Type Stability In julia

A Simple and Efficient Optimization Technique

Artem Pelenitsyn

PurPL Seminar
Purdue University
October 19, 2023

Marketing

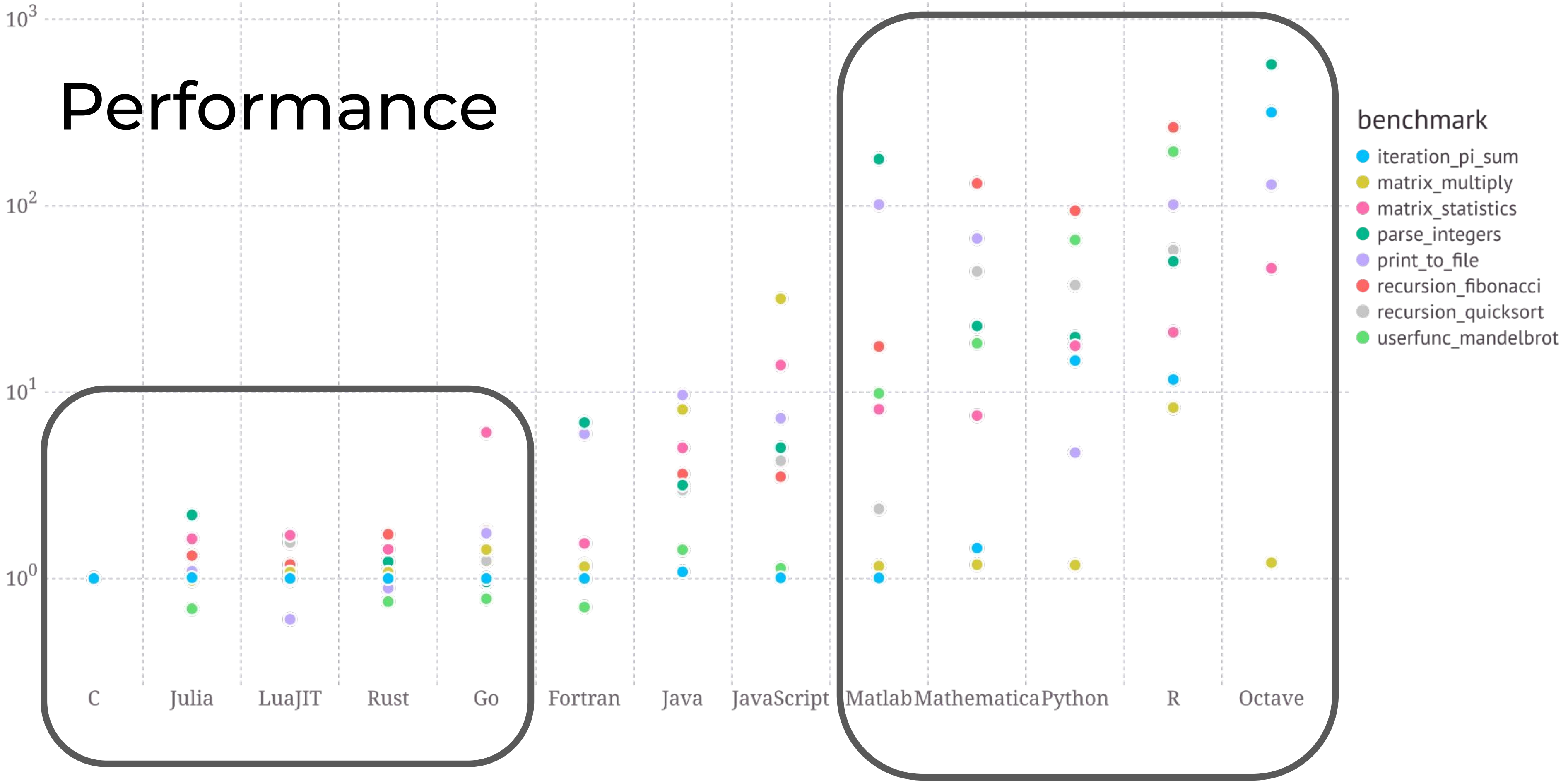
The two-language problem

"Performance"

"Productivity"

Dynamically typed,
garbage-collected,
JIT-compiled (via LLVM) language
developed at MIT with first release
in 2012

