JULIA SUBTYPING RECONSTRUCTED

Francesco Zappa Nardelli
Artem Pelenitsyn, Benjamin Chung, Jan Vitek
Jeff Bezanson

Inria
Northeastern U.
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
Figure: benchmark times relative to C (smaller is better, C performance = 1.0).
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.SignedMultiplicativeInverse{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.SignedMultiplicativeInverse{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
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)

myadd(3.0, 4.0)      Float64, Float64
> 0
myadd(3, 4.0)         Int64, Float64
> 2

Tuple{Float64, Float64}

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

Tuple{Number, Number}
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: Subtyping in Julia 0.6:
   ➤ Specified by 2400 lines of highly optimised C code
   ➤ Inspired by semantic subtyping, but original design in the end

0.2: Immutables

0.3: UnionAll types, variable lower bounds, complete overhaul of subtyping

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

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

Subtyping in Julia 0.6:
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)*

```plaintext
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
User defined types can be parametrised by other types
—and by values of primitive types (e.g. Int or Bool)

```plaintext
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 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}

≠

Vector{Point{T}} where T
PARAMETRIC TYPES ARE INVARIANT

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

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

![Bar chart showing the number of method declarations for various packages](chart.png)
ARE TYPED METHODS COMMON?

65% of method signatures are fully typed
85% have at least some type
15% have no types
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\{t_1..t_n\} | Tuple\{t_1..t_n\}
   | t\{t_1..t_n\} | t where t_1<:T<:t_n | 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 $t_1 <: T <: t_2$ by $t_2$
  ➤ replace $t$ where $T$ by $t$ whenever $T \notin \text{fv}(t)$

➤ Remove redundant Union types:
  ➤ replace $\text{Union}\{t\}$ by $t$
  ➤ given $\text{Union}\{t_1, \ldots, t_n\}$ 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{t₁..tₙ} and name expects n arguments
- t = (t’ where t₁<:T<:t₂){t₃} and typeof(t’[t₃/T]) = DataType

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

- t = Union{..}
- t = (t’ where t₁<:T<:t₂){t₃} and typeof(t’[t₃/T]) = Union

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

- name{t₁..tₙ} and name expects m > n arguments
- t = t’ where t₁<:T<:t₂
- t = (t’ where t₁<:T<:t₂){t₃} and typeof(t’[t₃/T]) = UnionAll
**SUBTYPING: STARTING POINTS**

Parametric type application is **invariant**:

\[
\text{Foo}\{t_1..t_n\} <: \text{Foo}\{t'_1..t'_n\} \text{ iff } \forall i, t_i <: t'_i \text{ and } t'_i <: t_i
\]

Tuples are **covariant**:

\[
\text{Tuple}\{t_1..t_n\} <: \text{Tuple}\{t'_1..t'_n\} \text{ iff } \forall i, t_i <: t'_i
\]

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

\[
\begin{align*}
\forall i, t_i &: t' \\
\Rightarrow & \\
\text{Union}\{t_1..t_n\} &: t' \\
\end{align*}
\]

\[
\begin{align*}
\exists i, t' &: t_i \\
\Rightarrow & \\
t' &: \text{Union}\{t_1..t_n\}
\end{align*}
\]

*The empty union plays the role of the Bottom type*
DISTRIBUTIVITY OF TUPLE WRT UNION

\[
\text{Tuple\{Union\{Int, String\}, Int\} } \prec:
\]

\[
\text{Union\{Tuple\{Int,Int\}, Tuple\{String,Int\}\}}
\]

Cannot be derived from the previous rules.

*Only* UnionRight applies but neither

\[
\text{Tuple\{Union\{Int, String\}, Int\} } \prec: \text{ Tuple\{Int, Int\}}
\]

\[
\text{Tuple\{Union\{Int, String\}, Int\} } \prec: \text{ Tuple\{String, Int\}}
\]

hold.
DISTRIBUTIVITY OF TUPLE WRT UNION


tuple

\[ \text{Tuple\{Union\{Int, String\}, Int\} <: Union\{Tuple\{Int,Int\}, Tuple\{String,Int\}\}} \]

