# The Beauty and the Beast

An Exercise in Comparative Computational Linguistics

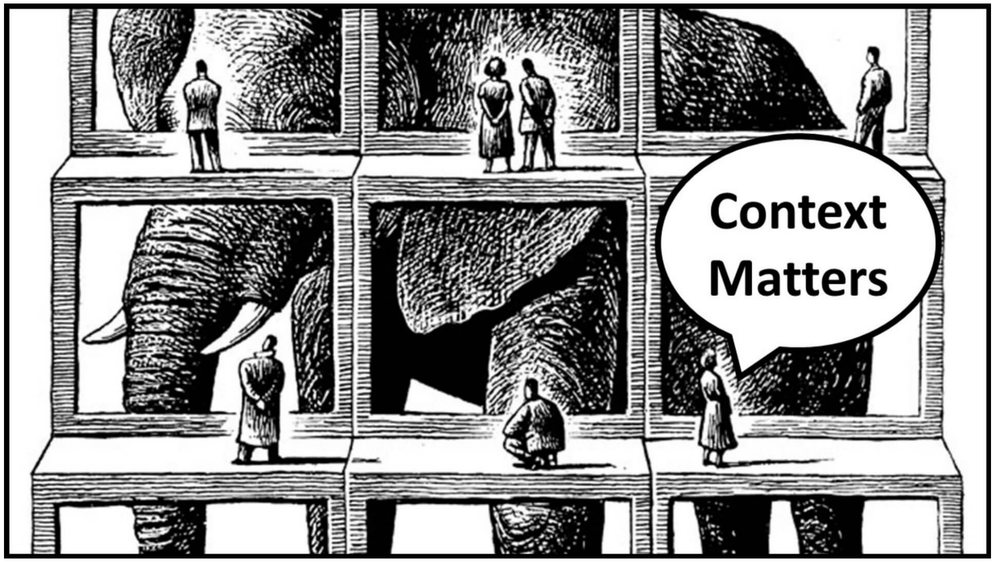
*collaboration with F. Zappa Nardelli, A. Pelenitsyn J. Belyakova, B. Chung*

**Speaker's note**: I have great affection for both languages. Neither title/contents should be construed negatively.

My goal is to make points about language design and, perhaps, meta-points about how we approach it.

**Warning**: You are going see code written in Fortress, Cecil, Julia, R, and C. And some reduction rules if we get around to it. And a demo.

**Warning:** The research part is work in progress…




DARPA HPCS was to be the Manhattan project of parallel computing. Initially modest funding (~$12m for Cray, IBM, Sun, SGI); ballooning to $500m for IBM and Cray.  HPCS started in ~02 and ran until ~10.  Sw. outcomes were Chapel from Cray, X10 from IBM and Fortress. X10 was an ambitious but stuck to the Java/C++ mold for adoption.

Fortress was a wildly ambitious. One can't do the language justice in the time, read the papers.

**Speaker's note**: neither associated with Fortress nor the Julia team. Some here is speculation and post hoc reconstruction.  Shortly part of the IBM's X10, very shortly.


Guy Steele's slogan.

One of the aim of Fortress was to be an extensible rather than building all possible bells and whistles in it, we should be able to write them on top of Fortressas modules that can be combined freely.

For this to be achievable Fortress needed to allow for software composition in more flexible and expressive ways than had been possible before. In particular the type system should not get in the way of composition — it should allows as much safe composition as possible while warning against ambiguities.

# Fortress

Object, Multiple inheritance & Traits

Modules and separate compilation

Symmetric multi methods

Polymorphic type inference

Contracts and tests

Parallelism

Mathematical notation

Garbage collected

Units and dimensions

Performance

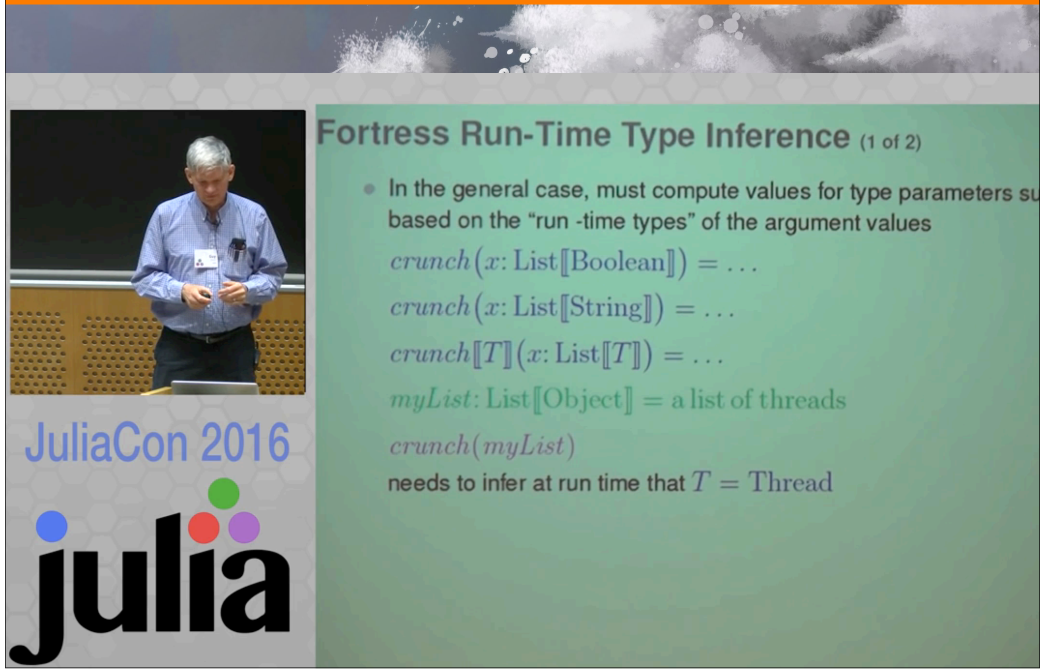
Fortress rethinks language design from the ground up. The team with some of the smartest language designers — several of them from NEU.

Fortress innovated on many of the fronts listed here. The ones that we are most interested in are the interplay between multi methods, type inference and separate compilation.

```
conjGrad⟦Elt extends Number, nat N,
         Mat extends Matrix⟦Elt,N×N⟧,
         Vec extends Vector⟦Elt,N⟧
        ⟧(A: Mat, x: Vec): (Vec, Elt) = do
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  ρ: Elt = r^T r
  for j ← seq(1:cgit_max) do
     q = A p
     α = ρ / p^T q
     z := z + α p
     r := r - α q
     ρ0 = ρ
     ρ := r^T r
     β = ρ / ρ0
     p := r + β p
  end
  (z, ‖x - A z‖)
end
```

**Notice 1)** Code uses Unicode character set, typesetting can match a mathematical paper. Feels like writing LaTeX, for good and ill.
**Notice 2)** Local variable types are inferred. As operations are overloaded, the compiler must statically determine the type signature of the applicable function at any call, infer the return type
**Notice 3**) Expressive parameterization over types and values (nat N capture matrix dimensions) for arguments and return types
**Notice 4)** Combining type checking, multiple dispatch, separate compilation, multiple inheritance, how hard can that be?

# Fortress




Too hard. That is where Fortress got stuck. Guy Steele recounts in his 2016 Keynote at JuliaCon. See: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjOzLb2vrXXAhWI8YMKHdyFCKIQtwIIKDAA&url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3DEZD3Scuv02g&usg=AOvVaw1GEHgOO_YqcwngJYlN8UfD

Getting any complex type system right is an extremely complex endeavor. The designers of Rust spent years on their type system and, arguably, there may still be a few quirks.

Fortress was trying to push boundaries and solve open problems while designing a commercial language. (Contrast: the myth of the design-in-a-week language such as JavaScript)

# Cecil

```
object shape;
  method draw(s) { … }

object circle isa shape;
  method draw(c@circle) { … }

object rectangle isa shape;
    method draw(r@rectangle) { … }

object odd isa rectangle, circle;

circle.draw(circle)
circle.draw(rectangle)
odd.draw(odd)
```

Chambers **Object-oriented multi-methods in Cecil**. ECOOP92

Some background…

Multi-methods go way back. They appeared in variants of LISP, such as CLOS, that Guy Steele worked on in the 1980s. They were used in experimental languages such as Craig Chambers' Cecil.

This example shows three calls to draw. One dispatches to circle. Another to shape. And the last one is ambiguous. How to disambiguate it? Especially in the presence of separate compilation and if the overloading method could be introduced after object odd is created?

# Fortress

$$f(a\colon \mathrm{Object}, b\colon \mathbb{Z})\colon \mathbb{Z} = 1$$
$$f(a\colon \mathbb{Z}, b\colon \mathrm{Object})\colon \mathbb{Z} = 2$$

`f(Z,Z)?`

Allen, Hilburn, Kilpatrick, Luchangco, Ryu, Chase, Steele: **Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance.** OOPSLA11

Fortress supported symmetric multiple dispatch. Symmetry here means that the receiver of the method is treated as any argument.

Examples from the paper.

In the above, the two methods definitions are such that it is possible to write a call that is ambiguous. Fortress should rule this out.

# Fortress

$$f(a\colon \text{Object}, b\colon \mathbb{Z})\colon \mathbb{Z} = 1$$
$$f(a\colon \mathbb{Z}, b\colon \text{Object})\colon \mathbb{Z} = 2$$
$$f(a\colon \mathbb{Z}, b\colon \mathbb{Z})\colon \mathbb{Z} = 3$$

```
f(Z,Z)?
```

Allen, Hilburn, Kilpatrick, Luchangco, Ryu, Chase, Steele: **Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance.** OOPSLA11

The solution here is to enforce the Meet Rule (from Guiseppe Castagna's work) where you define a method that is the greatest lower bound of the tuple of argument types that have a non-empty intersection and where they are not directly related by subtyping.

# Fortress

**No Duplicates Rule** For every $d_1, d_2 \in \mathcal{D}_f$, if $d_1 \preceq d_2$ and $d_2 \preceq d_1$ then $d_1 = d_2$.

**Meet Rule** For every $d_1, d_2 \in \mathcal{D}_f$, there exists a declaration $d_0 \in \mathcal{D}_f$ (possibly $d_1$ or $d_2$) such that, $d_0 \preceq d_1$ and $d_0 \preceq d_2$ and $d_0$ is applicable to any type $T \in \mathcal{T}$ to which both $d_1$ and $d_2$ are applicable.

**Return Type Rule** For every $d_1, d_2 \in \mathcal{D}_f$ with $d_1 \preceq d_2$, and every type $T \not\equiv \text{Bottom}$ such that $d_1 \in \mathcal{D}_f(T)$, if an instance $f\ S_2 : T_2$ of $d_2$ is applicable to $T$, then there is an instance $f\ S_1 : T_1$ of $d_1$ that is applicable to $T$ with $T_1 <: T_2$.

Allen, Hilburn, Kilpatrick, Luchangco, Ryu, Chase, Steele: **Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance.** OOPSLA11

The key definition in the paper: what does it mean for a set of methods to be sound? They have to abide by the following three rules. The more specific relation implies subtyping. Checking this can be quite expensive as alluded to by Guy Steele in his JuliaCon talk.

# Fortress

Failure I: Type soundness still elusive

Failure II: Performance goals unmet

Failure III: Empirical validation not complete

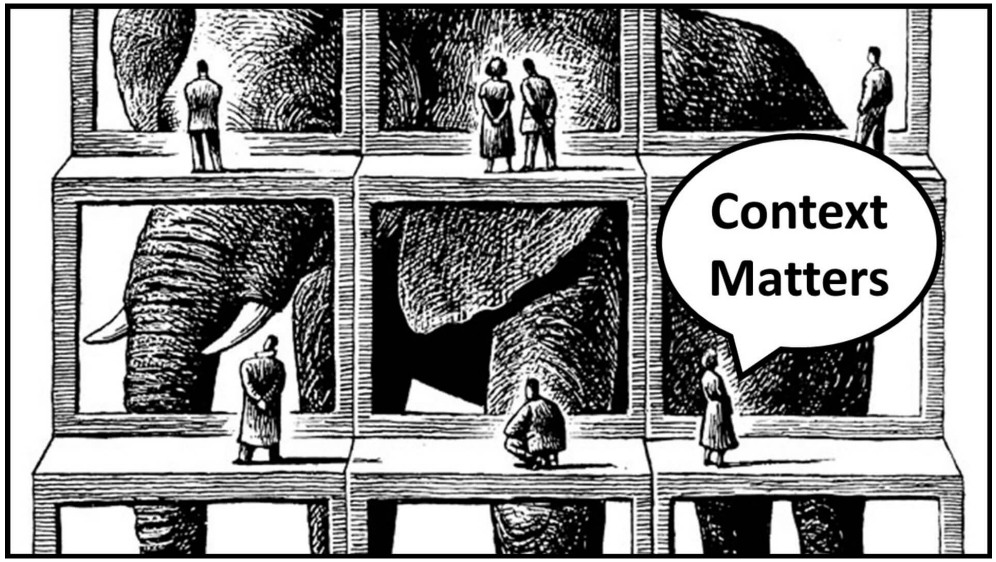
*Qui embrasse trop, mal étreint*

The three failures of Fortress were that (1) the goal of a sound type system for separate compilation of multi methods was never reached, (2) the implementation ran on top of a JVM but was never performant. [Aparte: my one contribution to X10 was to argue to Vivek Sarkar in favor of a subset of C++ that aligned with Java and build the first implementation of X10 on top of a JVM, but I digress.] Performance was just something the team did not get to. Due to the time taken by (1).

Finally, and most important, the type system was designed based on a vision of what would be useful and what could be adopted by users- Yet, since the language was not at a point where it could deliver the kind of performance that users would find acceptable, Fortress could not be empirically evaluated. There was never feedback from users into the design.

julia

Now let’s turn our attention to Julia.  A language designed to appeal to the same general audience.  A language built around the concept of multiple dispatch. But with much less ambitious goals.  It has a vanilla syntax, no parallelism, no objects, no type system.




My interest in Julia is accidental and geographic.  I have been working on R for a few years. When I moved to Boston in 2014, I gave a talk at an R meetup and met the Julia team afterwards. Conversations led to a collaboration.

I promised to help with writing an overview paper on Julia. Our first step was to try to write down the couple of rules needed to define the subtype relation. And six month later…

# julia is…

…a dynamic language for high-performance scientific computing

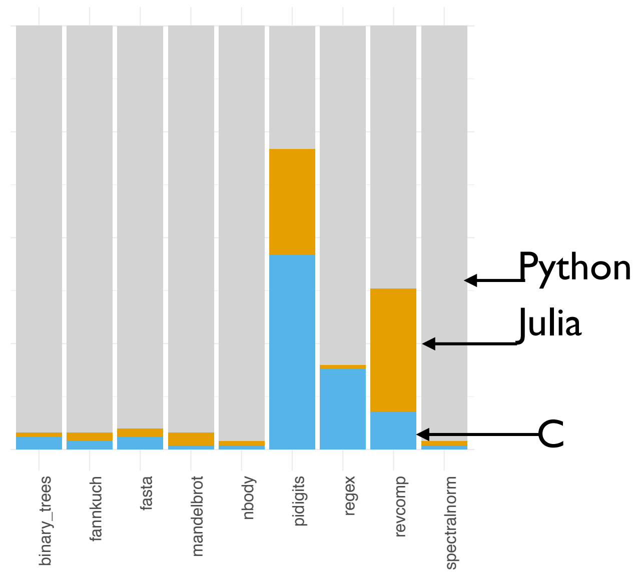
…open source since its inception by Jeff Bezanson circa 2012

| | R | julia |
|---|---|---|
| **Dynamic** | **yes** | **yes** |
| **Vectorized** | **yes** | **yes** |
| **Memory management** | **automatic** | **automatic** |
| **Implementation** | **interpreted** | **native** |
| **Type declarations** | **—** | **user-defined generic types** |
| **Meta-programming** | `substitute()` | **macros** |
| **Parameter passing** | **by promise** | **by value** |

Julia is a dynamic OO language for scientific computing; available since 2012; A quickly growing ecosystem with 6000+ packages, open source; Like R it is dynamic, it can operate on entire vectors and matrices, and is memory safe with a garbage collector

# julia is…

…surprisingly fast




These are numbers obtained by Ben Chung on the Language Benchmark Game. These are small programs that are not necessarily representative of the targets of various languages, but they do give a first approximation of language implementations. The Julia code was not overly optimized. The results are normalized to the performance of Python (dynamic and interpreted). These are the sequential benchmarks. Typically Julia runs <2x of C. The Julia compiler represents a relatively small implementation effort compared to, e.g., Java or JavaScript. How is this possible?

# julia is…

Focus on **polymorphism** and **performance**

Use R's `colMeans` function as running example

Show same level of *dynamism* and *polymorphism* **without** sacrificing performance

`https://github.com/janvitek/can_R_learn_from_Julia.git`

While Julia does not target statistics, I will demonstrate that it is extensible in ways that permit it to solve the same problems as R; To this end I will take an example, the colMeans function, which I will quickly review  and then proceed to described two implementations, one in R and one in C for efficiency. Then I will show you how to replicate colMeans in Julia with less code and without having to switch languages; These practical example will highlight some of the key features of Julia. All code presented today is on Github colMeans

```
x = 1:100
dim(x) = c(50,2)
colMeans(x)
[1] 25000.5 75000.5

x = complex(r=1:60,i=1:60)
dim(x) = c(10,3,2)
colMeans(x)
           [,1]        [,2]
[1,]  5.5+ 5.5i 35.5+35.5i
[2,] 15.5+15.5i 45.5+45.5i
[3,] 25.5+25.5i 55.5+55.5i
```

The strengths of R lie in how easy it is for end-users to specify complex operations with small vocabulary of reusable abstractions. colMeans is a good example, this function works just as well on two dimensional vectors of integer or three dimensional vectors of complex values. **polymorphism** helps end users, as they don't need to learn many different functions, and library developers who can write a single version of any code regardless of the types being processed. We now turn to how this is achieved in the library.

# colMeans

```
colMeans = function(x, na=FALSE, dims=1L) {
 dn   = dim(x)
 id   = 1:dims
 n    = prod(dn[id])
 dn   = dn[-id]
 pdn = prod(dn)
 z    = if (is.complex(x))
  .Internal(colMeans(Re(x),n,pdn,na))+(0+1i)*
  .Internal(colMeans(Im(x),n,pdn,na))
  else .Internal(colMeans(x,n,pdn,na))
 z
}
```

The default implementation of colMeans has a part that is truly polymorphic, where we compute how to traverse the vector and which dimensions to aggregate; Already in the R code, a part that dispatch on the type of the vector. Complex or not. In either case, bottoming out in an .Internal call.

# colMeans

Most of the behavior implemented in C in do_colsum()

```c
SEXP attribute_hidden do_colsum(SEXP call, SEXP op, SEXP args, SEXP rho) {
    SEXP x, ans = R_NilValue;
    int type;
    Rboolean NaRm, keepNA;

    checkArity(op, args);
    x = CAR(args); args = CDR(args);
    R_xlen_t n = asVecSize(CAR(args)); args = CDR(args);
    R_xlen_t p = asVecSize(CAR(args)); args = CDR(args);
    NaRm = asLogical(CAR(args));
    if (n == NA_INTEGER || n < 0)
        error(_("invalid '%s' argument"), "n");
    if (p == NA_INTEGER || p < 0)
        error(_("invalid '%s' argument"), "p");
    if (NaRm == NA_LOGICAL) error(_("invalid '%s' argument"), "na.rm");
    keepNA = !NaRm;

    int OP = PRIMVAL(op);
    switch (type = TYPEOF(x)) {
    case LGLSXP: break;
    case INTSXP: break;
    case REALSXP: break;
    default:
        error(_("'x' must be numeric"));
    }

    if (OP == 0 || OP == 1) { /* columns */
        PROTECT(ans = allocVector(REALSXP, p));
        for (R_xlen_t j = 0; j < p; j++) {
            R_xlen_t  cnt = n, i;
            LDOUBLE sum = 0.0;
            switch (type) {
            case REALSXP:   {
                double *rx = REAL(x) + (R_xlen_t)n*j;
                if (keepNA)
                    for (sum = 0., i = 0; i < n; i++) sum += *rx++;
                else
                    for (cnt = 0, sum = 0., i = 0; i < n; i++, rx++)
                        if (!ISNAN(*rx)) {cnt++; sum += *rx;}
                break;
            }
            case INTSXP: {
                int *ix = INTEGER(x) + (R_xlen_t)n*j;
                for (cnt = 0, sum = 0., i = 0; i < n; i++, ix++)
                    if (*ix != NA_INTEGER) {cnt++; sum += *ix;}
                    else if (keepNA) {sum = NA_REAL; break;}
                break;
            }
            case LGLSXP: {
                int *ix = LOGICAL(x) + (R_xlen_t)n*j;
                for (cnt = 0, sum = 0., i = 0; i < n; i++, ix++)
                    if (*ix != NA_LOGICAL) {cnt++; sum += *ix;}
                    else if (keepNA) {sum = NA_REAL; break;}
            break;
            }
            }
            if (OP == 1) sum /= cnt; /* gives NaN for cnt = 0 */
            REAL(ans)[j] = (double) sum;
        }
    } else { /* rows */
        PROTECT(ans = allocVector(REALSXP, n));

        /* allocate scratch storage to allow accumulating by columns
           to improve cache hits */
        int *Cnt = NULL;
        LDOUBLE *rans;
        if(n <= 10000) {
            R_CheckStack2(n * sizeof(LDOUBLE));
            rans = (LDOUBLE *) alloca(n * sizeof(LDOUBLE));
            Memzero(rans, n);
        } else rans = Calloc(n, LDOUBLE);
        if (!keepNA && OP == 3) Cnt = Calloc(n, int);

        for (R_xlen_t j = 0; j < p; j++) {
            LDOUBLE *ra = rans;
            switch (type) {
            case REALSXP:
            {
                double *rx = REAL(x) + (R_xlen_t)n *
                if (keepNA)
                    for (R_xlen_t i = 0; i < n; i++) *ra++ += *rx++;
                else
                    for (R_xlen_t i = 0; i < n; i++, ra++, rx++)
                        if (!ISNAN(*rx)) {
                            *ra += *rx;
                            if (OP == 3) Cnt[i]++;
                        }
                break;
            }
            case INTSXP:      {
                int *ix = INTEGER(x) + (R_xlen_t)n * j;
                for (R_xlen_t i = 0; i < n; i++, ra++, ix++)
                    if (keepNA) {
                        if (*ix != NA_INTEGER) *ra += *ix;
                        else *ra = NA_REAL;
                    }
                    else if (*ix != NA_INTEGER) {
                        *ra += *ix;
                        if (OP == 3) Cnt[i]++;
                    }
                break;
            }
            case LGLSXP:
            {
                int *ix = LOGICAL(x) + (R_xlen_t)n * j;
                for (R_xlen_t i = 0; i < n; i++, ra++, ix++)
                    if (keepNA) {
                        if (*ix != NA_LOGICAL) *ra += *ix;
                        else *ra = NA_REAL;
                    }
                    else if (*ix != NA_LOGICAL) {
                        *ra += *ix;
                        if (OP == 3) Cnt[i]++;
                    }
                break;
            }
            }
        }
        if (OP == 3) {
            if (keepNA)
                for (R_xlen_t i = 0; i < n; i++) rans[i] /= p;
            else
                for (R_xlen_t i = 0; i < n; i++) rans[i] /= Cnt[i];
        }
        for (R_xlen_t i = 0; i < n; i++) REAL(ans)[i] = (double) rans[i];

        if (!keepNA && OP == 3) Free(Cnt);
        if(n > 10000) Free(rans);
    }

    UNPROTECT(1);
    return ans;
}
```

Performance concern force many R library developers to resort to C to write loopy code. This is challenging as C is more error prone and is opaque to many end users. The do_colsum() function implements the behavior of colMeans and several related functions. At a high level, the C code consist of three stylized parts

```
SEXP attribute_hidden
do_colsum(SEXP call, SEXP op,
          SEXP args, SEXP rho) {
    SEXP x, ans = R_NilValue;
    int type;
    Rboolean na;
    checkArity(op, args);
    x    = CAR(args);
    args = CDR(args);
    R_xlen_t n=asVecSize(CAR(args));
    args = CDR(args);
    R_xlen_t p=asVecSize(CAR(args));
    args = CDR(args);
    na   = !asLogical(CAR(args));
```

The first part is rather tedious as it must extract arguments passed by the caller in R. This is typically not performance critical — at least for larger arrays.

```
if (n == NA_INTEGER || n < 0)
    error(_("invalid '%s'"), "n");
if (p == NA_INTEGER || p < 0)
    error(_("invalid '%s'"), "p");
if (na == NA_LOGICAL)
error(_("invalid '%s'"), "na.rm");

int OP = PRIMVAL(op);
switch (type = TYPEOF(x)) {
case LGLSXP: break;
case INTSXP: break;
case REALSXP: break;
default:
    error(_("'x' must be numeric"));
}
```

The second part implements sanity checks to prevent the C code from crashing and thus avoid vulnerabilities. Again not performance critical.  The switch statement makes sure that the argument is of one of the type for which this makes sense.

```
PROTECT(ans=allocVector(REALSXP,p));
for(R_xlen_t j=0; j<p; j++) {
  R_xlen_t  cnt=n, i;
  LDOUBLE sum = 0.0;
  switch (type) {
  case REALSXP:   {
   double *rx = REAL(x)+ n*j;
   if(na)
      for(sum=0,i=0;i<n;i++)
        sum += *rx++;
   else
      for(cnt=sum=i=0;i<n;i++,rx++)
         if(!ISNAN(*rx)){
           cnt++; sum += *rx;
         }
   break;
```

The last part has a switch statement with a loop for computing the mean for each element type.  This has to be fast.  To sum up: R solution is polymorphic in the end user code, but as we get deeper in, the library the code becomes less elegant due to the special cases that have to be added for all variants.

The inner loop performs the type specific operations. Depending if the user has requested special handling of missing values or not, the code iterates over the array summing values and possibly counting the number of non-NA observations.

How can we do the same thing in Julia?   One step at a time, by writing code that is obviously correct

```julia
function colMeans(x, na=true, dims=1)
    dn  = size(x)
    id  = [1:dims;]                # 1:dims
    n   = prod(dn[id])
    dn  = extract(dn,id)           # dn[-id]
    pdn = prod(dn)
    res = zeros(pdn)               # 0
    for j = 0:pdn-1                # for(j in 0:pdn-1) {
        sum = z(x[1])              # 0
        cnt = 0
        off = j*n
        for i = 1:n                # for(i in 1:n) {
            v = x[i+off]
            cnt += 1               # cnt = cnt + 1
            sum += v               # sum = sum + v
        end
        res[j+1] = sum/cnt
    end
    res
end
```

The most straightforward Julia implementation is one that could be written in R/ It has the advantage of being elegant and not requiring any special casing. But does it perform well enough? But is it as polymorphic? Let's look at this in a Jupyter notebook. DEMO!

## Multi-Dispatch

```
z(x::AbstractFloat) = 0.0
z(x::Complex) = complex(0.0,0.0)
z(x) = 0

sum = z(x[1])       # 0
```

Julia functions are multi-dispatched

Types part of language syntax

To avoid boxing, variables initialized with the “right” type

Julia uses a native compiler, it generates intermediate code compiled on the fly by LLVM.  Speed comes from a combination of specialization, for each call to a function with different argument types, a new version of the function is generated, and static analysis, types are propagated through the function. This works extremely well. But static analysis can get confused. For instance, if we assign our sum variable an initial value of a different type than the result of addition, the compiler will “box” the variable, allocate it on the heap. To avoid this we specialize the initialization of the variable.

The z() function has different behavior depending on the type vector element. There is one result for all floating point types, one for complex, and one for everything else.

# Generics

```
is_na{T}(x::T) =
     x == typemin(T)

typemin{T<:Complex}(::Type{T}) =
     T(-NaN)
```

Julia lacks a builtin missing value; we steal smallest member of each data type.

Generic functions can operate over types; type variables can be bounded.

julia

Typemin is already defined for Integer and Float, we need to add one for Complex. The version for Complex returns one NaN.

The function is_na specializes on the type and compares the argument to the proper typemin.

```
if (!is_na(v))
    ...
elseif na_rm
    sum = typemin(typeof(x[1]))
```

Other changes to support missing values are straightforward.

Changes to the rest of the code to support NAs are minimal, we must check if an array element is missing and if it is either ignore it or replace sum with the right kind of NA.

# User Defined Types

```
primitive type ThreeWay 8 end
ThreeWay()          = reinterpret(ThreeWay, 0xff)
ThreeWay(x::Bool) = reinterpret(ThreeWay, x)

const true3  = ThreeWay(true)
const false3 = ThreeWay(false)
const na3    = ThreeWay()

typemin(::Type{ThreeWay}) = na3

==(x::ThreeWay, y::Bool) =
    ifelse(x==na3, false, Bool(x)==y)

+(x::Union{Int, ThreeWay}, y:: ThreeWay) =
     Int(x) + Int(y)
```

This leaves us with one challenge: how to deal with missing values for logical.  Here we are going to use one of the rather impressive capacities of Julia, we will define a primitive data type that extends Bool with a third value.  To do the job we need to define constants true, false and na, as well as define a typemin function. Lastly we need to add variants for the comparison and addition.  What is remarkable is that this type will be treated as a builtin type from now on. It can be compactly allocated in 8 bits and does not require boxing.

Reinterpret turns a value of one type into a value of another type without any checking. Certainly not typesafe.

julia

Julia achieves polymorphism and performance with a combination of three features

1. Specialization and runtime code generation
2. User defined generic data types
3. Efficient multi-dispatch

To sum up the DEMO. Is a bit slower than R/C for simple type but 2x faster for Complex. R is 60 times slower than C (and without the bytecode compiler 600x).


Now back to what caused Fortress so much distress. Multi dispatch.

## OVERLOADING * (SELECTED ENTRIES OUT OF 181 INSTANCES)

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

*(x::Number,r::Range) = range(x*first(r),x*st…

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

Three overloadings of the multiplication operator. The last one works only on either two signed or two unsigned values but not a combination.

## 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…*

And some fancier overloadings…

## OVERLOADING * (SELECTED ENTRIES OUT OF 181 INSTANCES)

```
*(X::Union{ReshapedArray{TX,2,A,MI} where
MI<:Tuple{Vararg{SignedMultInverse{Int64},N} where N} where
A<:DenseArray, DenseArray{TX,2}, SubArray{TX,2,A,I,L} where
L} where I<:Tuple{Vararg{Union{AbstCartesianIndex, Int64,
Range{Int64}},N} where N} where
A<:Union{ReshapedArray{T,N,A,MI} where
MI<:Tuple{Vararg{SignedMultInverse{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…*

… all the way to the ridiculous.

## STILL A DYNAMIC LANGUAGE

```
h(x::Int64, y::Any) = 1
h(x::Any, y::Int64) = 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)
```

Instead of preventing ambiguities they are detected. This happens once per call-site/argument combination.

## 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
```

*No representation specified*
*Abstract Type*

```
primitive type Bool <: Integer 8 end

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

*Representation specified*
*Concrete Type*

julia

And now a little bit about the type system. The only types that can be instantiated are concrete types. They are also the only types that can have fields. To ease unboxing, concrete types have no subtypes.

# TYPES ARE PARAMETRIC

Datatypes can be parametrized by types & values of primitive types

```
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 refers to types with where clauses.

