Julia Subtyping: A Rational Reconstruction

FRANCESCO ZAPPA NARDELLI, Inria and Northeastern U.
JULIA BELYAKOVA, Czech Technical U. in Prague
ARTEM PELENITSYN, Czech Technical U. in Prague
BENJAMIN CHUNG, Northeastern U.
JEFF BEZANSON, Julia Computing
JAN VITEK, Northeastern U. and Czech Technical U. in Prague

Programming languages that support multiple dispatch rely on an expressive notion of subtyping to specify method applicability. In these languages, type annotations on method declarations are used to select, out of a potentially large set of methods, the one that is most appropriate for a particular tuple of arguments. Julia is a language for scientific computing built around multiple dispatch and an expressive subtyping relation. This paper provides the first formal definition of Julia's subtype relation and motivates its design. We validate our specification empirically with an implementation of our definition that we compare against the existing Julia implementation on a collection of real-world programs. Our subtype implementation differs on 122 subtype tests out of 6,014,476. The first 120 differences are due to a bug in Julia that was fixed once reported; the remaining 2 are under discussion.

1 INTRODUCTION

Multiple dispatch is used in languages such as CLOS [DeMichiel and Gabriel 1987], Perl [Randal et al. 2003], R [Chambers 2014], Fortress [Allen et al. 2011], and Julia [Bezanson 2015]. It allows programmers to overload a generic function with multiple methods that implement the function for different type signatures; invocation of the function is resolved at run-time depending on the actual types of the arguments. The expressive power of multiple dispatch stems from the way it constrains the applicability of a method to a particular set of values. With it, programmers can write code that is concise and clear, as special cases, such as optimized versions of matrix multiplication, can be relegated to dedicated methods. The inset shows three of the 181 methods implementing multiplication in Julia's standard library. The first method implements the case where a range is multiplied by a number. The second method is invoked on generic numbers: it explicitly converts the arguments to a common type via the `promote` function. The last method invokes native multiplication; its signature has a type variable `T` that can be instantiated to any integer type.

```
*(x::Number, r::Range)  = range(x*first(r),...)
*(x::Number, y::Number) = *(promote(x,y)...)
*(x::T, y::T)
  where T <: Union{Signed,Unsigned} =
    mul_int(x,y)
```

For programmers, understanding multiple dispatch requires reasoning about the subtype relation. Consider the infix call `3 * x`. If `x` is bound to a float, only the second method is applicable. If, instead, `x` is an integer, then two methods are applicable and Julia's runtime must identify the *most specific* one. Now, consider `3 * 4`, with argument type `Tuple{Int,Int}`. The signature of the first method is `Tuple{Number,Range}`. Tuples are covariant, so the runtime checks that `Int <: Number` and `Int <: Range`. Integers are subtypes of numbers, but not of ranges, so the first method is not applicable, but the second is, as `Tuple{Int,Int} <: Tuple{Number,Number}`. The third method is also applicable, as `Tuple{Int,Int}` is a subtype of `Tuple{T,T} where T<:Union{Signed, Unsigned}`; because there *exists* an instance of the variable `T` (namely `Int`) for which subtyping holds. As multiple methods are applicable, subtyping is used to compare their signatures; it holds that `Tuple{T,T} where T<:Union{Signed,Unsigned}` is a subtype of `Tuple{Number,Number}` because this holds *for all* instances of the variable `T`. The call will be dispatched, as expected, to the third method.

Subtyping can surprise programmers. For instance, is type `Tuple{String, Int}` a subtype of type `Tuple{Union{Bool, T}, T} where T`? One could choose to instantiate `T` with `Union{String, Int}`, and, in a system with union and tuple types such as [Vouillon 2004], subtyping would hold. In Julia this is not the case because of the *diagonal rule*. This rule requires that if a type variable appears more than once in covariant position, it can be instantiated only with a *concrete* type (e.g. `Int`). A `Union` is not concrete and thus cannot be used to instantiate `T`. The diagonal rule is used to restrict applicability of methods to values that have the same representation, which enables expressing common scientific computing idioms: it correctly prevents `3 * 0x4`, whose type is `Tuple{Int,UInt8}`, to dispatch to the third method above. However, the rule's interaction with other features can be complex. Consider `Tuple{Bool,Int}`; it is a subtype of `Tuple{Union{Bool, T}, T} where T` because `T` can be instantiated to `Int` and the union type matches with `Bool`, which lets us build a derivation.