Performance



Source: https://julialang.org/benchmarks/

Productivity: multiple dispatch

```
julia> 1 + 2
3
```

dispatches to one of

Method table:

```
# 206 methods for generic function "+" in Base:
[1] +(x::T, y::T) where T<:Union{Int, UInt} in Base at int.jl:87
[2] +(x::T, y::T) where T<:Union{Float32, Float64} in Base at float.jl:383
…
[42] +(z::Complex, w::Complex) in Base at complex.jl:288
…
[52] +(x::Dates.Period, y::Dates.Period) in Dates at periods.jl:367
…
```

Type Stability by example

```
function pisum()
  sum = 0.0
  for j = 1:500
  sum = 0.0
  for k = 1:10000
    sum += 1.0/(k*k)
  end
  end
  sum
end
```

Time: 5 µs

```
function pisum1()
  sum = 0.0
  for j = 1:500
  sum = 0.0
  for k = 1:10000
    sum += id1(1.0/(k*id1(k)))
  end
  end
  sum
end
```

```
function id1(x)
  never ? x : x
end
```

Time (1): 5 µs

```julia
function pisum2()
  sum = 0.0
  for j = 1:500
  sum = 0.0
  for k = 1:10000
    sum += id2(1.0/(k*id2(k)))
  end
  end
  sum
end
```

```julia
function id2(x)
  never ? "❌" : x
end
```

Time (1): 5 µs

Time (2): 12 µs

```
julia> @code_warntype pisum2()
MethodInstance for pisum2()
...
Body::Any
1 ─      (sum = 0.0)
│   %2  = (1:500)::Core.Const(1:500)
...
│   %20 = k::Int64
│   %21 = Main.id2(k)::Union{Int64, String}
│   %22 = (%20 * %21)::Any
│   %23 = (1.0 / %22)::Any
...
```

```
julia> @code_warntype id2(1)
MethodInstance for id2(::Int64)

Body::Union{Int64, String}
1 ─  goto #3 if not Main.never
2 ─  return x
3 ─  return "❌"
```

```
function id2(x)
  never ? x : “❌”
end
```

Type Stability – a key to performance?

- A method is **type stable**
  if for all calls to the method,
  the compiler can infer a precise return type

- `id2` is not type stable
  - `id2: Int64` → **`Int64 or String`**
  - `id2: Float64` → **`Float64 or String`**

- `pisum2` suffers from instability of `id2`

Typeful julia

Concrete types

- Types of values
- No subtypes
- Examples:
  - Primitive:
    `Bool`, `Int64`, `Float64`, …
  - ```
    struct Add <: Expr
      lhs :: Expr
      rhs :: Expr
    end
    ```

Abstract types

- Heap-allocated
- Opaque for the optimizer
- Examples:
  - `Union{Int, String}`
  - `Any`
  - ```
    abstract type Expr
    end
    ```

Overview of the talk

1. Type-stable code inside **Julia's ecosystem**: amount, patterns
2. **Formal correspondence** between type stability and code optimizations
3. Type stability **statically** and subtyping in Julia

Overview of the talk

1. Type-stable code inside **Julia's ecosystem**: amount, patterns
2. **Formal correspondence** between type stability and code optimizations
3. Type stability **statically** and subtyping in Julia

Corpus Analysis: Methodology

- Consider registered Julia packages, sort by GitHub stars, take top 1K
- Run their test suites (760 succeeded)
- Analyze method instances left in the VM, record:
  - type stability,
  - method size,
  - amount of control flow,
  - ...

Quantitative Analysis

Top 10:

Package	Methods	Instances	Stable
DifferentialEquations	1355	7381	70%
Flux	381	4288	76%
Gadfly	1100	4717	81%
Gen	973	2605	64%
Genie	532	1401	93%
IJulia	39	136	84%
JuMP	2377	36406	83%
Knet	594	9013	16%
Plots	1167	5377	74%
Pluto	727	2337	80%

Top 1K:

	Stable
Mean	74%
Median	80%
Std. Dev.	22%

Qualitative analysis: patterns of stability

- Type-constant functions:

```
function is_even(x)
  mod(x, 2) == 0; end
```

- Generic transformations:

```
function union(
      a::Multiset{T},b::Multiset{T}) where T
 ... end
```

- Generic functions inspecting values of the parameter type

```
function head(v::Vector{T}) where T
  v[1] end
```

Qualitative analysis: a pattern of instability

```
abstract type Expr   end
struct Add <: Expr … end
struct Lit <: Expr … end
...
function parse(input :: String) :: Expr
  ...
end
```

Overview of the talk

1. Type-stable code inside **Julia's ecosystem**: amount, patterns
2. **Formal correspondence** between type stability and code optimizations
3. Type stability **statically** and subtyping in Julia

julia

```julia
function f(x, y)
  h(g(x, y), 42)
end
```

Our model

Jules

```
f(%0::Any, %1::Any)
  %2 = g(%0, %1)
  %3 = 42
  %4 = h(%2, %3)
```

Jules features:

- Julia’s notion of type imprecision
- a special syntactic form for direct method calls

Method calls in Jules

Dispatched

vs

direct calls

in Jules

`g(%n, %m)` dispatches to one of

`g!{T3,T4}(%n, %m)` calls directly

Method table:

```
g(%0,     %1) …
g(%0::T1, %1::T2)
…
g(%0::T3, %1::T4)
…
```

Jules' Two Semantics

- Dynamic-dispatch semantics – baseline (no direct calls)
- Type-specializing, devirtualizing JIT compiler

Theorem 4.10 (Correctness ot JIT). *For any original well-formed method table* M *the following holds:*

$$\epsilon\ \overline{\mathsf{st}},\ \mathsf{M} \rightarrow^*_{\mathcal{D}} \mathsf{E}\ \epsilon,\ \mathsf{M} \quad \Longleftrightarrow \quad \epsilon\ \overline{\mathsf{st}},\ \mathsf{M} \rightarrow^*_{\mathsf{JIT}} \mathsf{E}\ \epsilon,\ \mathsf{M}',$$

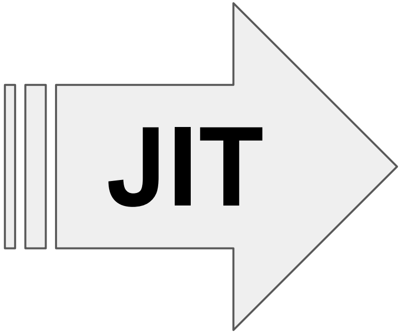
1. Method specialization for concrete types

Program:

```
f(7, 3.14)
```

Method table:

```
f(%0, %1) ...
g(%0, %1) ...
  ...
```



Program:

```
f(7, 3.14)
```

Extended method table:

```
f(%0, %1) ...
g(%0, %1) ...
f(%0::Int,
  %1::Float) ...
  ...
```

2. Devirtualization

JIT-compile `f` for concrete types `T0`, `T1`

Method table:

```
f(%0, %1)
  %2 = g(%0, %1)
  %3 = 42
  %4 = h(%2, %3)
...
```

Type inference result:

```
f(%0::T0,%1::T1)
  %2 = … ::Any
  %3 = … ::Int
  %4 = … ::T4
...
```

Extended method table:

```
f(%0::T0, %1::T1)
  %2 = g!{T0,T1}(%0, %1)
  %3 = 42
  %4 = h(%2, %3)
g(%0::T0, %1::T1) …
...
```

direct call

type inference (oracle)

optimization

Type Stability vs Type Groundedness

Type stable

- enables optimizations of the clients

Type grounded

- relies on type-stable callees (**theorem**)
- enables *full devirtualization* (**theorem**)

T_1	T_1
T_2	T_2
...	...
T_{N-1}	T_{N-1}
T_N	T_N

return type is concrete

all types are concrete

Overview of the talk

1. Type-stable code inside **Julia's ecosystem**: amount, patterns
2. **Formal correspondence** between type stability and code optimizations
3. Type stability **statically** and subtyping in Julia