## PARAMETRIC TYPES ARE INVARIANT

```
struct Point{T} <: Any          Int64 <: Signed
  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

All parametric types are invariant — again for unboxing.

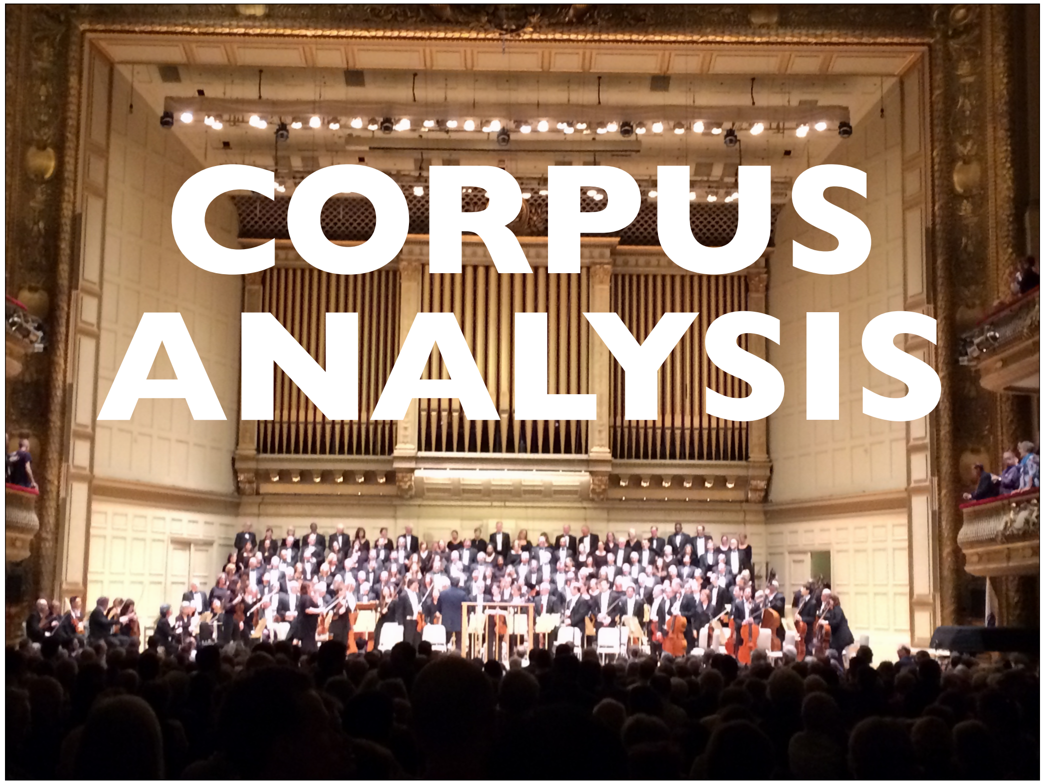
## TUPLES: ABSTRACTIONS OF FUNCTION PARAMETERS

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

Subtyping is **covariant** for tuples:

- enables sorting of method interfaces according to <:

Tuples are only used for function arguments — they cannot be created by user code.




We wanted to convince ourselves that going through the pain of formalizing the relation was worthwile. I.e. this was something that reflected actual needs rather than over engineering.

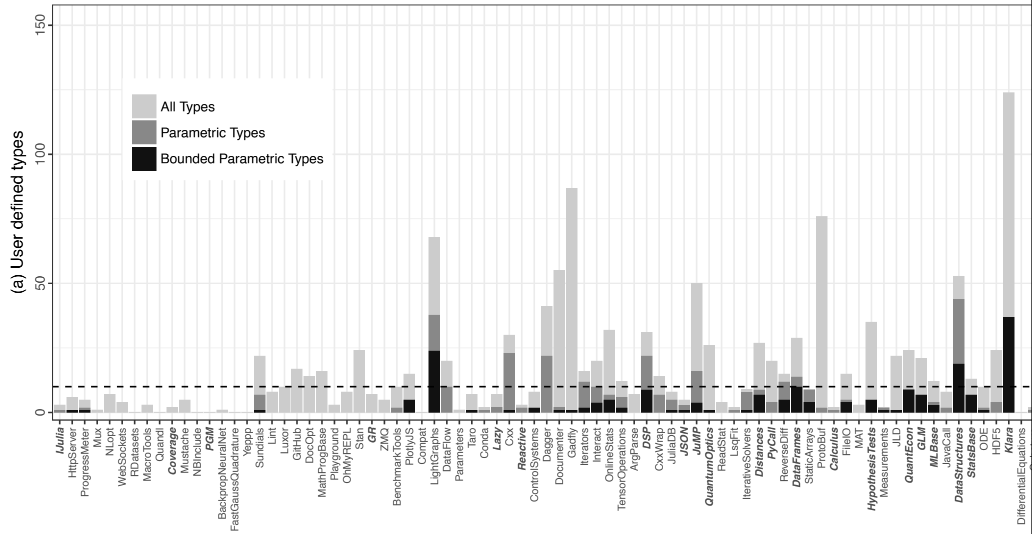
## METHODOLOGY

Take the 100 *most starred* Julia packages on GitHub

Parse source code of each package, and extract:

- the method signatures
- the declared types

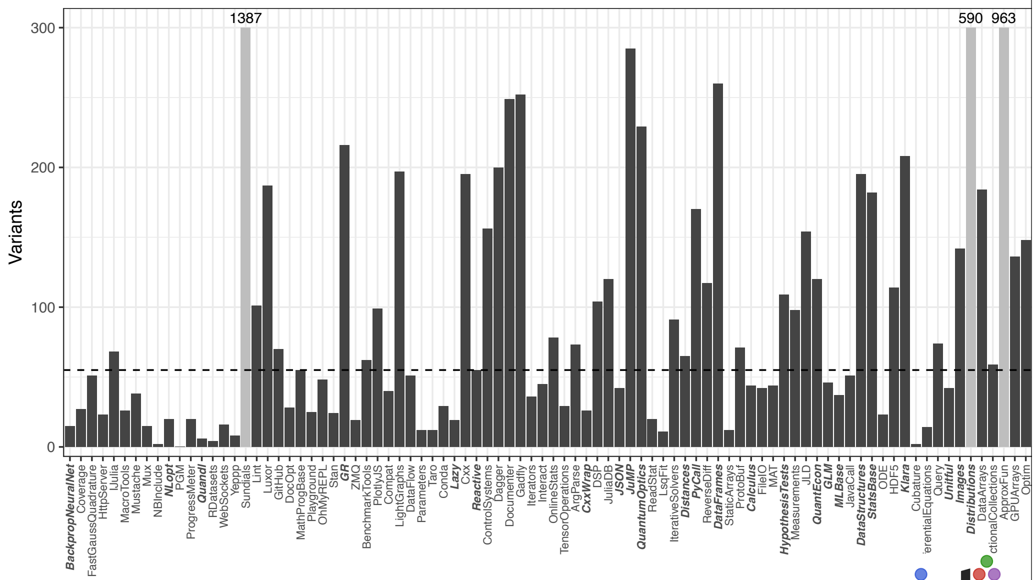
# DO PROGRAMMERS DECLARE THEIR OWN TYPES?




*Total: 1920 type declarations*

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

# NO OF METHOD DECLARATIONS

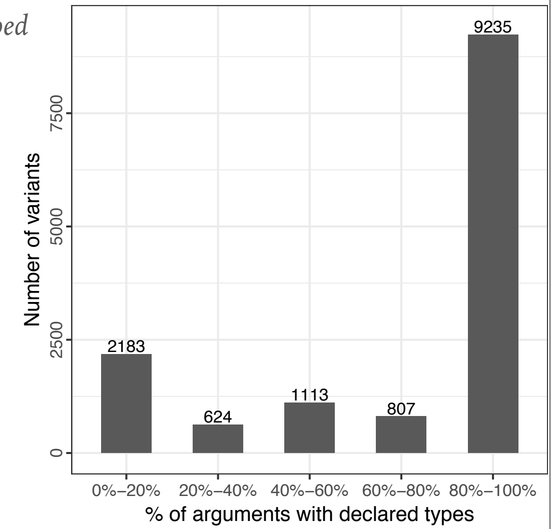


## ARE TYPED METHODS COMMON?

*65% of method signatures are fully typed*

*20% are partly typed*

*15% have no types*

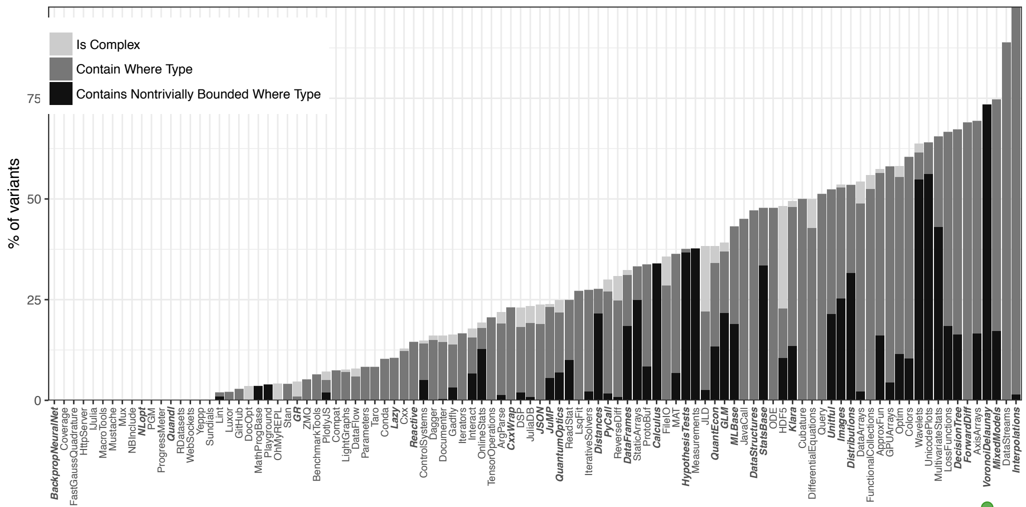



julia

So is Julia a dynamic language or a static one?  The question is how much pain would it be to get to 100%

Also — a real type system would force users to deal with ambiguities … as in Fortress.

# HOW COMPLEX ARE METHOD SIGNATURES




RECONSTRUCTING
THE SUBTYPE
RELATION

## TYPES

$t ::= \text{Any} \mid \text{name} \mid \text{Union}\{t_1..t_n\} \mid \text{Tuple}\{t_1..t_n\}$
$\mid t\{t_1..t_n\} \mid t \text{ where } t_1\text{<:}T\text{<:}t_n \mid T \mid \text{Type}\{t\}$
$\mid \text{DataType} \mid \text{Union} \mid \text{UnionAll}$

**We ignore:**

- 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

## SUBTYPING: STARTING POINTS

Parametric type application is **invariant**:

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

Tuples are **covariant**:

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

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

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

*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**

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

into

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

for **tuples at top-level**

## SUBTYPING UNIONALL

UnionAll types obey a forall /exists semantics as well

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

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

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

- 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

## 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*

$$\frac{}{E \vdash t \mathrel{<:} \mathsf{Any} : E}\ [\textsc{top}]$$

$$\frac{E \vdash t_1 \mathrel{<:} t : E_1 \quad .. \quad \mathit{merge}(E_{n-1}, E) \vdash t_n \mathrel{<:} t : E_n}{E \vdash \mathsf{Union}\{t_1, .., t_n\} \mathrel{<:} t : E_n}\ [\textsc{union\_left}]$$

