, c::**Union{Int16, Int32, Int64, Int8}**) in Base.GMP at gmp.jl:446

Why does it matter?

3. Python adding + to dicts

An approximate pure-Python implementation of the merge operator will be:

```
def __add__(self, other):
    if isinstance(other, dict):
        new = type(self)()  # May be a subclass of dict.
        new.update(self)
        new.update(other)
        return new
    return NotImplemented
def __radd__(self, other):
    if isinstance(other, dict):
        new = type(other)()
        new.update(other)
```

Detour: Performance

All high level abstractions, type checks, boxing etc compiled out

In [38]:
```
a = 1+2im
b = 3+5im
```

Out[38]:
```
3 + 5im
```

In [39]:
```
typeof(a)
```

Out[39]:
```
Complex{Int64}
```

In [40]:
```
@code_native a + b
```

```
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ complex.jl:266 within `+' @ complex.jl:266
        vmovdqu (%edx), %xmm0
        vpaddq  (%esi), %xmm0, %xmm0
; │ @ complex.jl:266 within `+'
        vmovdqu %xmm0, (%edi)
        decl    %eax
        movl    %edi, %eax
        retl
; └
```

In [41]:
```
a = [1+2im, 2+3im, 4+5im, 6+7im]
b = [2+2im, 3+3im, 4+5im, 5+7im]
```

Out[41]:
```
4-element Array{Complex{Int64},1}:
 2 + 2im
 3 + 3im

 4 + 5im
 5 + 7im
```

In [42]:
```
typeof(a)
```

Out[42]:
```
Array{Complex{Int64},1}
```

In [43]:

```
function add(x, y)
    z=zeros(x)
    for i in 1:length(x)
        z[i] = x[1]+z[i]
    end
end

@code_native add(a, b)
```

```
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ In[43]:2 within `add'
        decl    %eax
        subl    $24, %esp
        decl    %eax
        movl    %esi, 16(%esp)
        decl    %eax

        movl    (%esi), %eax
        decl    %eax
        movl    $323962384, %ecx        ## imm = 0x134F4610
        addl    %eax, (%eax)
        addb    %al, (%eax)
        decl    %eax
        movl    %ecx, (%esp)
        decl    %eax
        movl    %eax, 8(%esp)
        decl    %eax
        movl    $242203600, %eax        ## imm = 0xE6FBBD0
        addl    %eax, (%eax)
        addb    %al, (%eax)
        decl    %eax
        movl    %esp, %edi
        movl    $2, %esi
        calll   *%eax
        ud2
; └
```

Why does it matter?

4. Miletus

In [44]:
```
using Miletus
```

In [45]:
```
using Dates
expiry=today()+Day(60)
```

Out[45]:
```
2019-08-25
```

In [46]:
```
eucall=EuropeanCall(expiry, SingleStock(), 105USD)
```

Out[46]:
```
When
├─{==}
│  ├─DateObs
│  └─2019-08-25
└─Either
   ├─Both
   │  ├─SingleStock
   │  └─Give
   │
   │     └─Amount
   │        └─105USD
   └─Zero
```

In [47]:
```
typeof(eucall)
```

Out[47]:
```
Miletus.When{Miletus.LiftObs{typeof(==),Tuple{Miletus.DateObs,Miletus.ConstObs
{Date}},Bool},Miletus.Either{Miletus.Both{SingleStock,Miletus.Give{Miletus.Amo
unt{Miletus.ConstObs{CurrencyQuantity{CurrencyUnit{:USD},Int64}}}},Miletus.Ze
ro}}
```

In [48]:
```
gbm = GeomBMModel(today(), 100.0USD, 0.1, 0.05, .15)
```

Out[48]:
```
Geometric Brownian Motion Model
-------------------------------
S₀ = 100.0USD
T = 2019-06-26
Yield Constant Continuous Curve with r = 0.1, T = 2019-06-26
Carry Constant Continuous Curve with r = 0.05, T = 2019-06-26
σ = 0.15
```

