SFB 876 Providing Information by Resource-Constrained Data Analysis

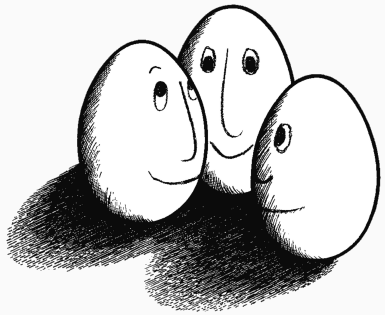
Let's meet Julia!

Mirko Bunse

March 13, 2020

TU Dortmund University, Artificial Intelligence Group

technische universität dortmund



Motivation

dynamic

e.g. Python, R,...

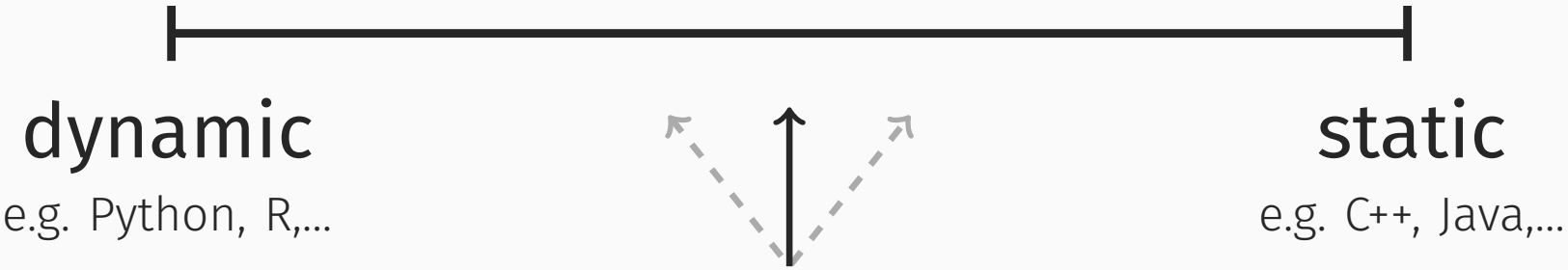
flexible?

static

e.g. C++, Java,...

fast?

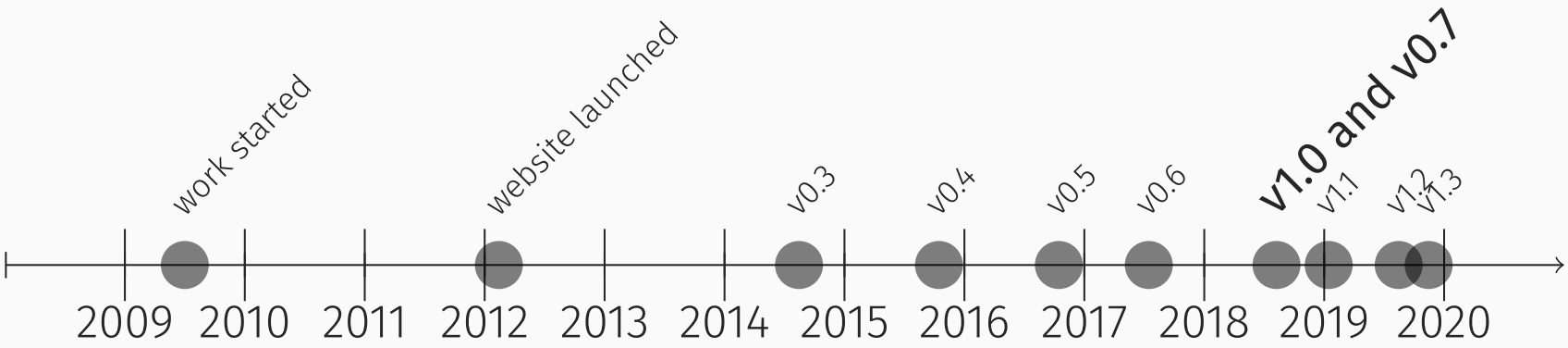
Motivation



flexible + fast:

- optionally typed
- vectorization ‘to go’
- JIT compilation
- multiple dispatch

History



Features:

- High-level, but also efficient
- Free and open source (incl. all packages)
- Rich ecosystem and interoperability
- Designed for parallel computation
- ...