Castagna & Frisch: rewrite types in \textit{disjunctive normal form}

Unsound for Julia due to \textit{invariance of type application}:

\[ \text{Vector\{Union\{Int, String\}\}} \]
\[ \text{Union\{Vector\{Int\}, Vector\{String\}\}} \]

are \textit{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

Tuple{Union{t₁..tₙ}, t}

Union{Tuple{t₁,t}..Tuple{tₙ,t}}

for tuples at top-level
UnionAll types obey a forall /exists semantics as well

\[ t \text{ where } t_1 <: T <: t_2 <: t' \]

- forall types \( t'' \), \( t_1 <: t'' <: t_2 \), it holds that \( t[t''/T] <: t' \)

\[ t' <: t \text{ where } t_1 <: T <: t_2 \]

- 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

\[ L \quad ub \quad \vdash \quad ub \quad <: \quad t \]
\[ L \quad T_{lb} \quad \vdash \quad T \quad <: \quad t \]
\[ R \quad ub \quad \vdash \quad lb \quad <: \quad t \]
\[ R \quad T_{lb} \quad \vdash \quad T \quad <: \quad t \]
\[ L \quad ub \quad \vdash \quad t \quad <: \quad lb \]
\[ L \quad T_{lb} \quad \vdash \quad t \quad <: \quad T \]
\[ R \quad ub \quad \vdash \quad t \quad <: \quad ub \]
\[ R \quad T_{lb} \quad \vdash \quad t \quad <: \quad T \]
TYPE VARIABLES AND INVARIANCE

\[
\begin{align*}
\text{R} & \quad \text{Any} \\
\text{T} & \quad \bot \quad \mathsf{Int} \quad \vdash \quad \mathsf{Int} \quad \mathsf{<:} \quad \text{Any} \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \bot \quad \mathsf{Int} \quad \vdash \quad \mathsf{Int} \quad \mathsf{<:} \quad \\ & \quad \quad \text{T} \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \bot \quad \mathsf{Bot} \quad \vdash \quad \mathsf{Bot} \quad \mathsf{<:} \quad \mathsf{Int} \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \\ & \quad \vdash \quad \mathsf{Bot} \quad \mathsf{<:} \quad \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \\ & \quad \vdash \quad \mathsf{Int} \quad \mathsf{<:} \quad \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \\ & \quad \vdash \quad \mathsf{Bot} \quad \mathsf{<:} \quad \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \\ & \quad \vdash \quad \mathsf{Int} \quad \mathsf{<:} \quad \\
\text{R} & \quad \text{Any} \\
\text{T} & \quad \\ & \quad \vdash \quad 
\end{align*}
\]

\[\vdash \quad \mathsf{Foo}\{\mathsf{Int}\} \quad \mathsf{<:} \quad \mathsf{Foo}\{\mathsf{T}\} \quad \text{where} \quad \mathsf{T}\]
TYPE VARIABLES AND INVARIANCE: NOT QUITE RIGHT

\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{String} <: \text{Any} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Bot} <: \text{String} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{String} <: \text{T} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{T} <: \text{String} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Int} <: \text{Any} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Bot} <: \text{Int} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Int} <: \text{T} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{T} <: \text{Int} } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Foo}[\text{Int}] <: \text{Foo}[\text{T}] } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Foo}[\text{String}] <: \text{Foo}[\text{T}] } \]
\[ \frac{ \text{Any} } { \text{T}_{\text{Bot}} \vdash \text{Tuple}[\text{Foo}[\text{Int}],\text{Foo}[\text{String}]] <: \text{Tuple}[\text{Foo}[\text{T}],\text{Foo}[\text{T}]] } \]

\[ \vdash \text{Tuple}[\text{Foo}[\text{Int}],\text{Foo}[\text{String}]] <: \text{Tuple}[\text{Foo}[\text{T}],\text{Foo}[\text{T}]] \text{ where } \text{T} \]
UPDATING CONSTRAINTS

\[
\text{search}(E, T) = T_{1b}^L \\
E \vdash \text{ub} <: t ; E' \\
\hline \\
E \vdash T <: t ; E \\
\]

\[
\text{search}(E, T) = T_{1b}^L \\
E \vdash t <: t ; E' \\
\hline \\
E \vdash t <: T ; E \\
\]

\[
\text{search}(E, T) = T_{1b}^R \\
E \vdash \text{lb} <: t ; E' \\
\hline \\
E \vdash T <: t ; \text{upd}(E, T_{1b}) \\
\]

\[
\text{search}(E, T) = T_{1b}^R \\
E \vdash t <: \text{ub} ; E' \\
\hline \\
E \vdash t <: T ; \text{upd}(E, T_{\text{Union}\{\text{lb}, t\}}) \\
\]