In [49]:
```
value(gbm, eucall)
```

Out[49]:
```
0.9292891123308178USD
```

In [50]:
```
euput=EuropeanPut(expiry, SingleStock(), 95USD)
```

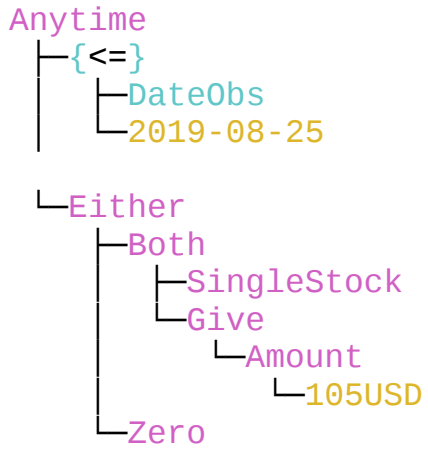
Out[50]:
```
When
├─{==}
│ ├─DateObs
│ └─2019-08-25
└─Either
  ├─Both
  │ ├─Give
  │ │ └─SingleStock
  │ └─Amount
  │   └─95USD
  └─Zero
```

In [51]:
```
value(gbm, euput)
```

Out[51]:
```
0.50790923470501 33USD
```

In [52]:
```
americall = AmericanCall(today()+Day(60), SingleStock(), 105USD)
```

Out[52]:



In [53]:
```
crrm  = CRRModel(today(), expiry, 1000, 100.0USD, 0.1, 0.05, 0.15)
```

Out[53]:
```
BinomialGeomRWModel{CurrencyQuantity{CurrencyUnit{:USD},Float64},Float64,Float
64}
```

In [54]:
```
value(crrm, americall)
```

Out[54]:
```
0.92950065486802 55USD
```

Why Does it matter?

4. Auto Differentiation

(Libraries are composed easily)

$$a + b\varepsilon \quad \text{where} \quad \varepsilon^2 = 0$$
$$f(a + b\varepsilon) = f(a) + bf'(a)\varepsilon$$

In [57]:
```
using Pkg
```

In [58]:
```
using ForwardDiff
import ForwardDiff.Dual
```

In [59]:
```
crrm_d  = CRRModel(today(), expiry, 1000, Dual(100.0, 1), 0.1, 0.05, 0.15)
```

Out[59]:
```
BinomialGeomRWModel{Dual{Nothing,Float64,1},Float64,Float64}
```

In [60]:
```
eucall=EuropeanCall(expiry, SingleStock(), 105);
euput=EuropeanPut(expiry, SingleStock(), 95);
americall=AmericanCall(today()+Day(60), SingleStock(), 105);
```

In [61]:
```
value(crrm_d, americall)
```

Out[61]:
```
Dual{Nothing}(0.9295006548680255,0.2638221381469555)
```

Using reverse mode

In [63]:
```
using Flux
using Flux.Tracker
```

In [64]:
```
Tracker.gradient(x->value(
        CRRModel(today(), expiry, 1000, x, 0.1, 0.05, 0.15) ,


        AmericanCall(expiry, SingleStock(), 105)),
    100)
```

Out[64]:
```
(0.26382213814695565 (tracked),)
```

Using Zygote

In [1]:
```
using Zygote
```

In [3]:
```
f(x) = 5x +3
```

Out[3]:
```
f (generic function with 1 method)
```

In [4]:
```
f(10.), f'(10.)
```

Out[4]:
```
(53.0, 5.0)
```

In [5]:
```
@code_llvm f'(10.0)
```

```
;  @ /Users/aviks/.julia/packages/Zygote/VeaFW/src/compiler/interface.jl:50 wi
thin `#36'
define double @"julia_#36_13501"(double) {
top:
  ret double 5.000000e+00
}
```

In [2]:
```
function s(x)
    t = 0.0
    sign = -1.0
    for i in 1:19
       if isodd(i)
          newterm = x^i/factorial(i)
          abs(newterm)<1e-8 && return t
          sign = -sign
          t += sign * newterm
       end
    end
    return t
  end
```

Out[2]:
```
s (generic function with 1 method)
```

In [71]:
```
@time Zygote.gradient(s, 1.0)
```

```
  1.109152 seconds (2.72 M allocations: 106.908 MiB)
```

Out[71]:
```
(0.5403023037918872,)
```

Example: Key Rate Durations

Key rate durations compute the sensitivities of bond prices with respect to the par rates used to constuct an yield curve. While the bond valuation functions are simple (they are non-stochastic), the calculation of an yield curve from the observed prices include an

interpolation, and an implicit problem using Newton iterations to compute forward prices from par rates. The sensitivities need to be computed over these conversions.

https://gist.github.com/simonbyrne/1ac1d8c769a10fb80b299524b0883590#file-key-rate-duration-ipynb

In [ ]: