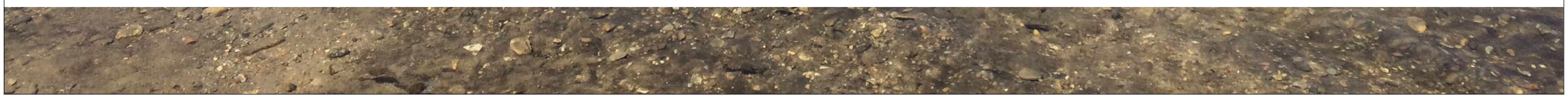
JULIA SUBTYPING RECONSTRUCTED

Francesco Zappa Nardelli — *Inria*

Artem Pelenitsyn, Benjamin Chung, Jan Vitek — *Northeastern U.*

Jeff Bezanson — *Julia Computing*



THE JULIA PROJECT

Julia is a language for the scientific computing community

- Released in 2012, design in Bezanson's thesis (2015)

  Bezanson thesis is issue #8839 in Julia bugzilla

- In 2017, more than 6000 Julia packages on `github`

- Its own conference, JuliaCon, since 2014
- JuliaComputing raised 4.6M$ in 2017

A HIGH-LEVEL, DYNAMIC, MEMORY SAFE LANGUAGE

Higher-order vectorized code as in R

No statically enforced type safety

Loads code with eval

Fortran-like nested loops

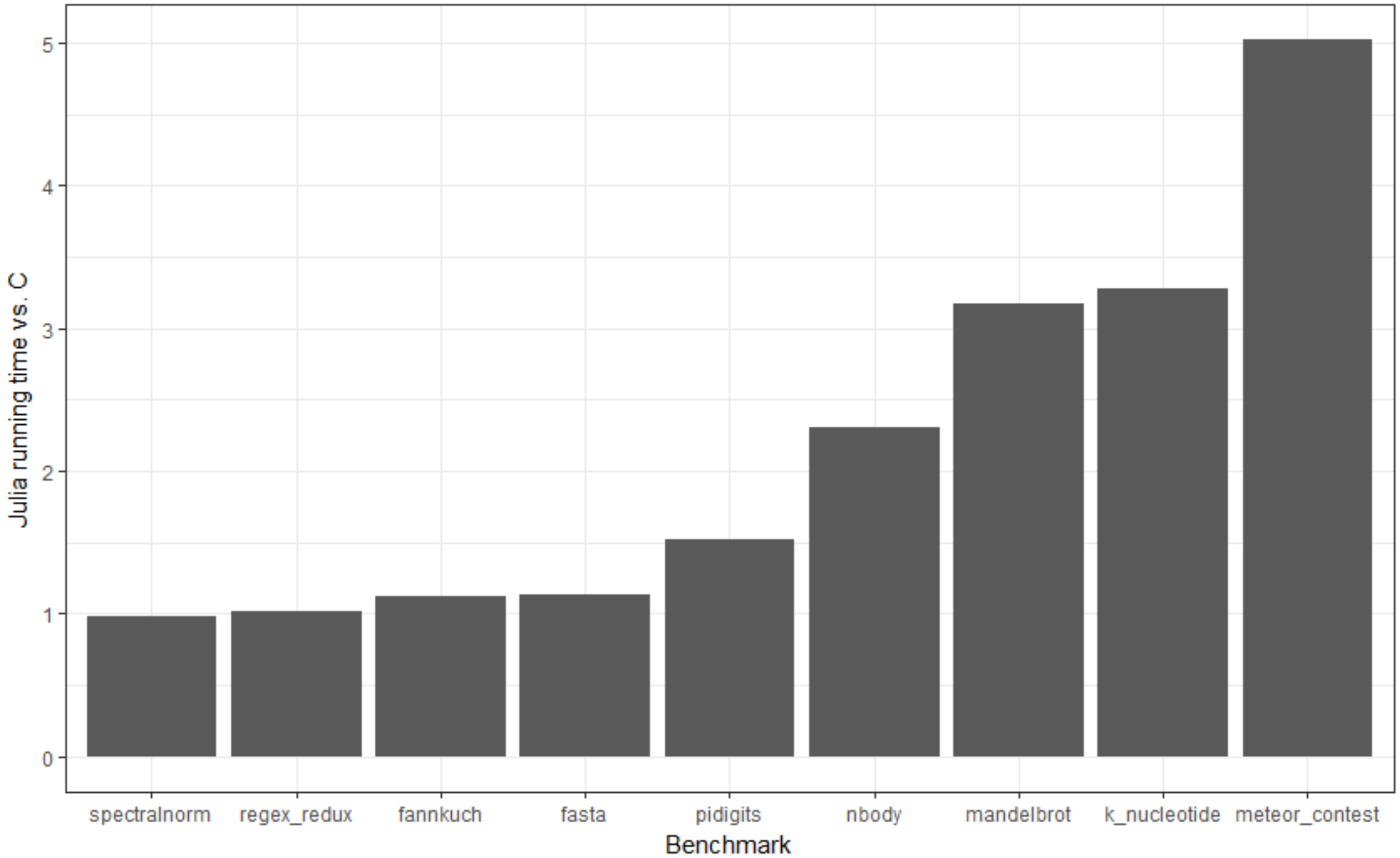
User-defined type definitions and rich type-annotation sublanguage for multiple dispatch

Julia's LLVM backend can be competitive with C or Fortran

julia ADVERTISING

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

ADVERTISING – BENCHMARK GAME




MULTIPLE DISPATCH

OVERLOADING * (SELECTED ENTRIES OUT OF 181 INSTANCES)

```
*(x::Bool, y::Bool)  =
         x & y


*(x::Number, r::Range) =
         range(x*first(r),x*step(r),length(r))


*(x::T, y::T) where T <: Union{Int128,UInt128}  =
         multint(x,y)
```

OVERLOADING * (SELECTED ENTRIES OUT OF 181 INSTANCES)

```
*(A::AbstractArray{T,2},
  B::AbstractArray{S,2}) where {T, S}) =
```

…matrix multiplication code…

```
*(A::AbstractArray{T,2} where T,
  D::Diagonal) =
```

…clever diagonal matrix multiplication code…

```
*(A::Hermitian{Complex{Float64},
              SparseMatrixCSC{Complex{Float64},Ti}},
   B::Union{SparseMatrixCSC{Complex{Float64},Ti},
           SparseVector{Complex{Float64},Ti}}) where Ti
```

…even fancier matrix multiplication code…

OVERLOADING * (SELECTED ENTRIES OUT OF 181 INSTANCES)

```
*(X::Union{Base.ReshapedArray{TX,2,A,MI} where
MI<:Tuple{Vararg{Base.MultiplicativeInverses.SignedMu
ltiplicativeInverse{Int64},N} where N} where
A<:DenseArray, DenseArray{TX,2}, SubArray{TX,2,A,I,L}
where L} where
I<:Tuple{Vararg{Union{Base.AbstractCartesianIndex,
Int64, Range{Int64}},N} where N} where
A<:Union{Base.ReshapedArray{T,N,A,MI} where
MI<:Tuple{Vararg{Base.MultiplicativeInverses.SignedMu
ltiplicativeInverse{Int64},N} where N} where
A<:DenseArray where N where T, DenseArray},

  A::SparseMatrixCSC{TvA,TiA}) where {TX, TvA, TiA}
```