Our goal is to provide an account of Julia's subtype relation that allows programmers to reason about their code, Julia implementors to evaluate the correctness of the compiler, and language designers to study an interesting point in the language design space. This has proved to be surprisingly difficult for the following three reasons. *Dynamic typing:* Julia does not have a static type system, so subtyping is only needed for multiple dispatch. Properties one would expect from such a relation may not hold. For instance, while working on this paper we discovered that, in the production implementation of subtyping, reflexivity did not hold. It was an implementation mistake that was promptly fixed, but it is telling that it went undiscovered. *No formal specification:* apart from a partial description in prose in Bezanson [2015], the only specification of subtyping is 2,800 lines of heavily optimized, undocumented C code (a snippet is shown in Fig. 1 for your enjoyment). Inspection of Julia's 2017 commit log shows that only three out of over 600 contributors made substantial edits to `subtype.c`, the file that implements it. Anecdotal evidence, based on discussion with users, suggests that the subtype relation is perceived as a black box that behaves mysteriously.

```
int forall_exists_subtype(jl_value_t *x,
  jl_value_t *y, jl_stenv_t *e, int param) {
 save_env(e,&saved,&se);
 memset(e->Lunions.stack, 0,
   sizeof(e->Lunions.stack));
 int lastset = 0; int sub;
 while (1) {
   sub = exists_subtype(x,y,e,saved,&se,param);
   int set = e->Lunions.more;
   if (!sub || !set) break;
   save_env(e, &saved, &se);
   for (int i = set; i <= lastset; i++)
     statestack_set(&e->Lunions, i, 0);
   lastset = set - 1;
   statestack_set(&e->Lunions, lastset, 1);
 }
 free(se.buf);
 return sub;
}
```

Fig. 1. Julia `subtype.c` extracted verbatim.

Unique combination of features: Julia's type language features an original combination of *nominal single subtyping*, *union types*, *existential types*, *covariant tuples*, *invariant parametric datatypes*, *distributivity*, and *singleton types*, as well as the *diagonal rule*. One source of inspiration for the design of subtyping in Julia was semantic subtyping [Frisch et al. 2002, 2008], but practical considerations caused the language to evolve in a unique direction. Table 1 illustrates Julia's unique combination of features; further discussion is in the related work section.

Given the absence of a denotational model of subtyping, it was clear from the outset that we would not be able to prove our specification correct. Instead, we provide an implementation of the subtype relation that mirrors the specification, and then validate empirically that our specification-based implementation agrees with the existing implementation. Our contributions are the following:

1. The first specification of Julia subtyping, covering all features of Julia except `Vararg` (omitted as it would decrease readability for little conceptual benefit).
2. An implementation of our specification and a validation of that implementation against the reference implementation on a suite of real-world packages.

	[Chambers ea. 94]	[Bourdoncle ea. 97]	[Litvinov 03]	[Vouillon 04]	[Frisch ea. 08]	[Cameron ea. 08]	[Smith ea. 08]	[Allen ea. 11]	Julia
Nominal Subtyping	●	◐	●	○	○	●	●	●	●
Invariant Type Constructors	○	●	●	○	◐	●	●	●	●
Covariant Type Constructors	◐[2]	●	●	●	◐[4]	○	○	○	◐[4]
Explicit Union Types	●	◐[3]	●	●	●	○	●	●	●
Distributivity[1]	○	○	○	○	●	○	○	○	●
Bounded Existential Types	○	○	○	○	○	●	◐	◐	●
Diagonal Rule	○	○	○	○	○	○	○	○	●

(1) Union/tuple or union/intersection distributivity visible to users.
(2) Built-in covariant vectors used internally for arguments but not available to users.
(3) Constraints on type parameters seen as union/intersection types.
(4) Only built-in covariant tuples.

Table 1. Julia subtyping compared.

(3) Identification of problems with Julia's design and implementation. Four bugs have been fixed and one proposal was accepted for the next revision of Julia.

Non-results. We do not provide a proof of soundness, as there is no formal semantics of Julia. We do not compare performance between our implementation and the Julia subtype code as our code is written so as to mirror our rules one to one, whereas the Julia implementation is written in C and is heavily optimized. We do not attempt to provide a "better" definition for subtyping; we leave that to future work. And, lastly, we do not prove decidability of Julia's subtyping or of its underlying algorithm.

Artifacts. Our implementation of subtyping is available in the supplementary material. All our infrastructure has been submitted for artifact evaluation. The paper web-page [Zappa Nardelli et al. 2018] complements the paper with additional data and resources.

2 BACKGROUND: JULIA

Julia is a language designed for scientific computing, released in 2012, which has achieved a degree of success — as evidenced by over 6,000 independently developed packages hosted on GitHub. Julia is a high-level, dynamic, memory-safe language without a type system but with user-defined type declarations and a rich type annotation sublanguage. Its design, therefore, reflects the tension between supporting dynamic features and ensuring efficient native code generation. As with other dynamic languages, the implementation executes any grammatically correct program and can load new code with `eval`. This is challenging for a compiler, yet Julia's LLVM-based back-end can be competitive with C [Bezanson et al. 2017].

```
f(x) = x + 1
f(x::String) = x * 3
```

While Julia has a rich type annotation language, we emphasize its lack of a static type system. The first method for function `f`, shown here, does not have type annotations on its argument and will work as long as there is an addition method for the actual value of `x`. The second method is specific to strings, but invocations will fail at run-time unless a multiplication method is provided between a string and an integer. There is no notion of soundness in Julia, even for fully type-annotated programs. If a method call does not have a most specific method, a runtime error will be reported. Ambiguity in dispatch is always resolved dynamically.

Julia types are *nominal*: the hierarchical relationship between types is specified explicitly by the programmer rather than inferred from their structure. This enables a function to behave differently on different types even if they have the same representation. Julia types are *parametric*: user-defined types can be parametrized by other types (and by values of primitive types as integers and booleans).

Top and Bottom. The abstract type `Any` is the type of all values and is the default when type annotations are omitted. The empty union `Union{}` is a subtype of all types; it is not inhabited by any value. Unlike many common languages, Julia does not have a null value or a null type that is a subtype of all types.

Datatypes. Datatypes can be *abstract* or *concrete*. Abstract datatypes may have subtypes but cannot have fields. Concrete datatypes have fields but cannot have declared subtypes. Every value is an instance of a concrete `DataType` that has a size, storage layout, supertype (`Any` if not otherwise declared), and, optionally, field names. Consider the inset definitions.

```
abstract  type   Integer <: Real      end
primitive type   Bool    <: Integer 8 end
mutable   struct PointRB <: Any   x::Real y::Bool end
```

The first declaration introduces `Integer` as a subtype of `Real`. The type is abstract; as such it cannot be instantiated. The second declaration introduces a concrete, primitive, type for boolean values and specifies that its size is 8 bits; this type cannot be further subtyped. The last declaration introduces a concrete, mutable structure `PointRB` with two fields, `x` of abstract type `Real` and `y` of concrete type `Bool`. Abstract types are always stored as references, while concrete types are unboxed.

Type Unions. A union is an abstract type which includes, as values, all instances of any of its argument types. Thus the type `Union{Integer,AbstractString}` denotes any values from the set of `Integer` and `AbstractString` values.

Parametric Datatypes. The following defines an immtuable, parametrized, concrete type.

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

`Rational`, with no argument, is a valid type, containing all instances `Rational{Int}`, `Rational{UInt}`, `Rational{Int8}`, etc. Thus, the following holds: `Rational{Int} <: Rational`. Type parameters are *invariant*, thus the following does not hold: `Rational{Int} <: Rational{Integer}`. This restriction stems from practical considerations: the memory layout of abstract types (`Integer`) and concrete types (`Int`) is different and can impact the representation of the parametric type. In a type declaration, parameters can be used to instantiate the supertype. This allows the declaration of an `AbstractVector` of as a mono-dimensional `AbstractArray` of values of type `T`:

```
abstract type AbstractVector{T} <: AbstractArray{T,1} end
```

Tuple types. Tuples are an abstraction of the arguments of a function; a tuple type is a parametrized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters, and they are *covariant* in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. `Tuple{Any}` is considered an abstract type; tuple types are only concrete if their parameters are.

UnionAll. A parametric type without argument like `Rational` acts as a supertype of all its instances (`Rational{Int}` etc.) because it is a different kind of type called a *UnionAll* type. Julia documentation describes UnionAll types as “the iterated union of types for all values of some parameter”; a more accurate way to write such type is `Rational{T} where Union{} <: T <: Any`, meaning all values whose type is `Rational{T}` for some value of `T`. UnionAll types correspond to bounded existential types in the literature, and a more usual notation for the type above would be $\exists$`T.Rational{T}`. Julia does not have explicit *pack/unpack* operations; UnionAll types are abstract. Each **`where`** introduces a single type variable. The combination of parametric and existential types is expressive: the type of 1-dimensional arrays can be simply specified by `Array{T, 1} where T`. Type variable bounds can refer to outer type variables. For example, `Tuple{T, Array{S}} where S <: AbstractArray{T} where T <: Real` refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` of any kind of array whose element type contains the type of the first tuple element. The **`where`** keyword itself can be nested. Consider the types `Array{Array{T,1}where T,1}` and `Array{Array{T, 1}, 1} where T`. The former defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. The latter type instead defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. UnionAll types can be explicitly instantiated with the type application syntax $(t\ \texttt{where T})\{t'\}$; partial instantiation is supported, and for instance `Array{Int}` denotes arrays of integers of arbitrary dimension.

Singleton Types. There are two special abstract parametric types. For an arbitrary type t, $\texttt{Type}\{t\}$ defines a type whose only instance is t itself; similarly `Val{3}` is used to create the singleton type for integer `3`.