Clever design: no need to compute the intersection of ub and t.
PROPAGATING CONSTRAINTS: TUPLES AND TYPE APPLICATION

\[ \begin{align*}
E & \vdash t_1 <: t'_1 ; E_1 \quad \ldots \quad E_{n-1} \vdash t_n <: t'_n ; E' \\
E & \vdash \text{Tuple}\{t_1..t_n\} <: \text{Tuple}\{t'_1..t'_n\} ; E'
\end{align*} \]

\[ \begin{align*}
E & \vdash t <: t' ; E_1 \quad E_1 \vdash t' <: t ; E' \\
E & \vdash \text{name}\{t\} <: \text{name}\{t'\} ; E'
\end{align*} \]
This solves the previous problem:

\[
\begin{align*}
R & \quad \text{Any} \\
T & \quad \vdash \text{Int} <: \text{Any} ; \ T \ 	ext{Int} \\
\text{Any} & \quad \vdash \text{Int} <: \ T ; \ T \ 	ext{Int} \\
T \ 	ext{Bot} & \quad \vdash \text{Bot} <: \ T ; \ T \ 	ext{Int} \\
\text{Any} & \quad \vdash \text{Bot} <: \ T ; \ T \ 	ext{Int} \\
\text{Int} & \quad \vdash \text{String} <: \ T \ 	ext{Int} \\
\text{Int} & \quad \vdash \text{String} <: \ T \\
\text{Int} & \quad \vdash \text{Foo} \{\text{String}\} <: \text{Foo} \{T\} ; \ T \ 	ext{Int}
\end{align*}
\]

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

\[
\begin{align*}
R & \quad \text{Int} \\
T \ 	ext{Int} & \quad \vdash \text{String} :/\: \text{Int} \\
R & \quad \text{Int} \\
T \ 	ext{Int} & \quad \vdash \text{String} :/\: \ T \\
R & \quad \text{Int} \\
T \ 	ext{Int} & \quad \vdash \text{Foo} \{\text{String}\} :/\: \text{Foo} \{T\}
\end{align*}
\]
UNSATISFIABLE CONSTRAINTS

⊢ Tuple{Real,Foo{Int}} ↜: Tuple{S,Foo{T}} where S <: T where T

But we get a derivation with final constraints:

\[
\begin{array}{cccc}
R & S & T & R \\
S & \text{Real} & R & T & \text{Int} \\
R & \text{Real} & R & T & \text{Int} \\
\end{array}
\]

Idea: when a variable is discharged (gets out of scope), check that \(lb < ub\).
KEEPING THE PAST AROUND

\[
\Gamma \vdash \text{Tuple}\{T, \text{Tuple}\{S\} \text{ where } S \text{ where } T \prec \text{Tuple}\{Q, \text{Tuple}\{R\}\} \text{ where } R \text{ where } Q}
\]

Before discarding \(S\), we have the environment below:

\[
\begin{array}{cccc}
L & \text{Any} & R & \text{Any} \\
\text{Bot} & Q & \text{Bot} & R \\
\end{array}
\]

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

\[ \vdash \text{Vector}\{\text{Vector}\{T\}\text{ where } T\} \少于: \text{Vector}\{\text{Vector}\{S\}\text{ where } S} \]

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\}\} \少于: 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
S is outside T: subtyping only allowed if T is constant, e.g. $ub <: lb$
FROM FORALL/EXISTS TO EXISTS/FORALL: THE UNION CASE

\[ \Gamma \vdash \text{Vector}\{\text{Union}\{\text{Vector}\{\text{Int}\}, \text{Vector}\{\text{String}\}\}\}\} <: \text{Vector}\{\text{Vector}\{S\}\} \text{ where } S \]

Intuitively, UnionLeft constraints are discarded:

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

*General rule:*

discard only constraints modified for variable

after the most recent barrier
THE DIAGONAL RULE

\[ \text{equiv}_{\text{:Number}}(x::T,y::T) = x === y \]

- The interface has type

  \[ \text{Tuple\{T,T\} where T<:Number} \]

  - equivalent to \( \text{Tuple\{Number,Number\}} \)
  - 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
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