Introduction to Julia
It is all about the :Dots...

Stanislav Kazmin[1,2]

[1]Max Planck Institute for Mathematics in the Sciences [2]University of Leipzig

June 28, 2018

Content

Julia is a high-level, high-performance dynamic programming language for numerical computing.[1]

- Download: `https://julialang.org/downloads/`
- Documentation: `https://docs.julialang.org`
- Products based on Julia: `https://juliacomputing.com/`
- JuliaBox (jupyter online): `https://www.juliabox.com/`

[1] *The Julia Language*, https://julialang.org/

Simulation Times Comparison

Simulation Parameters

- Swendsen-Wang cluster algorithm
- 1000 pre-thermalization sweeps
- β from 0.1 to 1.0 with $\Delta\beta = 0.1$
- 100 thermalization sweeps
- 1000 measurements

Attention

The following times do **not** mean that cpp is slower!
It was a real application case study and not a benchmarking!

Simulation Times Comparison

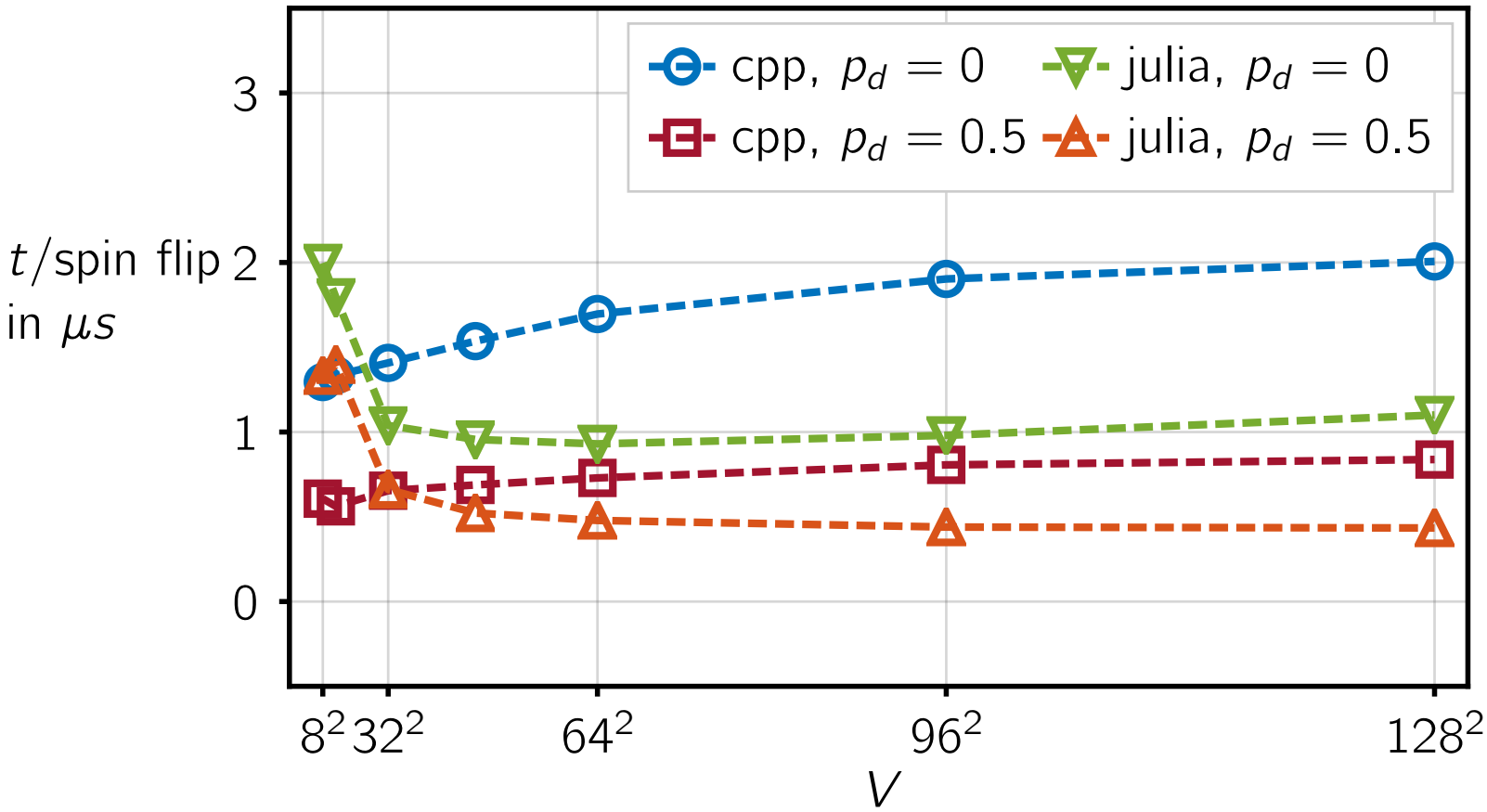


Figure 1: Time per spin flip over lattice volume for the 2d Ising model

Simulation Times Comparison

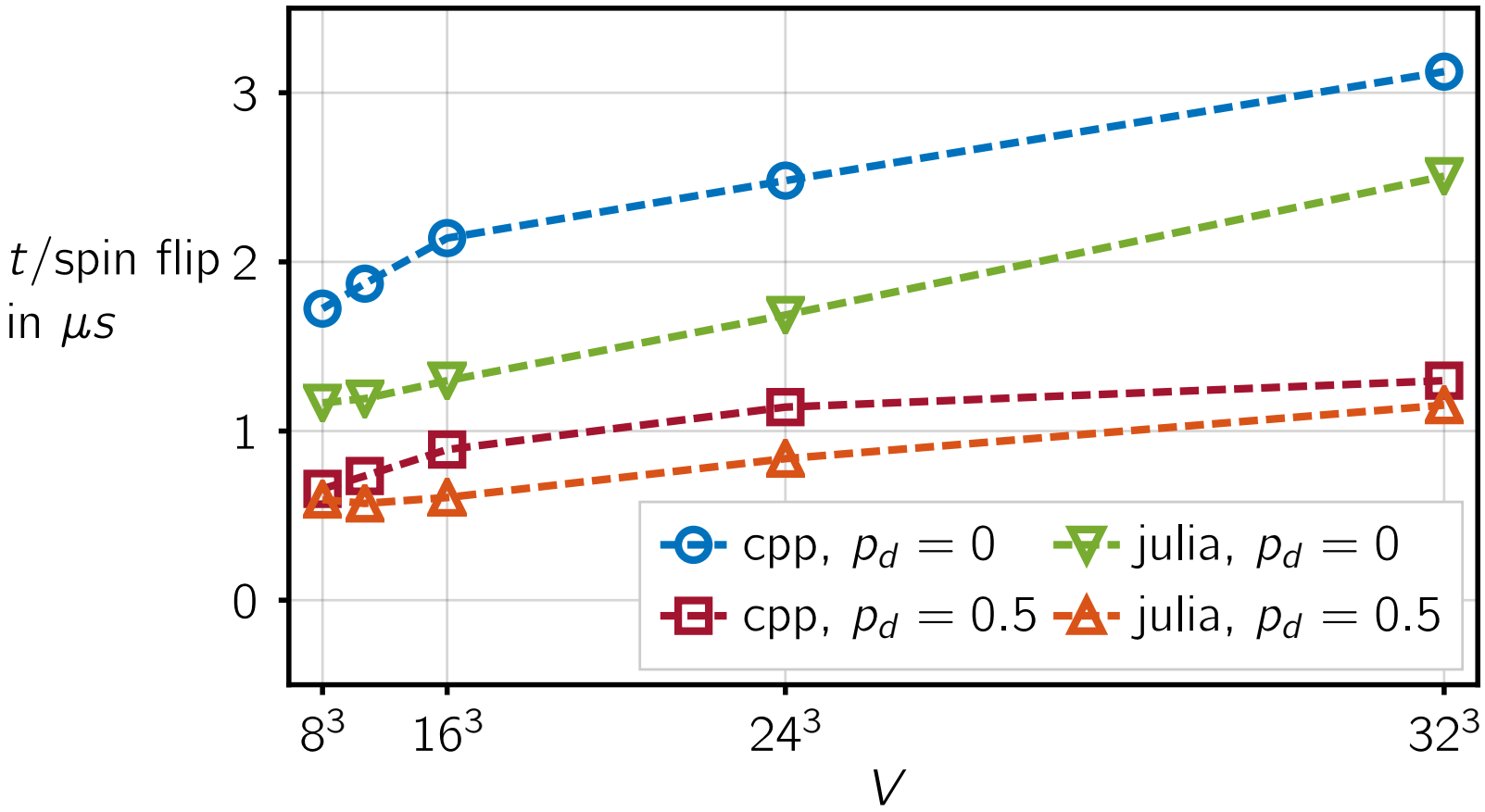


Figure 2: Time per spin flip over lattice volume for the 3d Ising model

Code Differences

aspect	cpp	Julia	ratio