3 SUBTYPING IN JULIA

We focus on the following grammar of types, denoted by t:

$$\begin{array}{rcl} t & ::= & \texttt{Any} \mid \texttt{Union}\{t_1, .., t_n\} \mid \texttt{Tuple}\{a_1, .., a_n\} \mid name\{a_1, .., a_n\} \mid t\ \texttt{where}\ t_1 <: \texttt{T} <: t_2 \\ & & \mid \texttt{T} \mid \texttt{Type}\{a\} \mid \texttt{DataType} \mid \texttt{Union} \mid \texttt{UnionAll} \\ a & ::= & t \mid v \end{array}$$

The variable v ranges over *plain-bit* values: in addition to types, plain-bit values can be used to instantiate all parametric types. Our only omission is the `Vararg` construct, discussed at the end of this section. We follow Julia’s conventions. We write type variables in big-caps. Given $t\ \texttt{where}\ t_1 <: \texttt{T} <: t_2$, the variable `T` binds in the type t, but not in t_1 or t_2. We abbreviate with `Bot` the empty union type `Union{}`, the subtype of all types. In the **`where`** construct, omitted lower bounds (resp. upper bounds) for type variables default to `Bot` (resp. `Any`); the notation $t\ \texttt{where T}$ is thus a shorthand for $t\ \texttt{where Bot} <: \texttt{T} <: \texttt{Any}$. We also remove empty applications and denote $name\{\,\}$ simply with *name*. We assume that all user-defined types are recorded in a global environment *tds* which for each type stores its name, attribute, type parameters with bounds, and the declared supertype. A supertype can refer to the parameters of the type being defined. Searching a type name, e.g. *name* in *tds*, returns either its definition, denoted:

$$attr\ name\{t_1 <: \texttt{T}_1 <: t'_1, .., t_m <: \texttt{T}_m <: t'_m\} <: t'' \in tds$$

or fails. The attribute, denoted *attr*, records whether the defined type is `abstract` or `concrete`. When not relevant, we omit the lower and upper bounds of the binding type variables.

Julia’s frontend simplifies types written by the programmer e.g. by removing redundant unions or parameters. We choose to formalize the subtype relation over the source syntax of types, rather than the internal Julia representation. Our approach enables reasoning about the type simplification phase itself: it is arguable that, to prevent unexpected behaviors, all frontend type transformations ought to be correct with respect to the type equivalence induced by the subtype relation. For instance this allowed us to identify a surprising behavior, discussed in Sec. 4.4.

Julia defines a *typeof* function that returns the concrete type of a value. Since types are themselves values, it is legitimate to invoke *typeof* on them, and the types `DataType`, `Union`, and `UnionAll` play the role of kinds. Intuitively, *typeof*(t) analyses the top-level constructor of t and returns `UnionAll` if it is a **`where`** construct, `Union` if it is a `Union` construct, and `DataType` otherwise. The *typeof* function plays a role in the subtyping rule for the `Type{`a`}` constructor, and we additionally rely on it to rule out badly formed types. A precise formalization of *typeof* is reported in Appendix A.

3.1 Understanding Subtyping

The literature never studied a subtype system with all the features of Julia. Unexpected, subtle, interactions between existential types and distributivity of union/tuple types forced us to depart from established approaches. We give an informal overview of the subtype relation, pointing out where, and why, standard rules fall short.

Building intuition. Two subtyping rules follow naturally from Julia's design: parametric types are *invariant* in their parameters, while tuples are *covariant*. The former follows immediately from Julia's memory representation of values. An array of dissimilar values is represented as a list of pointers to the boxed values, under type `Vector{Any}`. However, if all the values are primitive, then an unboxed representation is used. For instance, a vector of 32-bit integers is represented as an array of machine integers, under type `Vector{Int32}`. It would be wrong to treat `Vector{Int32}` as a subtype of `Vector{Any}`, as pointers can require more than 32 bits. This is *incompatible* with a semantic subtyping interpretation of the subtype relation [Frisch et al. 2002]. Invariance of type application is enforced via $name\{t_1, .., t_n\}$ `<:` $name\{t'_1, .., t'_n\}$ iff forall i, t_i `<:` t'_i and t'_i `<:` t_i. Tuples are an abstraction of the arguments of a function: covariance enables dispatch to succeed when the function arguments are a subtype of a more general function signature. Covariance of tuple types is usually enforced via `Tuple{`$t_1, .., t_n$`}` `<:` `Tuple{`$t'_1, .., t'_n$`}` iff forall i, t_i `<:` t'_i.

Subtyping union types follows instead the semantic subtyping intuition, of Vouillon [2004] or Frisch et al. [2002]. Subtyping union types is asymmetrical but intuitive. Whenever a union type appears on the left hand side of a subtyping judgment, as in `Union{`$t_1, .., t_n$`}` `<:` t, *all* the types t_1 .. t_n must be subtypes of t. In contrast, if a union type appears on the right-hand side of a judgment instead, as in t `<:` `Union{`$t_1, .., t_n$`}`, then there needs to be only one type t_i in t_1 .. t_n that is a supertype of t. Combining the two, a judgment `Union{`$t_1, .., t_n$`}` `<:` `Union{`$t'_1, .., t'_n$`}` thus reads as: *forall* types t_i, there *exists* a type t'_j such that t_i `<:` t'_j.

These rules are simple in isolation, but their interaction with other Julia features is not.

Unions and Tuples. Covariant tuples should be *distributive* with respect to unions. In particular, it should hold that:

$$\texttt{Tuple\{Union\{}t_1, t_2\texttt{\}}, t\texttt{\}} \;\texttt{<:}\; \texttt{Union\{Tuple\{}t_1, t\texttt{\}}, \texttt{Tuple\{}t_2, t\texttt{\}\}}$$

but it is known since Vouillon [2004] that the judgment cannot be derived from the above rules. The rule for tuples does not apply, while decomposing immediately the union on the right, picking up either `Tuple{`t_1, t`}` or `Tuple{`t_2, t`}` does not allow to conclude. Indeed, if a derivation commits to an element of a union type in the right-hand side before having explored all the (possibly nested) union types in the left-hand side, the derivation has effectively performed an exist/forall search, rather than a forall/exist one, losing the option to choose a different matching type for all the types in the union on the left-hand side.

A standard approach, relied upon e.g. by the CDuce language, solves this conundrum by rewriting types into their disjunctive normal form, that is, as unions of intersections of literals, *ahead-of-time*. Rewriting types as top-level unions of other types is correct in CDuce semantic model, but it is unsound in Julia, due to invariant constructors. For example, in Julia, the type `Vector{Union{Int, Bool}}`, denoting the set of vectors whose elements are integers or booleans, cannot be expressed with a top-level union. It would be incorrect to rewrite it as `Union{Vector{Int}, Vector{Bool}}`, the set of vectors whose elements are all integers or all booleans. Despite this, the distributivity law above holds in Julia and the subtype relation must successfully derive similar judgments. Julia's implementation thus relies on an efficient, but complex and fragile, backtracking mechanism to keep the forall/exist ordering of quantifications correct independently of the syntactic structure of types. This algorithm is hard to formalize and to reason about.

It is however possible to formalize an exhaustive search on top of the aforementioned rules for tuples and unions. The key intuition is that rewriting a $\texttt{Tuple\{Union\{}t_1, t_2\texttt{\}}, t\texttt{\}}$ type that occurs on the left-hand side of a subtyping judgment into the equivalent $\texttt{Union\{Tuple\{}t_1, t\texttt{\}}, \texttt{Tuple\{}t_2, t\texttt{\}\}}$ has the effect of syntactically anticipating the union types (and thus the induced forall quantifications) as much as possible in a derivation. Similarly, performing the opposite rewriting whenever a `Union` type occurs on the right-hand side of a subtyping judgment delays the existential quantifications. Care is required to lift, or unlift, **where** constructs correctly, but by adding rules that apply these rewritings *dynamically*, while building the derivation tree, we define a complete subtype relation on top of the intuitive subtype rules for tuples, invariant constructors, and union types above.

UnionAll, environments, and bounds. Julia's type system features bounded existential types, denoted $t\ \textbf{where}\ t_1 <: \mathtt{T} <: t_2$, and (confusingly) referred to as *iterated unions* or *UnionAll* types. Analogously to union types, the subtyping rules for bounded existentials must have either a forall or an exist semantics according to whether the existential appears on the left or right of the subtyping judgment. So

$$(t\ \textbf{where}\ t_1 <: \mathtt{T} <: t_2) \quad <: \quad t'$$

is satisfied if *forall* types t_3 supertype of t_1 and subtype of t_2 it holds that $t[t_3/\mathtt{T}] \ <: \ t'$. Conversely,

$$t' \quad <: \quad (t\ \textbf{where}\ t_1 <: \mathtt{T} <: t_2)$$

is satisfied if *there exists* a type t_3 supertype of t_1 and subtype of t_2 such that $t[t_3/\mathtt{T}] \ <: \ t'$. The correct quantification of a variable is specified by the position of the **where** construct that introduced it, not by where the variable occurs in the judgment. Intuitively, when checking if:

$$\texttt{Ref\{Int\}} \ <: \ \texttt{Ref\{T\}}\ \textbf{where}\ \mathtt{T}$$

the invariance of `Ref` will force us to check both `Int <: T` and `T <: Int`. In both cases, the subtyping check must be performed assuming `T` has an *exist* semantics; in this case both constraints can be satisfied by picking `T` to be `Int`.