$$\frac{\exists j.\, E \vdash t \mathrel{<:} t_j : E'}{E \vdash t \mathrel{<:} \mathsf{Union}\{t_1, .., t_n\} : E'}\ [\textsc{union\_right}]$$

$$\frac{E \vdash t_1 \mathrel{<:} t'_1 : E_1 \quad .. \quad E_{n-1} \vdash t_n \mathrel{<:} t'_n : E_n}{E \vdash \mathsf{Tuple}\{t_1, .., t_n\} \mathrel{<:} \mathsf{Tuple}\{t'_1, .., t'_n\} : E_n}\ [\textsc{tuple}]$$

$$\frac{\begin{array}{l} t' = \mathit{lift\_union}(\mathsf{Tuple}\{t_1, .., t_n\}) \\ E \vdash t' \mathrel{<:} t : E' \end{array}}{E \vdash \mathsf{Tuple}\{t_1, .., t_n\} \mathrel{<:} t : E'}\ [\textsc{tuple\_lift\_union}]$$

$$\frac{\begin{array}{l} \mathit{tds} \triangleq \mathit{name}\{\mathsf{T}_1, .., \mathsf{T}_m, ..\} \mathrel{<:} t'' \\ E \vdash t''[t_1/\mathsf{T}_1 .. t_m/\mathsf{T}_m] \mathrel{<:} t' : E' \end{array}}{E \vdash \mathit{name}\{t_1, .., t_m\} \mathrel{<:} t' : E'}\ [\textsc{app\_super}]$$

$$\frac{\begin{array}{l} n \leq m \quad E_0 = \mathit{add}(\mathsf{Barrier}, E) \\ \forall\, 0 < i \leq n.\ \ E_{i-1} \vdash t_i \mathrel{<:} t'_i : E'_i \ \wedge\ E'_i \vdash t'_i \mathrel{<:} t_i : E_i \\ E' = \mathit{del}(\mathsf{Barrier}, E_n) \end{array}}{E \vdash \mathit{name}\{t_1, .., t_m\} \mathrel{<:} \mathit{name}\{t'_1, .., t'_n\} : E'}\ [\textsc{app\_inv}]$$

$$\frac{E \vdash t[t_1/\mathsf{T}]\{t_2, .., t_n\} \mathrel{<:} t' : E'}{E \vdash (t \text{ where } t'_1{<:}\mathsf{T}{<:}t'_2)\{t_1, t_2, .., t_n\} \mathrel{<:} t' : E'}\ [\textsc{beta\_left}]$$

$$\frac{E \vdash t' \mathrel{<:} t[t_1/\mathsf{T}]\{t_2, .., t_n\} : E'}{E \vdash t' \mathrel{<:} (t \text{ where } t'_1{<:}\mathsf{T}{<:}t'_2)\{t_1, t_2, .., t_n\} : E'}\ [\textsc{beta\_right}]$$

$$\frac{\mathit{add}(\mathsf{Barrier}, E) \vdash t_1 \mathrel{<:} t_2 : E' \quad E' \vdash t_2 \mathrel{<:} t_1 : E''}{E \vdash \mathsf{Type}\{t_1\} \mathrel{<:} \mathsf{Type}\{t_2\} : \mathit{del}(\mathsf{Barrier}, E'')}\ [\textsc{Type\_Type}]$$

$$\frac{\mathit{add}({}^{L}\mathsf{T}^{t_2}_{t_1}, E) \vdash t \mathrel{<:} t' : E'}{E \vdash t \text{ where } t_1{<:}\mathsf{T}{<:}t_2 \mathrel{<:} t' : \mathit{del}(\mathsf{T}, E')}\ [\textsc{L\_intro}]$$

$$\frac{\begin{array}{l} \mathit{add}({}^{R}\mathsf{T}^{t_2}_{t_1}, E) \vdash t \mathrel{<:} t' : E' \\ \mathit{consistent}(\mathsf{T}, E') \end{array}}{E \vdash t \mathrel{<:} t' \text{ where } t_1{<:}\mathsf{T}{<:}t_2 : \mathit{del}(\mathsf{T}, E')}\ [\textsc{R\_intro}]$$

$$\frac{}{E \vdash t \mathrel{<:} t : E}\ [\textsc{refl}]$$

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{L}\mathsf{T}^{u}_{l} \\ E \vdash u \mathrel{<:} t : E' \end{array}}{E \vdash \mathsf{T} \mathrel{<:} t : E}\ [\textsc{L\_left}]$$

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{L}\mathsf{T}^{u}_{l} \\ E \vdash t \mathrel{<:} l : E' \end{array}}{E \vdash t \mathrel{<:} \mathsf{T} : E}\ [\textsc{L\_right}]$$

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{R}\mathsf{T}^{u}_{l} \\ E \vdash l \mathrel{<:} t : E' \end{array}}{E \vdash \mathsf{T} \mathrel{<:} t : \mathit{upd}({}^{R}\mathsf{T}^{t}_{l}, E)}\ [\textsc{R\_left}]$$