# core lines	≈ 800	≈ 200	4
# core characters	≈ 17000	≈ 5500	3
# util. lines	≈ 600	≈ 70	8
# util. characters	≈ 10000	≈ 2000	5
boundary conditions	hard coded	any mapping	
lattice size and dim.	L^d	$(L_1, \ldots, L_d)$	
lattice elements	int's	arbitrary	

Further advantages of Julia code:

- can save binary files and open with Julia analysis scripts
- no need for a separate config file
- simple debugging and new algorithm testing

Introduction

- current stable release: 0.6.3
- 0.7 beta is out (as 1.0 but with deprecations)
- version 1.0 planned for 2018
- over 1.8 Million Downloads by January 2018
- used in companies like: Amazon, Apple, Disney, Facebook, Ford, Google, Grindr, IBM, Microsoft, NASA, Oracle and Uber[2]

[2] S. D. D'Cunha, *How A New Programming Language Created By Four Scientists Now Used By The World's Biggest Companies*, https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/

Features

Julia Website

- good performance approaching C/C++
- macros and metaprogramming
- designed for parallelism and distributed computation
- user-defined types are as fast and compact as built-ins
- efficient support for Unicode
- MIT licensed: free and open source

Personal Favorites

- Julia is sweet (a lot of syntactic sugar)
- math-like notation
- fast by design
- easy to use but with high optional control
- tabs are not part of the syntax

Reasons to Use

Julia is general propose language with numerical computing in mind

Consider Julia if you[3]

- write your own algorithms
- want to prototype and scale up and analyze using the same language
- need easy access to parallel computing
- want your code be generic but fast
- prefer math-like short syntax and Unicode
- want to use automatic differentiation

[3] *John Pearson | Introduction to Julia for Pythonistas*, `https://www.youtube.com/watch?v=Cj6bjqS5otM`

Why We Created Julia?
In short, because we are greedy.

...
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?)[4]

[4] *Why We Created Julia*,
https://julialang.org/blog/2012/02/why-we-created-julia

History

- started in 2009 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman
- first public release 0.1 in 2012

Goals

Solve the "Two Language Problem"

- easy to learn and flexible
- fast and optimized

Improve the "Expression Problem"[5]

- add **new types** and use **implemented methods** versus add **new methods** for **implemented types** without the need of recompilation