To keep track of the semantics of each variable, we record them in an environment E. A variable `T` introduced by a **where** *on the left* of the subtyping judgment is recorded as ${}^{L}\mathtt{T}_{lb}^{ub}$, a variable introduced on the *right* as ${}^{R}\mathtt{T}_{lb}^{ub}$: lb and ub are the lower bound and upper bound types for the variable. The judgments we consider thus have the form $E \vdash t_1 \ <: \ t_2$. Given an environment in which `T` has a forall (that is, L) semantics, we distinguish two additional cases. If `T` appears on the left of the judgment, then the judgment can be satisfied only if the upper-bound for `T` is smaller than t:

$${}^{L}\mathtt{T}_{lb}^{ub} \vdash \mathtt{T} \ <: \ t \quad \text{if} \quad {}^{L}\mathtt{T}_{lb}^{ub} \vdash ub \ <: \ t\ .$$

If instead `T` appears on the right, then it is the lower bound for `T` that must be a supertype of t:

$${}^{L}\mathtt{T}_{lb}^{ub} \vdash t \ <: \ \mathtt{T} \quad \text{if} \quad {}^{L}\mathtt{T}_{lb}^{ub} \vdash t \ <: \ lb\ .$$

Right-introduced variables have exist semantics, so the least constraining bound can be chosen to satisfy a judgment, resulting in:

$$^{R}\mathtt{T}^{ub}_{lb} \vdash t \ <:\ \mathtt{T} \quad \text{if} \quad ^{R}\mathtt{T}^{ub}_{lb} \vdash t \ <:\ ub \qquad\qquad ^{R}\mathtt{T}^{ub}_{lb} \vdash \mathtt{T} \ <:\ t \quad \text{if} \quad ^{R}\mathtt{T}^{ub}_{lb} \vdash lb \ <:\ t$$

It might be surprising that variables introduced by a **`where`** on the left of the judgment suddenly appear on its right, but this is a consequence of the invariance check for type application. For instance, when checking $(\mathtt{Ref\{T\}}\ \mathbf{where}\ \mathtt{Int}\mathtt{<:}\mathtt{T}\mathtt{<:}\mathtt{Int})\ <:\ \mathtt{Ref\{Int\}}$, the `T` variable is introduced on the left, but we must prove both $^{L}\mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}} \vdash \mathtt{T}\ <:\ \mathtt{Int}$ and $^{L}\mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}} \vdash \mathtt{Int}\ <:\ \mathtt{T}$.

Matching right-introduced variables requires extra care because these types are not in subtype relation:

$$\mathtt{Tuple\{Ref\{Int\},Ref\{Bool\}\}} \not<:\ \mathtt{Tuple\{Ref\{T\},Ref\{T\}\}}\ \mathbf{where}\ \mathtt{T} \tag{1}$$

Indeed, there does not exist a type t for `T` that satisfies both the constraints $\mathtt{Int}\ <:\ t\ <:\ \mathtt{Int}$ and $\mathtt{Bool}\ <:\ t\ <:\ \mathtt{Bool}$. To account for this, whenever an existential variable is matched against a type, its bounds are updated to handle the new hypotheses on the variable, and the updated environments are propagated across the subtyping derivation tree. The actual subtyping judgment thus has the form:

$$E \vdash t_1\ <:\ t_2 \vdash E'$$

and should be read as: *in the environment E, type t_1 is a subtype of t_2, with updated constraints E'.*

For instance, the judgment:

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

states that if `T` has exist semantics and no bounds, then $\mathtt{Ref\{Int\}}\ <:\ \mathtt{Ref\{T\}}$ holds, and later uses of `T` must satisfy the updated bounds $\mathtt{Int}\ <:\ \mathtt{T}\ <:\ \mathtt{Int}$. The subtyping rule for tuples thus chains the environments across subtype tests of tuple elements. In the judgment (1) the second element $\mathtt{Ref\{Bool\}}\ <:\ \mathtt{Ref\{T\}}$ is thus checked assuming $^{R}\mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}}$ and the derivation fails accordingly.