…super fancy matrix multiplication code…

MULTIPLE DISPATCH IN A NUTSHELL

```
myadd(x::Float64, y::Float64) =
      return 0
myadd(c::Union{Float16, Float32, Float64}, x::BigFloat) =
      return 1
myadd(x::Number, y::Number) =
      return 2

myadd(3.0, 4.0)
> 0
myadd(3, 4.0)
> 2
```

TYPES OF METHODS INTERFACES ARE EXTRACTED

```
myadd(x::Float64, y::Float64)

                 Tuple{Float64, Float64}

myadd(c::Union{Float16, Float32, Float64}, x::BigFloat)

     Tuple{Union{Float16, Float32, Float64}, BigFloat}

myadd(x::Number, y::Number)

                  Tuple{Number, Number}
```

TYPES ARE SORTED ACCORDING TO <: (FALSE, BUT WILL DO)

```
Tuple{Float64, Float64}

Tuple{Union{Float16, Float32, Float64}, BigFloat}
```

```
myadd(3.0, 4.0)        Float64, Float64
> 0                   Tuple{Number, Number}
myadd(3, 4.0)          Int64, Float64
> 2
```

STILL A DYNAMIC LANGUAGE

```
f(x::Int64) = return x.g
f(42)
> ERROR: type Int64 has no field g

h(x::Int64) = return x(42)
h(3)
> ERROR: objects of type Int64 are not callable
```

STILL A DYNAMIC LANGUAGE

```
h(x::Int64, y::Any) = return 1
h(x::Any, y::Int64) = return 2
h(3,4)
> ERROR: MethodError: h(::Int64, ::Int64) is ambiguous.
> Candidates:
>  h(x, y::Int64) in Main at REPL[7]:1
>  h(x::Int64, y) in Main at REPL[6]:1
> Possible fix, define
>  h(::Int64, ::Int64)
```

JULIA HISTORY - TYPE SYSTEM PERSPECTIVE

0.1 :

0.2 : *Immutables*

0.3 :

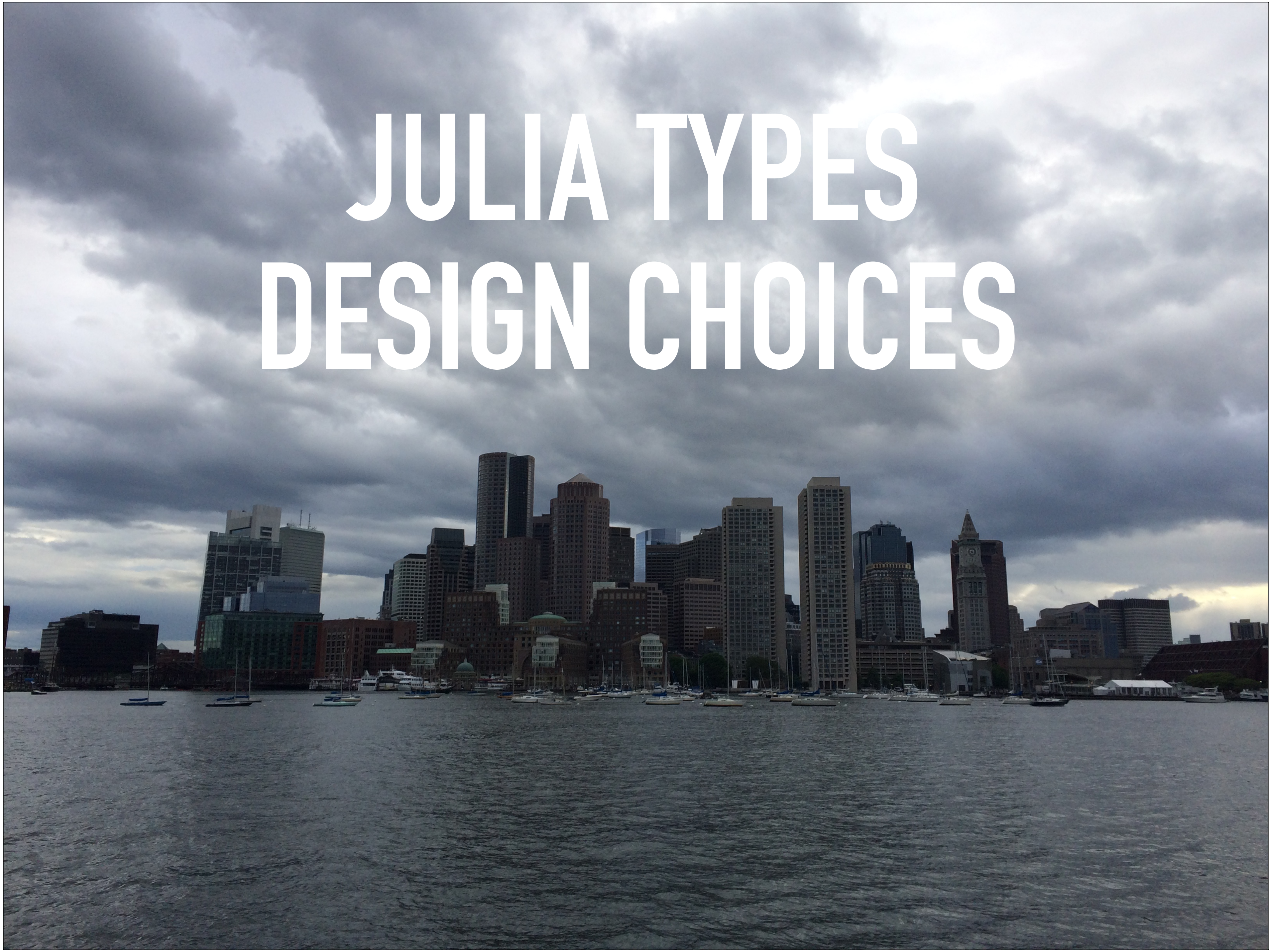
0.4 : *Tuple overhaul: (A,B) vs. Tuple{A,B}*

0.5 : *Vararg{T, N}, nominal function types*

0.6 : *UnionAll types, variable lower bounds, complete overhaul of subtyping*

Subtyping in Julia 0.6:

- Specified by 2400 lines of highly optimised C code
- Inspired by semantic subtyping, but *original design* in the end


JULIA TYPES
DESIGN CHOICES

TYPES ARE NOMINAL

Relations between types are **declared** by the programmer and not inferred from representation

Enables a function to behave differently on types even if these share the same representation (e.g. Bool & Int8)

```
 abstract  type Integer <: Real end

 primitive type Bool <: Integer 8 end

 struct PointRB <: Any
        x::Real
        y::Bool
end
```

No representation specified
Abstract Type

Representation specified
Concrete Type

TYPES ARE PARAMETRIC

User defined types can be parametrised by other types

— *and by values of primitive types (e.g. Int or Bool)*

```
struct Point{T} <: Any
  x::T
  y::T
end

struct Rational{T<:Integer} <: Real
  num::T
  den::T
end

abstract type Vector{T} <: Array{T,1} end
```

UNION AND UNIONALL

Union is an abstract type which *includes as objects all instances of any of its components*

```
Union{Point{Int64}, Point{Int32}, Point{Int16}}
```

Union of types can be **iterated**:

```
Point{T} where T
```

possibly with lower and upper bounds:

```
Vector{T} where T <: Integer
```

UNIONALL: CAREFUL TO PARENS

```
Vector{Point{T} where T}
```

$\neq$

```
Vector{Point{T}} where T
```

PARAMETRIC TYPES ARE INVARIANT

```
struct Point{T} <: Any
  x::T
  y::T
end
```

```
Int64 <: Signed
```

```
Point{Int64}  </:  Point{Signed}
```

Pragmatic design choice: ***values have different representation in memory***

- the former can be represented as a pair of 64-bit values
- the latter is a pair of pointers to individually allocated Signed objects

TUPLES: ABSTRACTIONS OF FUNCTION PARAMETERS

```
Tuple{Int, Float, Any}
```

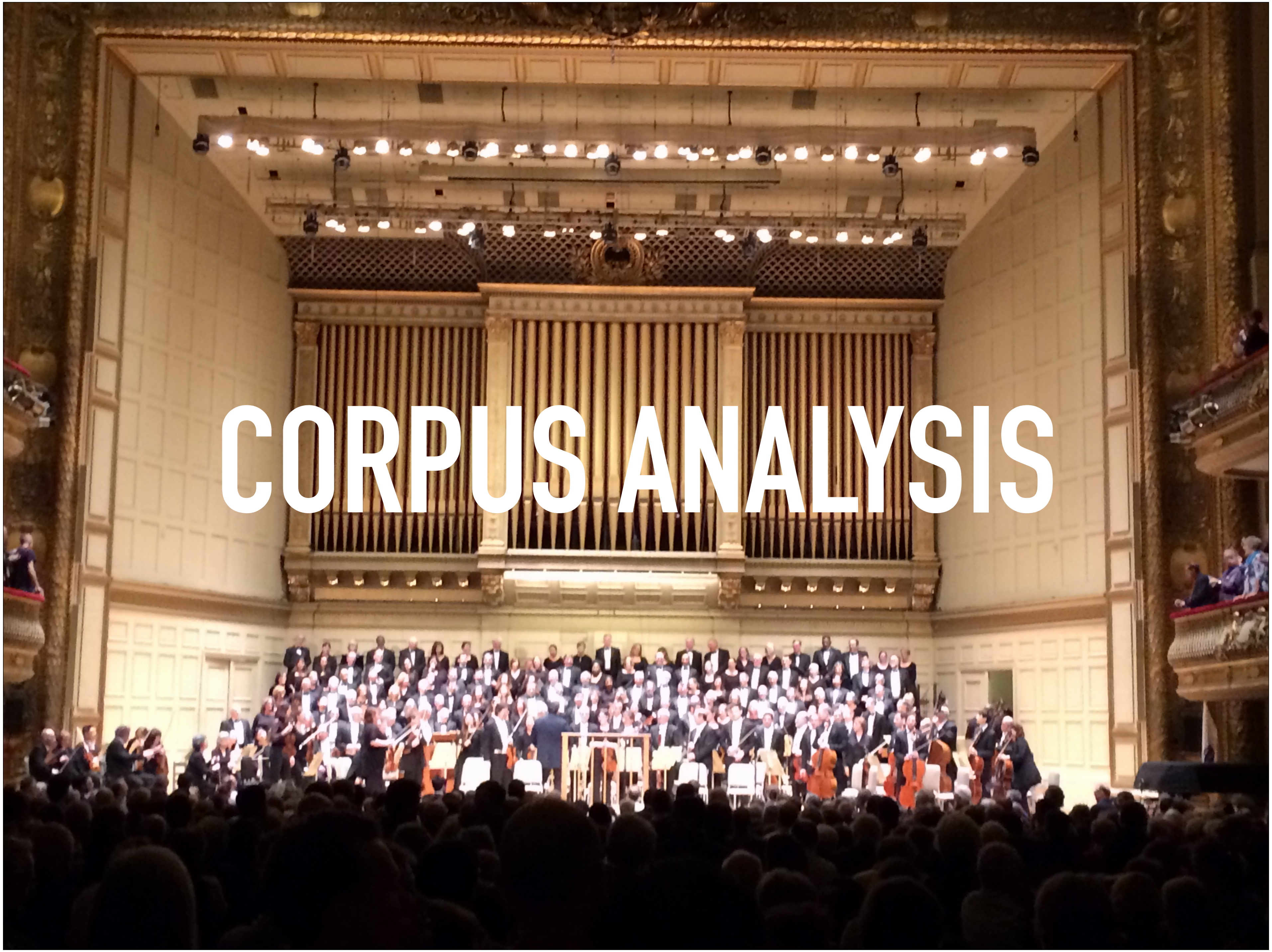
Subtyping is **covariant** for tuples:

- enables sorting of method interfaces according to <:

Dispatching on a single tuple type rather than types of all arguments:

- methods can specify only partially the type of the arguments

```
Union{Tuple{Any,Int}, Tuple{Int,Any}}
```


CORPUS ANALYSIS

METHODOLOGY

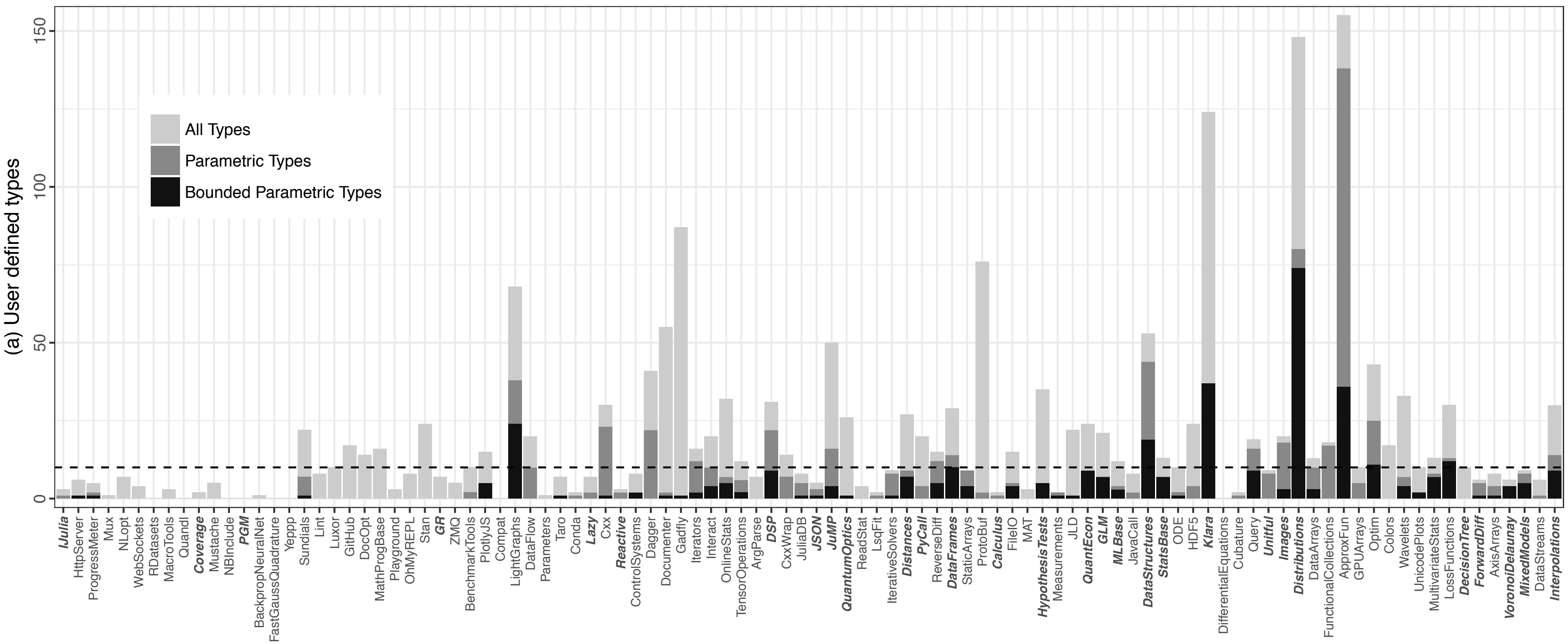
Take the 100 *most starred* Julia packages on GitHub
(compatible with Julia 0.6rc2)