Adopters: Amazon, Apple, Disney, Facebook, Ford, Google, IBM, Microsoft, NASA, Oracle, Blackrock, Uber, ... [Forbes 2017]

Optional Typing

Implicit typing:

Python feeling

```
x = 1 # assign
typeof(x) # Int64

x = "1" # re-assign with a different type
typeof(x) # String

# function block with an UNTYPED parameter
function printtype(x)
    println(typeof(x))
end
```

Optional Typing

Implicit typing:

Python feeling

```
x = 1 # assign
typeof(x) # Int64

x = "1" # re-assign with a different type
typeof(x) # String

# function block with an UNTYPED parameter
function printtype(x)
    println(typeof(x))
end
```

Explicit typing in functions:

compiler hint

```
# one-line function with a TYPED parameter
printtype_int(x::Integer) =
    println("Some Integer: ", typeof(x))

printtype_int(1) # prints the message
printtype_int("1") # throws an error
```

```
MethodError: no method matching printtype_int(::String)
Closest candidates are:
  printtype_int(!Matched::Integer) at In[2]:7
```

Vectorization

Loops:

```
for i in [ 1, 2, 3 ] # regular loop
    println(i)
end

foreach(println, 1:3) # functional style

println.(1:3) # dot syntax
```

fast!

Vectorization

Loops:

fast!

```
for i in [ 1, 2, 3 ] # regular loop
    println(i)
end

foreach(println, 1:3) # functional style

println.(1:3) # dot syntax
```

Parallelization:

```
using Distributed # import package
addprocs(2)       # add worker processes

A = rand(1000, 1000) # random matrix
future = @spawn A^2  # square matrix computed at worker
A_sq = fetch(future) # fetch result

# distribute the flipping of a coin
nheads = @sync @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end
```

```
99999051
```

The impact of

JIT compilation

JIT compilation

Implication: Keep your kernel alive!

- Prefer `julia> include('script.jl')` over `$> julia script.jl`
- Use Revise to update imports
- Consider static compilation for deployment
- Jupyter is an alternative to the REPL

Background: Each method is compiled on its first call

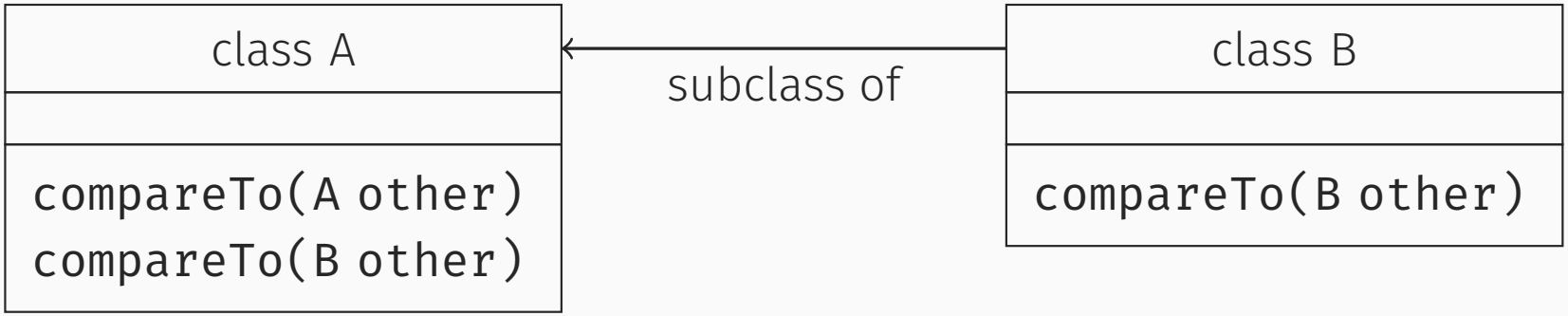
- The first call is ridiculously slow
- Every subsequent call is ridiculously fast

Multiple Dispatch

Background: One function can refer to multiple methods

named reference

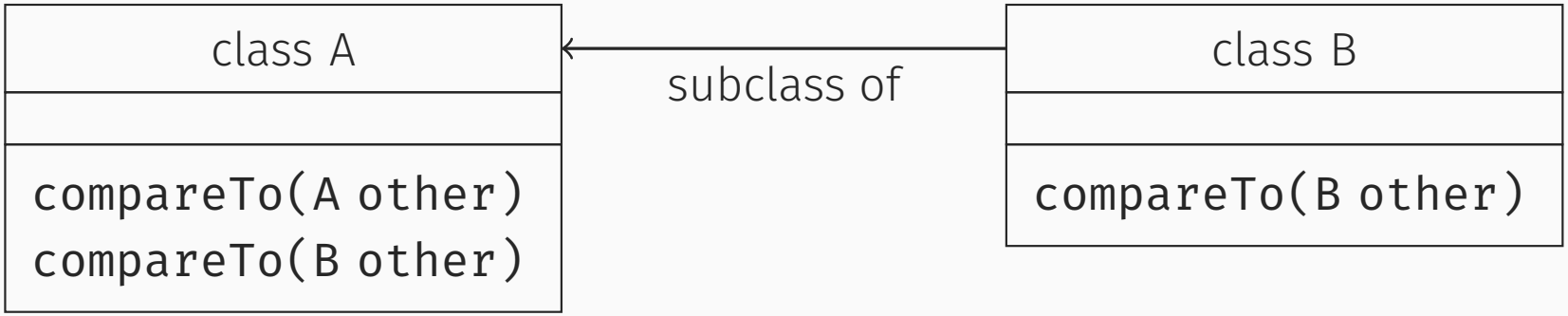
signatures with implementations



Multiple Dispatch

Background: One function can refer to multiple methods

named reference — signatures with implementations



Single (object) dispatch:

```
A thing = funnyFactory.randomAorB();
A otherA = new A();
A otherB = new B();
```
<- object of class B declared as A