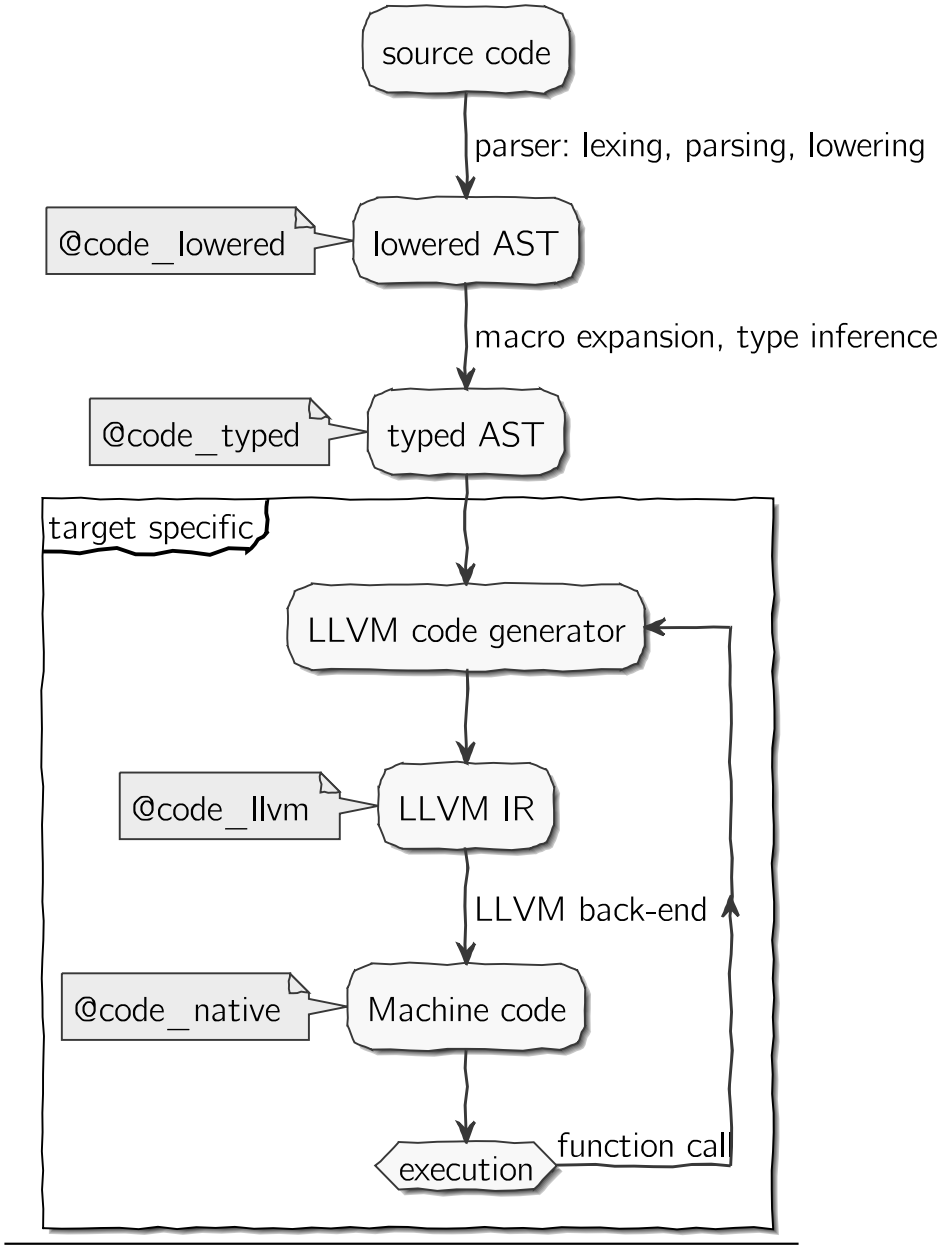
[5] *Expression problem*, in *Wikipedia* (Jan. 28, 2018), `https://en.wikipedia.org/w/index.php?title=Expression_problem&oldid=822799993`

History

Inspiration

- C/C++: speed
- Fortran: Type names, indexing, column majority
- Lisp: metaprogramming, functional programming
- MATLAB: mathematical notation
- Python: simple control flow, API
- R: statistics

Design



- Just in Time Compiler
- at first cal function gets compiled
- subsequent calls use the compiled version
- no conversion to C/C++/Fortran/...
- target specific optimization: GPU[6], Cluster, ...