Parse source code of each package, and extract:

- the method signatures
- the declared types

(Learn how to drive R and) do some simple stats…

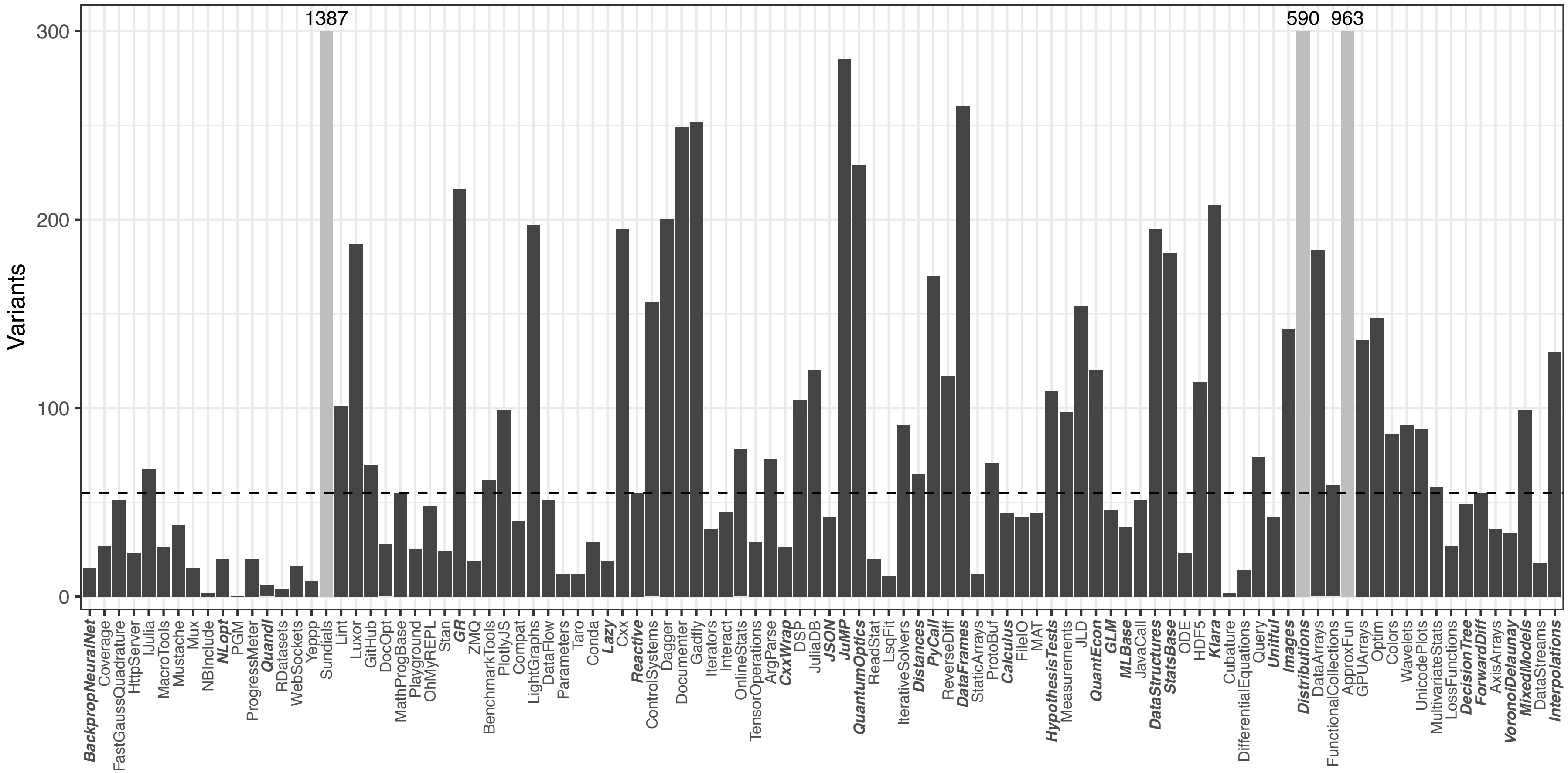
DO PROGRAMMERS DECLARE THEIR OWN TYPES?



Total: 1920 type declarations

of which: 784 parametric, and 369 parametric with non trivial bounds

NO OF METHOD DECLARATIONS

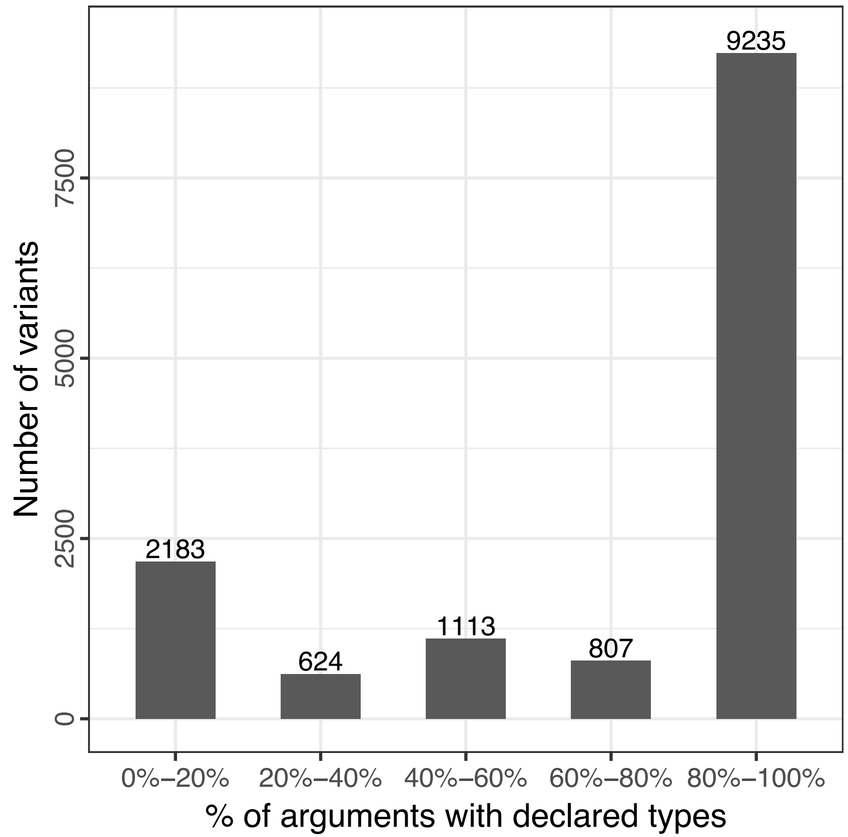


ARE TYPED METHODS COMMON?