Environment structure. The environment itself has a non-trivial structure. First, an environment E is composed of two lists, denoted by E.curr and E.past. The former, E.curr, is a stack of the variables currently in scope (growing on the right), reflecting the order in which variables have been added to the scope. In addition to variables, E.curr records *barriers*: tags pushed in the environment whenever the subtype check encounters an invariant constructor. Barriers will be discussed later. The second list, E.past, keeps track of variables which are not any longer in scope. Consider the judgment:

$$\mathtt{Tuple\{Ref\{S\}}\ \mathbf{where}\ \mathtt{S}\mathtt{<:}\mathtt{Int\}}\ <:\ \mathtt{Tuple\{Ref\{T\}\}}\ \mathbf{where}\ \mathtt{T}$$

In the derivation the variable `T` is introduced before the variable `S` and the judgment

$$^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}},\ {}^{L}\mathtt{S}^{\mathtt{Int}}_{\mathtt{Bot}} \vdash \mathtt{Ref\{S\}}\ <:\ \mathtt{Ref\{T\}} \vdash {}^{R}\mathtt{T}^{\mathtt{S}}_{\mathtt{S}},\ {}^{L}\mathtt{S}^{\mathtt{Int}}_{\mathtt{Bot}}$$

thus appears in the derivation tree. A naive rule for **`where`** would discharge the variable `S` from the environment, obtaining:

$$^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash (\mathtt{Ref\{S\}}\ \mathbf{where}\ \mathtt{S}\mathtt{<:}\mathtt{Int})\ <:\ \mathtt{Ref\{T\}} \vdash {}^{R}\mathtt{T}^{\mathtt{S}}_{\mathtt{S}}$$

The type variable `S` is still mentioned in constraints of variables in scope, but it is not any longer defined by the environment. If the variable `T` is subsequently matched against other types, the subtyping algorithm cannot know if the variable `S` appearing in the bounds of `T` has a forall or exist semantics, nor which are its bounds. Discharged variables are thus stored in E.past and accessed whenever required. The subtyping rules guarantee that it is never necessary to update the bounds of a no-longer-in-scope variable. Relying on a separate E.past environment avoids confusion when rules must determine precisely the scope of each variable, as motivated in the next paragraph.

Variables can be subject to unsatisfiable constraints. For instance, the subtype relation

$$\mathtt{Tuple\{Real,Ref\{Int\}\}} \not<: \mathtt{Tuple\{S,Ref\{T\}\}}\ \mathbf{where}\ \mathtt{S}<:\mathtt{T}\ \mathbf{where}\ \mathtt{T}$$

does not hold because the type variables are subject to the three unsatisfiable constraints below:

$$\mathtt{Real}\ <:\ \mathtt{S} \qquad \mathtt{Int}\ <:\ \mathtt{T}\ <:\ \mathtt{Int} \qquad \mathtt{S}\ <:\ \mathtt{T}$$

and in Julia, $\mathtt{Real} \not<: \mathtt{Int}$. The subtype algorithm records these constraints in the environment as $^{R}\mathtt{T}^{\mathtt{Int}}_{\mathtt{Int}}, {}^{R}\mathtt{S}^{\mathtt{T}}_{\mathtt{Real}}$, and whenever a right-variable is discharged, it checks that its lower bound is a subtype of its upper bound. In the example above, the derivation is invalidated by the failure of the consistency check for `S`:

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

From forall/exist to exist/forall. In some cases enforcing the correct ordering of type variable quantifications requires extra care. Consider the judgment:

$$\mathtt{Vector\{Vector\{T\}}\ \mathbf{where}\ \mathtt{T\}} \not<: \mathtt{Vector\{Vector\{S\}\}}\ \mathbf{where}\ \mathtt{S}$$

The type on the left denotes the set of all the vectors of vectors of elements of some type; the type on the right requires a common type for all the inner vectors. For instance the value `[[1,2],[True,False]]` belongs to the first, but not the second, type. Unfortunately, the rules sketched so far let us build a successful subtype derivation. The variables `S` and `T` are introduced in the environment, and then the left-to-right and right-to-left checks

$$^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{Bot}}, {}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T}\ <:\ \mathtt{S} \qquad \text{and} \qquad ^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{T}}, {}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{S}\ <:\ \mathtt{T}$$

are performed. These trivially succeed because for all instances of `T` there is a matching type for `S`.

Let us focus on the quantification order of the variables in the above judgment. It is still true that variables introduced on the left have a forall semantics, and variables introduced on the right have exist semantics. However, here we must find an instance of `S` such that forall `T` the judgment holds: perhaps surprisingly, the outer invariant construct `Vector` forces the inversion of the order of quantifications. Instead of a forall/exist query we must solve an *exist/forall* one. To correctly account for inversion in the order of quantifications, derivations must keep track of the relative ordering of variable introductions and invariant constructors. For this, the environment E.curr is kept ordered, and *barrier* tags are pushed in E.curr whenever the derivation goes through an invariant constructor.