The Problem:

```julia
function pisum2()
   sum = 0.0
   for j = 1:500
    sum = 0.0
    for k = 1:10000
       sum += id2(
         1.0/(k*id2(k)))
    end
   end
   sum
end
```

```
julia> @code_warntype pisum2()
MethodInstance for pisum2()
  from pisum2() in Main at REPL[4]:1
Arguments
  #self#::Core.Const(pisum2)
Locals
  @_2::Union{Nothing, Tuple{Int64, Int64}}
  sum::Float64
  @_4::Union{Nothing, Tuple{Int64, Int64}}
  j::Int64
  k::Int64
Body::Float64
1 ─       (sum = 0.0)
│   %2  = (1:500)::Core.Const(1:500)
│         (@_2 = Base.iterate(%2))
│   %4  = (@_2::Core.Const((1, 1)) === nothing)::Core.Const(false)
│   %5  = Base.not_int(%4)::Core.Const(true)
└──       goto #7 if not %5
2 ┄ %7  = @_2::Tuple{Int64, Int64}
│         (j = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│         (sum = 0.0)
│   %11 = (1:10000)::Core.Const(1:10000)
│         (@_4 = Base.iterate(%11))
│   %13 = (@_4::Core.Const((1, 1)) === nothing)::Core.Const(false)
│   %14 = Base.not_int(%13)::Core.Const(true)
└──       goto #5 if not %14
3 ┄ %16 = @_4::Tuple{Int64, Int64}
│         (k = Core.getfield(%16, 1))
│   %18 = Core.getfield(%16, 2)::Int64
│   %19 = sum::Float64
│   %20 = k::Int64
│   %21 = Main.id2(k)::Union{Int64, String}
│   %22 = (%20 * %21)::Int64
│   %23 = (1.0 / %22)::Float64
│   %24 = Main.id2(%23)::Union{Float64, String}
│         (sum = %19 + %24)
│         (@_4 = Base.iterate(%11, %18))
│   %27 = (@_4 === nothing)::Bool
│   %28 = Base.not_int(%27)::Bool
└──       goto #5 if not %28
4 ─       goto #3
5 ┄       (@_2 = Base.iterate(%2, %9))
│   %32 = (@_2 === nothing)::Bool
│   %33 = Base.not_int(%32)::Bool
└──       goto #7 if not %33
6 ─       goto #2
7 ┄       return sum


julia> 
```

Attempt at Type Stability Inference

```
function id(x)
  x
end
```

T concrete

T concrete

```
function inc(x)
  x + 1
end
```

T concrete

X unknown

Insight: julia already has a type inferencer

Compile `f` for concrete types `T0`, `T1`

Method table:

```
f(%0, %1)
  ...
```

Type inference result:

```
f(%0::T0,%1::T1)
  %2 = … ::Any
  %3 = … ::Int
  %4 = … ::T4
...
```

Extended method table:

```
...
```

type inference (oracle)

optimization

Type Stability Inference

1. Get the `Method` object

2. Get its input type

3. Find a concrete subtype of the input type

4. Run Julia’s type inference

5. Check return type for concreteness

6. goto 3

Type Stability Inference: Example

```
julia> m = @which length([1,2,3])
length (a::Array) in Base at array.jl:215
julia> m.sig
Tuple{ typeof(length), Array }
julia> t = Array{Float64, 1}
julia> r = code_typed(m, t)...
Int64
julia> isconcretetype(r)
true
```

Type Stability Inference

1. Get the `Method` object
2. Get its input type
3. Find a concrete subtype of the input type
4. Run Julia’s type inference
5. Check return type for concreteness
6. goto 3

```
julia> subtypes(Number)
 Complex
 Real
```

Julia's type language

```
t ::= name
  | Union{t1..tn}
  | Tuple{t1..tn}
  | t where t1<:T<:t2
  | t{t1..tn}
  | T
  | Any
```

```
julia> subtypes(Union{Number, Char})
[]

julia> subtypes(Tuple{Number,Char})
[]

julia> subtypes(Array)
[]
```

My solution: `direct_subtypes`

```
julia> direct_subtypes(Union{Number,Char})
 Number
 Char
julia> direct_subtypes(Tuple{Number,Char})
 Tuple{Complex,Char}
 Tuple{Real,Char}
```

Subtyping bounded existentials

```
julia> direct_subtypes(
           Ptr{T} where T<:Number)
 Ptr{Float32}
 Ptr{Float16}
 Ptr{Float64}
 Ptr{Bool}
 Ptr{BigInt}
 Ptr{Int32}
 ...
```

Instantiate the variable to **all subtypes**, not only `direct_subtypes`

Subtyping **un**bounded existentials

Subtypes of `Array` a.k.a.