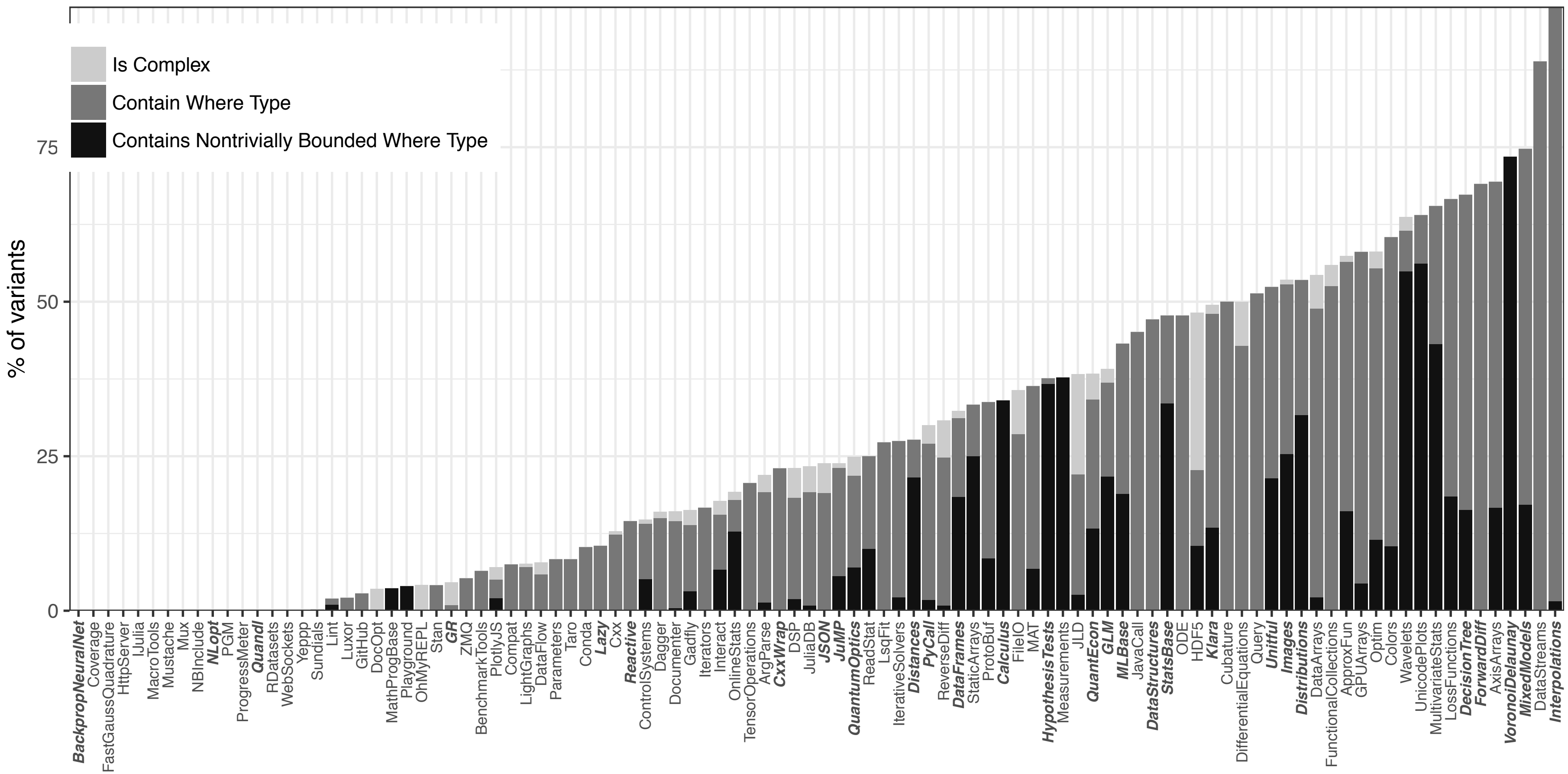
65% of method signatures are fully typed