[6] T. Besard et al., "High-level GPU programming in Julia", (2016)

Core Design Decision[7]

type-stability through specialization via **multiple-dispatch**

- design should be optimizable through LLVM
- Julia is (mainly) written in Julia itself through metaprogramming
- partly in C, assembly and its parser in Scheme ("FemtoLisp")
- functions are first class elements and do **not** belong to a class

[7] *WhyJulia*,
http://ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

Types

Type of a Value

describes how the value is structured (stored in memory)

- compiler needs to know the type for optimization
- only types with exact known memory layout can be used fast
- other types are stored in the heap

Problem with Dynamical Languages

types are generally not known at compile time

Solution in Julia

multiple-dispatch and type-stability

Type Stability

types of the parameters **exactly** define the type of the result

- in Julia **all** types are **generic** and defined in Julia itself[8]
- custom types are as fast as predefined types like `Float64`
- Julia uses a type hierarchy to define dependencies of types
- the types tree is fully connected with `Any` as top element
- **concrete types** (real instances in memory) are always leafs and cannot have sub-types → compiler knows exactly how the memory is structured
- **abstract types** only form the hierarchical structure of the tree

[8] *Types · The Julia Language*, https://docs.julialang.org/en/stable/manual/types/

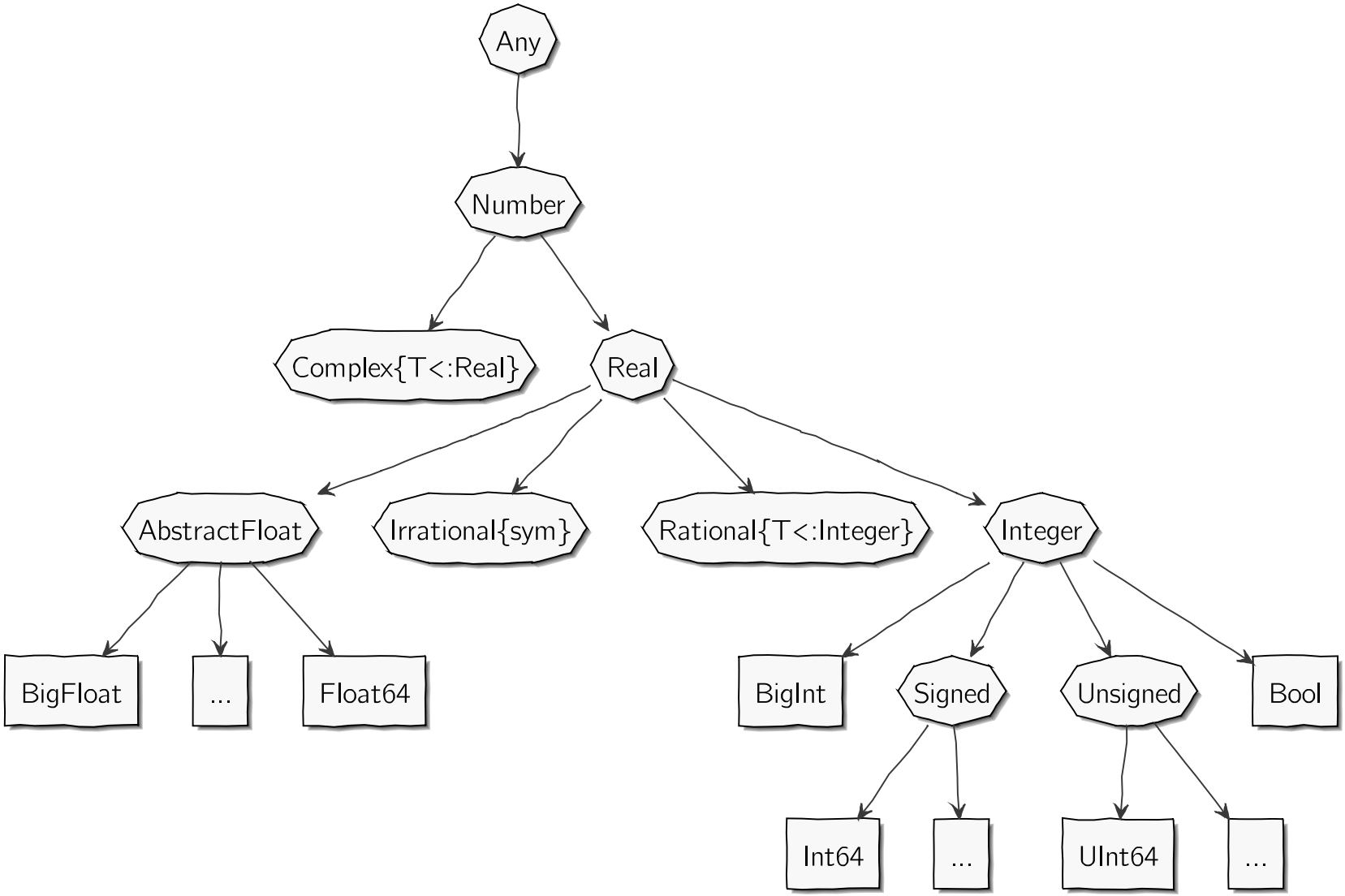


Figure 3: Number type hierarchy in Julia

- **parametric types** are invariant

```julia
struct Point{T} # point with two coords of type T
  x::T
  y::T
end
Float64 <: Real # true
Point{Float64} <: Point{Real} # false
```

Difference to OOP

- no structure inheritance from supertypes
- "if it quacks like a duck, then it might be a duck"[9]
- abstract types are defined by how they act and **not** how they are structured

[9] *Type-Dispatch Design: Post Object-Oriented Programming for Julia*, (May 29, 2017) http://www.stochasticlifestyle.com/type-dispatch-design-post-object-oriented-programming-julia/