`thing.compareTo(otherA):` <- `A.compareTo(A)` or `B.compareTo(A)`

`thing.compareTo(otherB):` <- same, since `otherB` is declared as A

Multiple Dispatch

Single (object) dispatch

At **runtime**, methods are determined **only** from the type of their first argument (usually: their object).

```
def itsFunction(self, a, b):
    ...
```

Multiple Dispatch

Single (object) dispatch

At **runtime**, methods are determined **only** from the type of their first argument (usually: their object).

```
def itsFunction(self, a, b):
    ...
```

Method overloading happens at compile time,
i.e. it depends on the **declared**, not on the actual type!

Multiple Dispatch

Single (object) dispatch

At **runtime**, methods are determined **only** from the type of their first argument (usually: their object).

```
def itsFunction(self, a, b):
    ...
```

Multiple dispatch

At runtime, methods are determined from **all** argument types.

```
def itsFunction(self, a, b):
    ...
```

Method overloading happens at compile time,
i.e. it depends on the **declared**, not on the actual type!

Advantages of MD:

- Great separation of concerns (and backtracking)
- Several object-oriented patterns can be implemented

Multiple Dispatch

Using MD in Julia:

```
function foo(x::Float64)
    return x^2
end

function foo(x::Array{Float64, 1})
    return x.^3 # dot syntax: element-wise operation
end

println(foo(5.0)) # a single float
println(foo([5.0])) # a float array with one element
```

```
25.0
[125.0]
```

Multiple Dispatch

Using MD in Julia:

```
function foo(x::Float64)
    return x^2
end

function foo(x::Array{Float64, 1})
    return x.^3 # dot syntax: element-wise operation
end

println(foo(5.0)) # a single float
println(foo([5.0])) # a float array with one element
```

```
25.0
[125.0]
```

Generics:

```
struct MyGenericType{T_1 <: Number, T_2 <: Any}
    X::AbstractArray{T_1, 2} # 2-dim Array of type T_1
    y::AbstractArray{T_2, 1} # 1-dim Array of any type
end

# constructor: MyGenericType(my_X, my_y)

function foo(bar::AbstractArray{T, 2}) where T
    println("This method accepts any 2-dim array")
end
```

Hack Session

Assignment: Implement a k-NN classifier for the Iris data

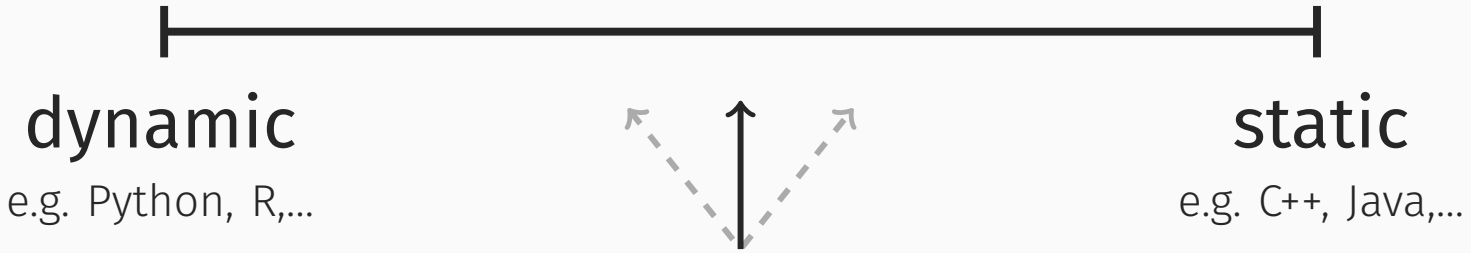
↳ `github.com/mirkobunse/julia-knn-tutorial`

Manual: `https://docs.julialang.org`

Differences from Python:

- Indexing is 1-based & includes the last element
- The **end** keyword terminates blocks (no indentation)
- Argument defaults are evaluated on each function call
- Strings are concatenated with an asterisk `*`
- ...

Discussion



julia

flexible + fast:

- optionally typed
- vectorization ‘to go’
- JIT compilation
- multiple dispatch

↳ `github.com/mirkobunse/julia-knn-tutorial`