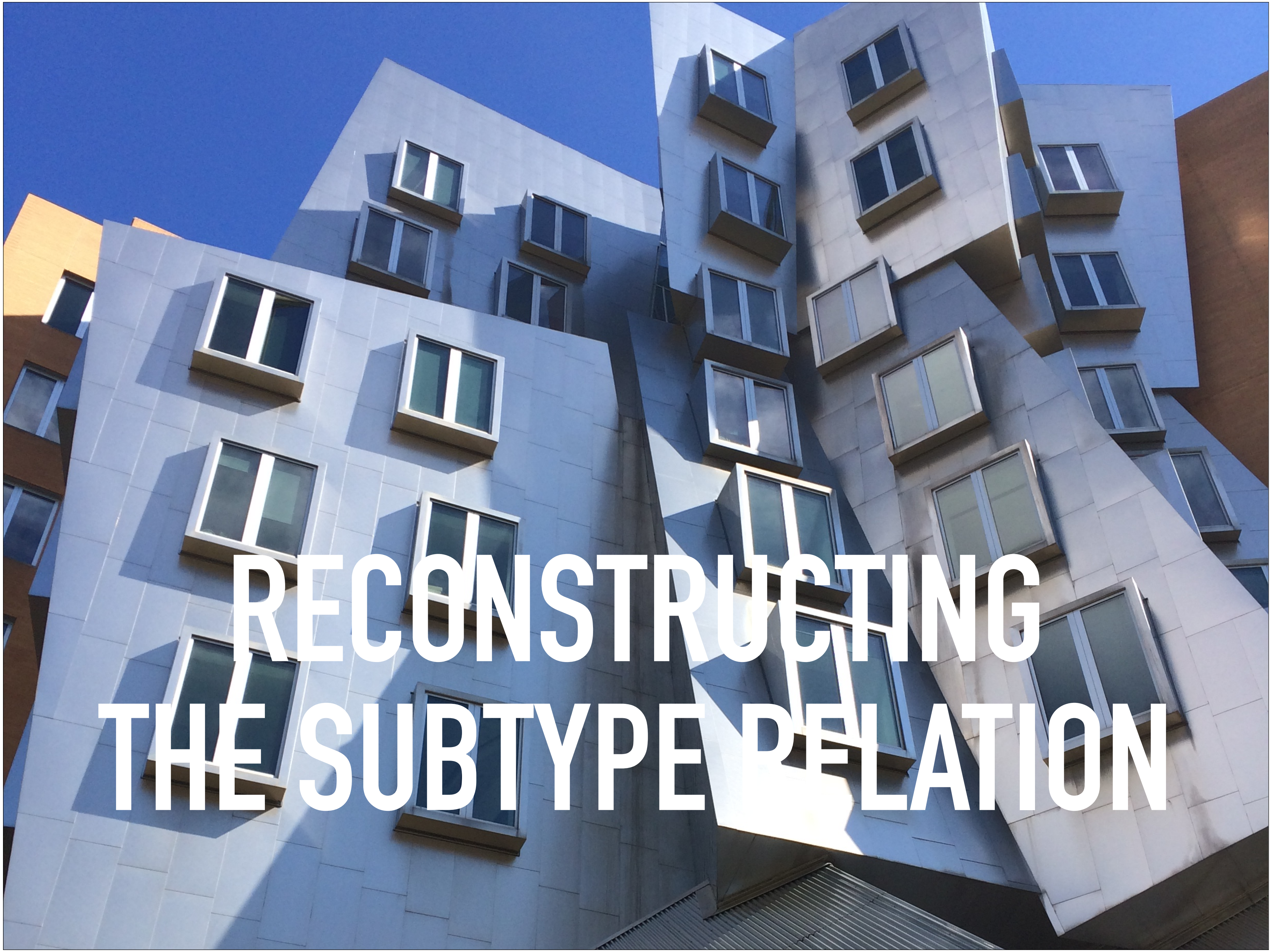
85% have at least some type

15% have no types



HOW COMPLEX ARE METHOD SIGNATURES




RECONSTRUCTING
THE SUBTYPE RELATION

TOOLS

- Interaction with top-level
- Julia regression suite for the `issubtype` function

- Source code: 2200 lines of *(hard to read)* C code
  - *two sources of backtracking: recursion and an explicit stack*
- Read and/or execute step-by-step in GDB

- Take the bike, cross the river, ask Jeff

TYPES

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

We ignore:

- Function types: singleton types, all subtype of `Function`
  supported by our implementation
- `Val`: singleton types, would be easy to add
- `VarArgs`: add a lot of machinery (e.g. to count no of arguments) without introducing new interesting features

STEP 0: IMPLICIT TYPE SIMPLIFICATION

- Simplify trivial `where` constructs:
  - replace `T where t1<:T<:t2` by `t2`
  - replace `t where T` by `t` whenever `T` $\notin$ `fv(t)`

- Remove redundant `Union` types:
  - replace `Union{t}` by `t`
  - given `Union{t1,..,tn}` remove duplicate and obviously redundant types from the list

Julia does more, but these rule are enough to ensure correctness of our `typeof` *formalisation.*

STEP 1: TYPEOF FUNCTION

`typeof(t,G) =` **`DataType`**

- `iskind(t)`
- `t = Any | Tuple{..}`
- `name{`t_1`..`t_n`}` and `name` expects `n` arguments
- `t = (t' where` t_1`<:T<:`t_2`){`t_3`}` and `typeof(t'[`t_3`/T]) = DataType`

`typeof(t,G) =` **`Union`**

- `t = Union{..}`
- `t = (t' where` t_1`<:T<:`t_2`){`t_3`}` and `typeof(t'[`t_3`/T]) = Union`

`typeof(t,G) =` **`UnionAll`**

- `name{`t_1`..`t_n`}` and `name` expects `m > n` arguments
- `t = t' where` t_1`<:T<:`t_2
- `t = (t' where` t_1`<:T<:`t_2`){`t_3`}` and `typeof(t'[`t_3`/T]) = UnionAll`

SUBTYPING: STARTING POINTS

Parametric type application is **invariant**:

`Foo{t₁..tₙ} <: Foo{t'₁..t'ₙ}` iff forall i, t_i `<:` t'_i and t'_i `<:` t_i

Tuples are **covariant**:

`Tuple{t₁..tₙ} <: Tuple{t'₁..t'ₙ}` iff forall i, t_i `<:` t'_i

Subtyping union types, following **types as set of values** idea:

$$\frac{\text{forall } i,\ t_i \texttt{ <: } t'}{\texttt{Union}\{t_1..t_n\} \texttt{ <: } t'} \qquad \frac{\text{exists } i,\ t' \texttt{ <: } t_i}{t' \texttt{ <: } \texttt{Union}\{t_1..t_n\}}$$

The empty union plays the role of the Bottom type

DISTRIBUTIVITY OF TUPLE WRT UNION

```
Tuple{Union{Int, String}, Int}   <:
           Union{Tuple{Int,Int}, Tuple{String,Int}}
```

Cannot be derived from the previous rules.

Only `UnionRight` *applies but neither*

```
Tuple{Union{Int, String}, Int}   <:   Tuple{Int, Int}
Tuple{Union{Int, String}, Int}   <:   Tuple{String, Int}
```

hold .

DISTRIBUTIVITY OF TUPLE WRT UNION

```
Tuple{Union{Int, String}, Int}   <:
               Union{Tuple{Int,Int}, Tuple{String,Int}}
```

Castagna & Frisch: rewrite types in **disjunctive normal form**

Unsound for Julia due to *invariance of type application*:

```
Vector{Union{Int, String}}
Union{Vector{Int},Vector{String}}
```

are *unrelated*.

DISTRIBUTIVITY OF TUPLE WRT UNION

```
Tuple{Union{Int, String}, Int}   <:
          Union{Tuple{Int,Int}, Tuple{String,Int}}
```

Julia implementation relies on **complex** backtracking algorithm

Poor man solution: **rewrite**

$$\texttt{Tuple\{Union\{}t_1..t_n\texttt{\}, t\}}$$

$$\Downarrow$$

$$\texttt{Union\{Tuple\{}t_1\texttt{,t\}..Tuple\{}t_n\texttt{,t\}\}}$$

for **tuples at top-level**

SUBTYPING UNIONALL

UnionAll types obey a forall /exists semantics as well

$$t \text{ where } t_1 <: T <: t_2 \quad <: \quad t'$$

- forall types `t''`, $t_1 <: t'' <: t_2$, it holds that `t[t''/T] <: t'`

$$t' \quad <: \quad t \text{ where } t_1 <: T <: t_2$$

- exists a type `t''`, $t_1 <: t'' <: t_2$, such that `t' <: t[t''/T]`

TYPE VARIABLES AND INVARIANCE

`Foo{Int}  <:  Foo{T} where T`

Invariance requires to check `Int <: T` and `T <: Int`

- in both cases `T` must have **exists** (right) semantics

For each variable, an ***environment*** keeps track

  - the name
  - the left or right semantics (L / R)
  - the lower and upper bounds

SUBTYPING VARIABLES

$$\frac{{}^{L}T^{ub}_{lb} \vdash ub <: t}{{}^{L}T^{ub}_{lb} \vdash T <: t} \qquad \frac{{}^{L}T^{ub}_{lb} \vdash t <: lb}{{}^{L}T^{ub}_{lb} \vdash t <: T}$$

$$\frac{{}^{R}T^{ub}_{lb} \vdash lb <: t}{{}^{R}T^{ub}_{lb} \vdash T <: t} \qquad \frac{{}^{R}T^{ub}_{lb} \vdash t <: ub}{{}^{R}T^{ub}_{lb} \vdash t <: T}$$