`Array{T,N} where Bot<:T<:`**`Any`**` where Bot<:N<:`**`Any`**

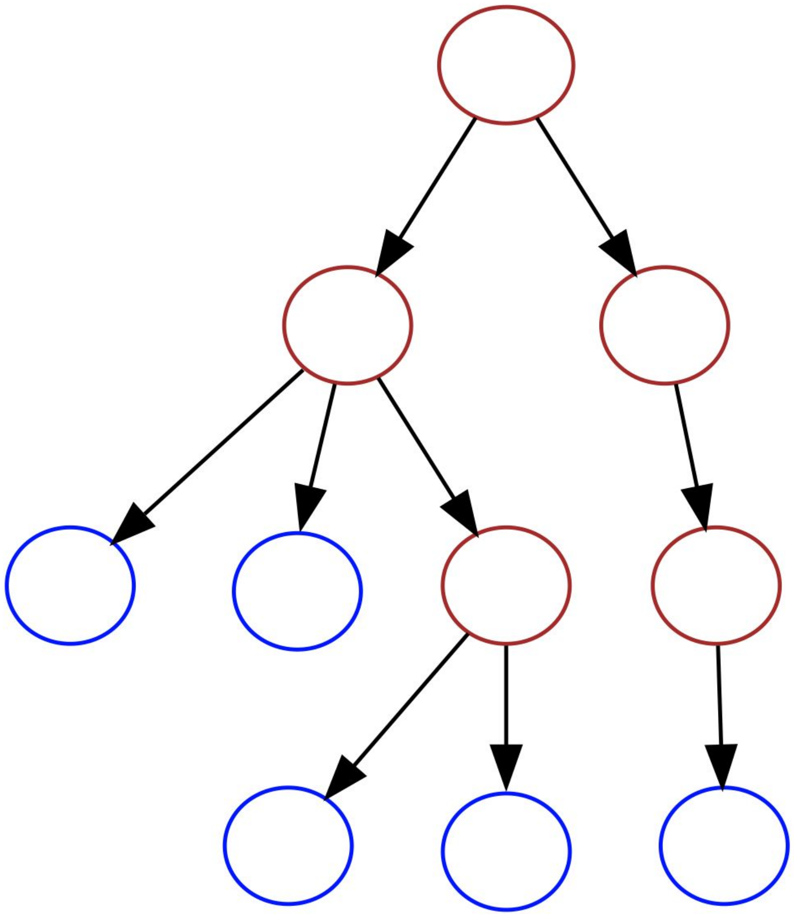
- `Array{Int,1}`
- `Array{Array{Int,1},1}`
- `Array{Array{Array{Int,1},1},1}`
- `etc.`

Handling Any, Idea #1: type inference with abstract types

```
julia> m = @which length([1,2,3])
length (a::Array) in Base at array.jl:215

julia> code_typed(m, (Array,))
Int64
```

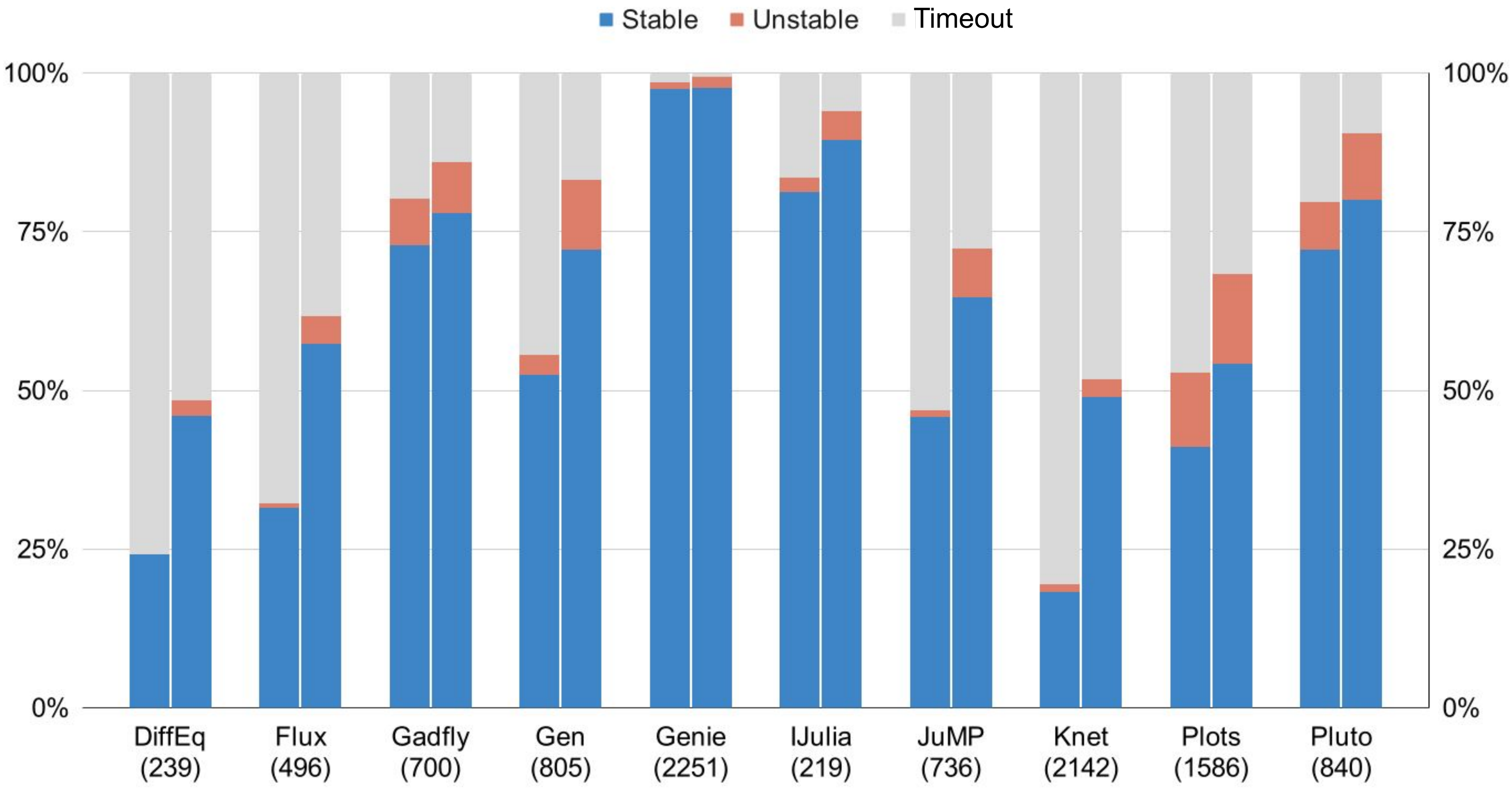
Handling `Any`, Idea #2: sample concrete types