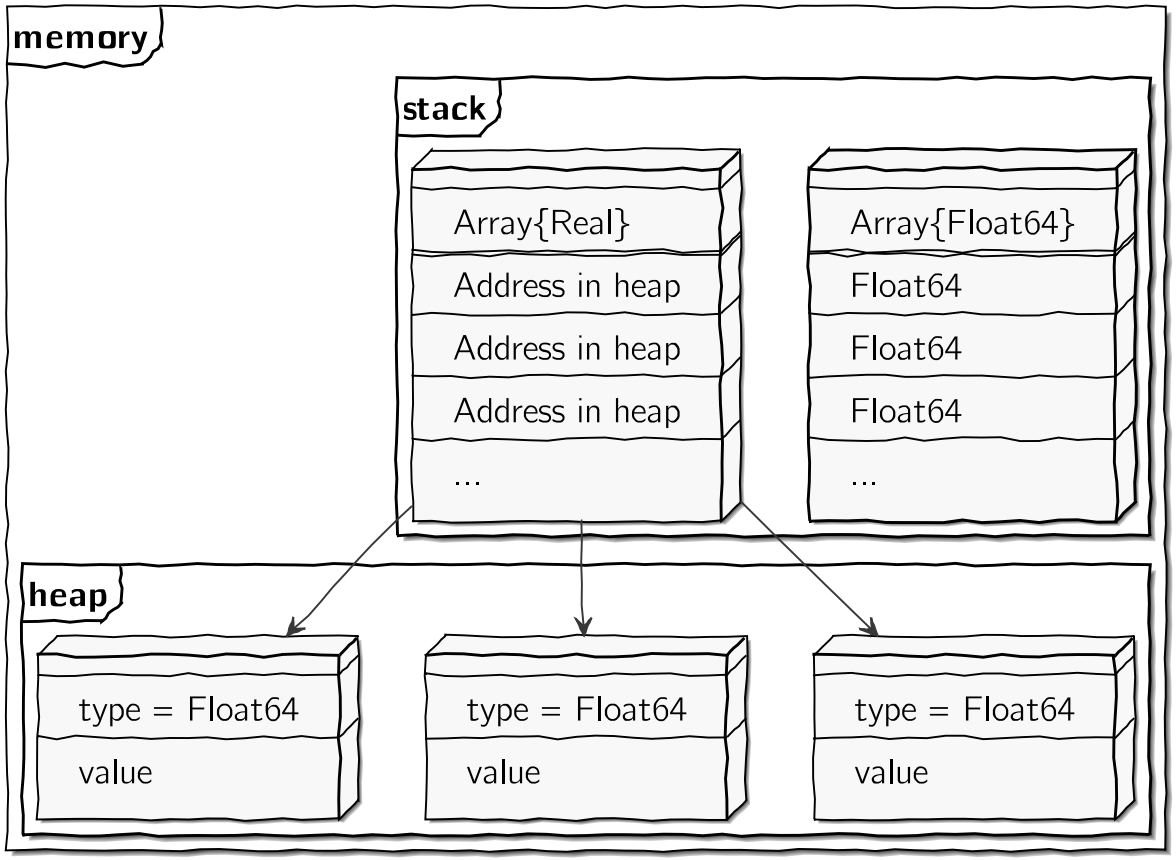


Figure 4: Memory allocation for arrays

Multiple Dispatch

Single Dispatch

search the appropriate function based on one type/subclass of the arguments at **runtime**

C++ call of the form `x->f(y)` will dispatch to the class `X` of `x` and the function `X::f` at runtime

Multiple Dispatch

search the appropriate function based on the type/subclass of **all** arguments at **runtime**

- function overloading happens at **compile time**
- templates are for generic programming at **compile time**
- good explanation for C++ is given here[10]

[10] *A polyglot's guide to multiple dispatch - Eli Bendersky's website*, https://eli.thegreenplace.net/2016/a-polyglots-guide-to-multiple-dispatch/

Algorithms Suitable for Multiple Dispatch

- invoke multiple different types/classes

```
solve(m::Model, bc::BC, solver::Solver) = ...
```

- no natural preferred class to belong to

```
intersect(c::Circle, s::Square) = ...
```

Multiple Dispatch

Basic Principle

on a **function** call → look for types of **all** passed arguments and decide which most specific **method** to use

- types can be hidden but are always present

```
julia>  f(x, y) = 1
        f(x::Int, y) = 2
        f(x::String, y::String) = 3
f (generic function with 3 methods)

julia> f(1, 2) # calls f(x::Int64, y::Int64)
2
julia> f(1.0, 2) # calls f(x::Float64, y::Int64)
1
julia> f("a", 2) # calls f(x::String, y::Int64)
1
julia> f("a", "b") # calls f(x::String, y::String)
3
```

Main Differences to Python

Spoilsport

indexing starts with 1

- blocks closed by `end` keyword
- slice indexing includes the last element
- negative array indexes not allowed
- arrays are column major
- `"Strings"` and `'c'` characters
- more syntax[11] and functional[12] differences

[12] *MATLAB–Python–Julia cheatsheet — Cheatsheets by QuantEcon documentation*, https://cheatsheets.quantecon.org/

[12] *Noteworthy Differences from other Languages — Julia Language X.Y.Z-unknown documentation*, https://juliakorea.github.io/latest/manual/noteworthy-differences.html

Lets Try it Out!

Stanislav Kazmin[1,2]

[1]Max Planck Institute for Mathematics in the Sciences [2]University of Leipzig

June 28, 2018