TYPE VARIABLES AND INVARIANCE

$$\dfrac{\dfrac{\dfrac{{}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Int} <: \mathtt{Any}}{{}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Int} <: \mathtt{T}} \qquad \dfrac{{}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Bot} <: \mathtt{Int}}{{}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T} <: \mathtt{Int}}}{{}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Foo\{Int\}} <: \mathtt{Foo\{T\}}}}{\vdash \mathtt{Foo\{Int\}} <: \mathtt{Foo\{T\}}\ \mathtt{where}\ \mathtt{T}}$$

TYPE VARIABLES AND INVARIANCE: NOT QUITE RIGHT

$$
\dfrac{\dfrac{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{String} <: \mathtt{Any}}{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{String} <: \mathtt{T}} \qquad \dfrac{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Bot} <: \mathtt{String}}{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T} <: \mathtt{String}}}{}
$$

$$
\dfrac{\dfrac{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Int} <: \mathtt{Any}}{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Int} <: \mathtt{T}} \qquad \dfrac{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Bot} <: \mathtt{Int}}{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T} <: \mathtt{Int}}}{}
$$

$$
\dfrac{\dfrac{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Foo\{Int\}} <: \mathtt{Foo\{T\}} \qquad {}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Foo\{String\}} <: \mathtt{Foo\{T\}}}{{}^{\mathtt{R}}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Tuple\{Foo\{Int\},Foo\{String\}\}} \;<:\; \mathtt{Tuple\{Foo\{T\},Foo\{T\}\}}}}{\vdash \mathtt{Tuple\{Foo\{Int\},Foo\{String\}\}} \;<:\; \mathtt{Tuple\{Foo\{T\},Foo\{T\}\} \ where\ T}}
$$

UPDATING CONSTRAINTS

$$\frac{\texttt{search(E,T)} = {}^{L}T^{ub}_{lb} \qquad E \vdash ub <: t \;;\; E'}{E \vdash T <: t \;;\; E}$$

$$\frac{\texttt{search(E,T)} = {}^{L}T^{ub}_{lb} \qquad E \vdash t <: t \;;\; E'}{E \vdash t <: T \;;\; E}$$

$$\frac{\texttt{search(E,T)} = {}^{R}T^{ub}_{lb} \qquad E \vdash lb <: t \;;\; E'}{E \vdash T <: t \;;\; \texttt{upd}(E, {}^{R}T^{t}_{lb})}$$

$$\frac{\texttt{search(E,T)} = {}^{R}T^{ub}_{lb} \qquad E \vdash t <: ub \;;\; E'}{E \vdash t <: T \;;\; \texttt{upd}(E, {}^{R}T^{t}_{\texttt{Union\{lb,t\}}})}$$

Clever design: no need to compute the intersection of ub and t.

PROPAGATING CONSTRAINTS: TUPLES AND TYPE APPLICATION

$$\frac{E \vdash t_1 <: t'_1 \;;\; E_1 \quad .. \quad E_{n-1} \vdash t_n <: t'_n \;;\; E'}{E \vdash \mathtt{Tuple}\{t_1..t_n\} <: \mathtt{Tuple}\{t'_1..t'_n\} \;;\; E'}$$

$$\frac{E \vdash t <: t' \;;\; E_1 \qquad E_1 \vdash t' <: t \;;\; E'}{E \vdash \mathtt{name}\{t\} <: \mathtt{name}\{t'\} \;;\; E'}$$

PROPAGATING CONSTRAINTS: TUPLES AND TYPE APPLICATION

This solves the previous problem:

$$
\dfrac{
\dfrac{\begin{smallmatrix} R & \mathtt{Any} \\ T & \end{smallmatrix} \vdash \mathtt{Int} <: \mathtt{Any} \;;\; \begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Int} \end{smallmatrix}}
{\begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Bot} \end{smallmatrix} \vdash \mathtt{Int} <: T \;;\; \begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Int} \end{smallmatrix}}
\qquad
\dfrac{\begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Int} \end{smallmatrix} \vdash \mathtt{Bot} <: \mathtt{Int} \;;\; \begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Int} \end{smallmatrix}}
{\begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Int} \end{smallmatrix} \vdash T <: \mathtt{Int} \;;\; \begin{smallmatrix} R & \mathtt{Int} \\ T & \mathtt{Int} \end{smallmatrix}}
}{
\boxed{\begin{smallmatrix} R & \mathtt{Any} \\ T & \mathtt{Bot} \end{smallmatrix}} \vdash \mathtt{Foo\{Int\}} <: \mathtt{Foo\{T\}} \;;\; \boxed{\begin{smallmatrix} R & \mathtt{Int} \\ T & \mathtt{Int} \end{smallmatrix}}
}
$$

…and we cannot derive the second half of the tuple rule:

$$
\dfrac{
\dfrac{\begin{smallmatrix} R & \mathtt{Int} \\ T & \mathtt{Int} \end{smallmatrix} \vdash \mathtt{String} \mathrel{</:} \mathtt{Int}}
{\begin{smallmatrix} R & \mathtt{Int} \\ T & \mathtt{Int} \end{smallmatrix} \vdash \mathtt{String} \mathrel{</:} T}
}{
\boxed{\begin{smallmatrix} R & \mathtt{Int} \\ T & \mathtt{Int} \end{smallmatrix}} \vdash \mathtt{Foo\{String\}} \mathrel{</:} \mathtt{Foo\{T\}}
}
$$

UNSATISFIABLE CONSTRAINTS

$\vdash$ `Tuple{Real,Foo{Int}}` **</:** `Tuple{S,Foo{T}} where S<:T where T`

But we get a derivation with final constraints:

$$\mathtt{R}\ \mathtt{S}^{\mathtt{T}}_{\mathtt{Real}}\quad \mathtt{R}\ \mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}}$$

***Idea**: when a variable is discharged (gets out of scope), check that lb <: ub.*

$$\mathtt{R}\ \mathtt{S}^{\mathtt{T}}_{\mathtt{Real}} \Rightarrow \frac{\mathtt{R}\ \mathtt{S}^{\mathtt{T}}_{\mathtt{Real}}\quad \mathtt{R}\ \mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}} \quad \vdash \mathtt{Real} \not<: \mathtt{Int}}{\mathtt{R}\ \mathtt{S}^{\mathtt{T}}_{\mathtt{Real}}\quad \mathtt{R}\ \mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}} \quad \vdash \mathtt{Real} \not<: \mathtt{T}}$$

KEEPING THE PAST AROUND

```
⊢ Tuple{T,Tuple{S} where S} where T  <:
                          Tuple{Q,Tuple{R}} where R where Q
```

Before discarding S, we have the environment below:

$$\mathtt{L}\ {}_{\mathtt{T}}{}^{\mathtt{Any}}_{\mathtt{Bot}} \qquad \mathtt{R}\ {}_{\mathtt{Q}}{}^{\mathtt{Any}}_{\mathtt{T}} \qquad \mathtt{R}\ {}_{\mathtt{R}}{}^{\mathtt{Any}}_{\mathtt{S}} \qquad \boxed{\mathtt{L}\ {}_{\mathtt{S}}{}^{\mathtt{Any}}_{\mathtt{Bot}}}$$

If we discard S, we cannot validate the consistency of R anymore.

***Solution**: keep discarded variables around — stored ad-hoc in the environment.*

FROM FORALL/EXISTS TO EXISTS/FORALL

`⊢ Vector{Vector{T} where T}  </: Vector{Vector{S}} where S`

But the rules we have until now do derive this judgment.