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{R}\mathsf{T}^{u}_{l} \\ \neg \mathit{is\_var}(t) \vee \mathit{search}(t, E) = {}^{R}\mathsf{T'}^{u'}_{l'} \\ E \vdash t \mathrel{<:} u : E' \end{array}}{E \vdash t \mathrel{<:} \mathsf{T} : \mathit{upd}({}^{R}\mathsf{T}^{u}_{\mathsf{Union}\{l,t\}}, E)}\ [\textsc{R\_right}]$$

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}_1, E) = {}^{R}\mathsf{T}_1{}^{u_1}_{l_1} \qquad \mathit{search}(\mathsf{T}_2, E) = {}^{L}\mathsf{T}_2{}^{u_2}_{l_2} \\ \mathit{outside}(\mathsf{T}_1, \mathsf{T}_2, E) \Rightarrow E \vdash u_2 \mathrel{<:} l_2 : E' \\ E \vdash u_1 \mathrel{<:} l_2 : E'' \end{array}}{E \vdash \mathsf{T}_1 \mathrel{<:} \mathsf{T}_2 : \mathit{upd}({}^{R}\mathsf{T}_1{}^{u_1}_{\mathsf{Union}\{\mathsf{T}_1, l_1\}}, E)}\ [\textsc{R\_L}]$$

$$\frac{\begin{array}{l} \neg \mathit{is\_var}(t_1) \\ E \vdash \mathit{typeof}(t_1) \mathrel{<:} t_2 : E' \end{array}}{E \vdash \mathsf{Type}\{t_1\} \mathrel{<:} t_2 : E'}\ [\textsc{Type\_left}]$$

$$\frac{\begin{array}{l} \mathit{is\_kind}(t_1) \ \ \mathit{is\_var}(t_2) \\ E \vdash \mathsf{Type}\{\mathsf{T}\} \text{ where } \mathsf{T} \mathrel{<:} \mathsf{Type}\{t_2\} : E' \end{array}}{E \vdash t_1 \mathrel{<:} \mathsf{Type}\{t_2\} : E'}\ [\textsc{Type\_right}]$$

The complete system is fiendishly complicated… there is much more to say… it took six months to get it right.


VALIDATION

## NEVER TRUST RULES

- **Implemented** a subtyping algorithm for Julia types
  - one-to-one mapping of rules to Julia code
  - add a search strategy on top of it
  - ~1kloc of Julia
- **Passes the subtype regression tests from Julia distribution**

## 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  50 packages
- **We validate all the logged subtype tests (~1,000,000)**
  - 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


CONCLUSIONS

## AN ORIGINAL 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

## WHY DOES THE COMPILER WORK SO WELL?

- Scientific code has some regularity that can be exploited
- Multi-dispatch encourages a style where all the type tests on the data are at method entry, body does not need checks
- Programmers have learned how to avoid boxing
- Methods can be monomorphized

Potential pathology is code size explosion but has not happened in our examples

## CAN WE REGAIN SOUNDNESS?

- Given usage data, how much do we give up to get soundness?
- Subtype algorithm  overly complex but data suggests every type rule is used by someone
- Can we evaluate the “cost” of simplifying it?
- Paley Li and Ben Chung are working on this, see NOOL17 paper.

## BOTTOM UP VS TOP DOWN LANGUAGE DESIGN

- How should we design languages, in particular languages for “new” domains?
- Pragmatic view: start with something that provides value to users, incrementally improve and evolve design in response to demand
  - Leads to warty designs
- Purist view: design an entire solution that provides the *right* answer from day one.
  - Leads to never-ending designs