Handling Any, Idea #2: sample concrete types

package	module	type name	occurs
DifferentialEquations	Core	Nothing	356045
DifferentialEquations	Core	Float64	219192
DifferentialEquations	Core	Vector{Float64}	65694
DifferentialEquations	Core	Int64	42132
DifferentialEquations	Core	Matrix{Float64}	39271
JuMP	Core	Float64	26626
DifferentialEquations	Base	Rational{Int64}	25793
DifferentialEquations	Core	Bool	25466
DifferentialEquations	Core	Tuple{}	24494
DifferentialEquations	SciMLBase	typeof(SciMLBase.DEFAULT_OBSERVED)	21798
DifferentialEquations	LinearAlgebra	LinearAlgebra.UniformScaling{Bool}	21437
DifferentialEquations	DiffEqBase	DefaultLinSolve	18495
DifferentialEquations	SciMLBase	SciMLBase.AutoSpecialize	16009
DifferentialEquations	SciMLBase	SciMLBase.NullParameters	14020
DifferentialEquations	Base.MPFR	BigFloat	12411
DifferentialEquations	OrdinaryDiffEq	OrdinaryDiffEq.var"#lorenz#583"	12390
DifferentialEquations	SciMLBase	ODEFunction{true, SciMLBase.AutoSpe	12389
DifferentialEquations	Base	Val{:forward}	12342

Evaluation: no sampling vs sampling



Evaluation: tracing vs approximation

Top 10 packages	Type stable acc. to tracing	Type stable acc. to approximation
Mean	72%	69%
Median	78%	68%

Overview of the talk

1. Type-stable code inside **Julia's ecosystem**: amount, patterns
2. **Formal correspondence** between type stability and code optimizations
3. Type stability **statically** and subtyping in Julia

Future work

- Evaluation of the approximation engine on a large-scale app, tailoring the types database for it
- A tool for fixing type instabilities
- Garbage code collection

Papers

1. *Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation*
   OOPSLA 2021
   By **Artem Pelenitsyn**, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek
2. *Julia Subtyping: A Rational Reconstruction*
   OOPSLA 2018
   By Francesco Zappa Nardelli, Julia Belyakova, **Artem Pelenitsyn**, Benjamin Chung, Jeff Bezanson, and Jan Vitek
3. *Approximating Type Stability in the Julia JIT (Work in Progress)*
   VMIL 2023
   By **Artem Pelenitsyn**