Semantics of the judgment:

exists one `S` such that **forall** `T`, `Vector{Vector{T}} <: Vector{Vector{S}}`

Tracking L/R is not enough: it misses

the relative order of type application wrt variable introduction.

Idea*: environment is kept sorted wrt order of introduction of variables*

whenever enter an invariant constructor, add a marker to the environment

FROM FORALL/EXISTS TO EXISTS/FORALL

*S is **outside** T: subtyping only allowed if T is constant, e.g. ub <: lb*

$$R_{S}{}^{\mathtt{Any}}_{\mathtt{T}} \;|\; L_{T}{}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{S} \mathrel{</:} \mathtt{T}$$

$$\begin{array}{l}
R_{S}{}^{\mathtt{Any}}_{\mathtt{Bot}} \;|\; L_{T}{}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T} <: \mathtt{S} \;;\; R_{S}{}^{\mathtt{Any}}_{\mathtt{T}} \;|\; L_{T}{}^{\mathtt{Any}}_{\mathtt{Bot}} \\
\hline
R_{S}{}^{\mathtt{Any}}_{\mathtt{Bot}} \;|\; L_{T}{}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Vector\{T\}} \mathrel{</:} \mathtt{Vector\{S\}} \\
\hline
R_{S}{}^{\mathtt{Any}}_{\mathtt{Bot}} \;|\; \vdash \mathtt{Vector\{T\}\ where\ T} \mathrel{</:} \mathtt{Vector\{S\}} \\
\hline
L_{S}{}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Vector\{Vector\{T\}\ where\ T\}} \mathrel{</:} \mathtt{Vector\{Vector\{S\}\}} \\
\hline
\vdash \mathtt{Vector\{Vector\{T\}\ where\ T\}} \mathrel{</:} \mathtt{Vector\{Vector\{S\}\}\ where\ S}
\end{array}$$

FROM FORALL/EXISTS TO EXISTS/FORALL: THE UNION CASE

```
⊢ Vector{Union{Vector{Int},Vector{String}}}  </:
                                 Vector{Vector{S}} where S
```

Intuitively, UnionLeft constraints are discarded:

$$\frac{\text{forall } i,\ E \vdash t_i <: t'}{E \vdash \text{Union}\{t_1..t_n\} <: t'}$$

General rule:

discard only constraints modified for variable
after the most recent barrier

THE DIAGONAL RULE

```
=={T<:Number}(x::T,y::T) = x === y
```

- The interface has type

```
Tuple{T,T} where T<:Number
```

  - equivalent to `Tuple{Number,Number}`
  - matches values as `(3,3.0)`
  - **Incorrect** to use `===` to compare `3` and `3.0`

THE DIAGONAL RULE

```
Tuple{T,T} where T
```

- A variable is in **covariant position** if only Tuple and Union types occur between the occurrence of the variable and its introduction
- **Diagonal rule**: if a variable occurs *more than once in covariant position* and *never in invariant position,* then it must range only over *concrete* types.

In the equality example, T is diagonal and cannot range over Union{Int,Float}.

- *Implementation*:
  - count occurrences in the type derivation
  - check if upper bounds of diagonal vars are concrete *(heuristic)*


VALIDATION

NEVER TRUST RULES ON PAPER

- **Implemented** a subtyping algorithm for Julia types
  - one-to-one mapping of rules to Julia code
  - add a search strategy on top of it

    given the top-level lhs and rhs constructors choose the rule to apply

- About 1k lines of Julia *(code needed disambiguate multiple dispatch)*

- **Passes the subtype regression tests from Julia distribution**
  - *three tests fail: two due the heuristics for the diagonal rule*
    one due to the search for Exist Right

VALIDATE ON REAL CODE

- Modify Julia VM to **log all calls to the subtype function**
  - removing duplicates
- Log importing and running the test suite of ~30 packages

```
Tuple{Type{DataStructures.SAOnlySemiTokensIteration{T} where
T<:Union{DataStructures.SortedDict{K, D, Ord} where Ord<:Base.Order.Ordering where D
where K, DataStructures.SortedMultiDict{K, D, Ord} where Ord<:Base.Order.Ordering where
D where K, DataStructures.SortedSet{K, Ord} where Ord<:Base.Order.Ordering where K,
DataStructures.AbstractExcludeLast{ContainerType} where
ContainerType<:Union{DataStructures.SortedDict{K, D, Ord} where
Ord<:Base.Order.Ordering where D where K, DataStructures.SortedMultiDict{K, D, Ord}
where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedSet{K, Ord} where
Ord<:Base.Order.Ordering where K}, DataStructures.AbstractIncludeLast{ContainerType}
where ContainerType<:Union{DataStructures.SortedDict{K, D, Ord} where
Ord<:Base.Order.Ordering where D where K, DataStructures.SortedMultiDict{K, D, Ord}
where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedSet{K, Ord} where
Ord<:Base.Order.Ordering where K}}}, T} where T<:Union{DataStructures.SortedDict{K, D,
Ord} where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedMultiDict{K,
D, Ord} where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedSet{K,
Ord} where Ord<:Base.Order.Ordering where K,
DataStructures.AbstractExcludeLast{ContainerType} where
ContainerType<:Union{DataStructures.SortedDict{K, D, Ord} where
Ord<:Base.Order.Ordering where D where K, DataStructures.SortedMultiDict{K, D, Ord}
where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedSet{K, Ord} where
Ord<:Base.Order.Ordering where K}, DataStructures.AbstractIncludeLast{ContainerType}
where ContainerType<:Union{DataStructures.SortedDict{K, D, Ord} where
Ord<:Base.Order.Ordering where D where K, DataStructures.SortedMultiDict{K, D, Ord}
where Ord<:Base.Order.Ordering where D where K, DataStructures.SortedSet{K, Ord} where
Ord<:Base.Order.Ordering where K}}
```

— longest type logged while importing JuMP

VALIDATE ON REAL CODE

- **We validate all the logged subtype tests (~1,000,000)**
  - Except ~10 due to diagonal rule heuristic
  - And one mysterious test

THE MYSTERIOUS SUBTYPE TEST

```
Ref(Pair{Pair{T, R}, R} where R) where T    <:
                                    Ref(Pair{A, B} where B) where A
```

- Julia says **true**, we say **false**
- After a long investigation, consensus that **false** is correct
- Jeff patched Julia 0.7-dev 15 days ago: *bounds cannot refer to out-of-scope variables anymore, unless the bound is just the variable itself.*
- *Unsatisfying strategy, work in progress…*


MY THOUGHTS ON
JULIA SUBTYPING

AN ORIGINAL AND CLEVER POINT IN THE DESIGN SPACE

- Compare with the Fortress experience *(Steele talk at JuliaCon'16)*
  - *Fortress supported multiple dispatch and a rich type system*
  - *Fortress failed due to difficulty of defining a* ***sound*** *semantics*
  - Unsoundness simplifies the design

- Julia provides a **gradual typing** system
  - users encouraged to write types
  - Compiler does not trust type annotation, but…
  - …programming style enables aggressive code specialisation


BIKES
NOT BOMBS
Cyclists Union