We say that a variable `S` is *outside* a variable `T` in an environment E if `S` precedes `T` in E.curr and they are separated by a barrier tag in E.curr. In our running example, the first check thus becomes:

$$^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{Bot}}, \text{Barrier}, {}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T}\ <:\ \mathtt{S}$$

The environment correctly identifies the variable `S` as outside `T` and the judgment should thus be interpreted as there exists an instance of `S` such that, forall instances of `T`, `T <: S` holds. The variable `S` must thus be compared with the upper bound of `T`, deriving `Any` as lower bound:

$$^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{Bot}}, \text{Barrier}, {}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Any}\ <:\ \mathtt{S} \vdash {}^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{Any}}$$

Again, given `S` outside `T`, the right-to-left check must now prove

$$^{R}\mathtt{S}^{\mathtt{Any}}_{\mathtt{Any}}, \text{Barrier}, {}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{S}\ <:\ \mathtt{T}$$

that is, it must conclude that there exists an instance of `S` such that, forall instances of `T`, `S <: T` holds. In other terms the variable `S` must be a subtype of the lower bound of `T`. This fails, as expected.

A subtlety: whenever the forall variable is constrained tightly and quantifies over only one type, the exist/forall quantification can still correctly succeed, as in the valid judgment below:

```
Vector{Vector{T} where Int<:T<:Int} <: Vector{Vector{S}} where S
```

The diagonal rule. Consider the Julia code in the inset that defines equality in terms of equality of representations (computed by `===`) for all numerical types. This is correct provided that only values of the same type are compared, as in Julia `Int` and `Float` have different representations. The type of the `==` method is `Tuple{T,T} where T<:Number`, and the usual interpretation of `UnionAll` allows `T` to range over all allowed types, including `Number`. This type is thus equivalent to `Tuple{Number,Number}` and would match values as `(3,3.0)`, where the types of components of the tuples are different.

```
==(x::T, y::T) where T<:Number  =  x === y
```

Being able to dispatch on whether two values have the same type is useful in practice, and the Julia subtype algorithm is extended with the so-called *diagonal rule* [The Julia Language 2018]. A variable is said to be in *covariant position* if only `Tuple`, `Union`, and `UnionAll` type constructors occur between an occurrence of the variable and the `where` construct that introduces it. The *diagonal rule* states that if a variable occurs more than once in covariant position, and never in invariant position, then it is restricted to ranging over only concrete types. In the type `Tuple{T,T} where T <: Number` the variable `T` is diagonal: this precludes it getting assigned the type `Union{Int,Float}` and matching the value `(3,3.0)`. Observe that in the type `Tuple{Ref{T},T,T} where T` the variable `T` occurs twice in covariant position, but also occurs in invariant position inside `Ref{T}`; the variable `T` is not considered diagonal because it is unambiguously determined by the subtyping algorithm. Albeit this design might appear arbitrary, it is informed by pragmatic considerations; the C# language implements similar constraints (the paper web-page has an example).

Enforcing the diagonal rule involves two distinct parts: counting the occurrences of covariant and invariant variables, and checking that diagonal variables are only instantiated with concrete types. Formalizing faithfully these tasks requires some additional boilerplate. The variable entries in the subtyping environment are extended with two counters to keep track of the number of occurrences encountered in the current subtyping derivation. These counters must be updated while the derivation is built. Consider again these judgments from Sec. 1:

```
 Tuple{Bool,Int}  <:  Tuple{Union{Bool,T},T} where T
Tuple{String,Int} ≮:  Tuple{Union{Bool,T},T} where T
```

The former holds because, even if in the right-hand side the variable `T` appears syntactically twice, it is possible to build a valid derivation that matches `T` only once. The variable `T` is not diagonal in the former judgment. In a valid derivation for the latter judgment, the variable `T` must occur twice in covariant position and its final lower bound is `Union{String,Int}`, which is not a concrete type. This is only one example but, in general, subtle interactions between union and existential types do not allow counting occurrences to be correctly performed statically; it must be a *dynamic* process. The check that diagonal variables are bound only to concrete types is then performed during the validation of the consistency of the environment.

3.2 Specification of the subtyping algorithm

Our formalization of Julia subtyping is reported in Fig. 2. It closely follows the intuitions presented in the previous section.

A *variable definition*, denoted $^{L}\mathtt{T}^{ub}_{lb}$ or $^{R}\mathtt{T}^{ub}_{lb\ (co,io)}$, specifies a variable name `T`, its lower bound lb and upper bound ub, and if it has forall (L) or exist (R) semantics. To model the diagonal rule, variable definitions for R-variables additionally keep two counters: co for covariant occurrences and io for invariant occurrences. Our notation systematically omits the counters as they are only accessed and modified by the auxiliary functions *add*, *upd* and *consistent*. A *barrier* is a tag, denoted Barrier. An *environment*, denoted by E, is composed by two stacks, denoted E.curr and E.past, of variable definitions and barriers. The following operations are defined on environments, where v ranges over variable definitions and barriers:

add(v, E): push v at top of E.curr, with occurrence counters initialised to 0;

del($\mathtt{T}, E$): pop v from E.curr, check that it defines the variable $\mathtt{T}$, and push v at top of of E.past;

del(Barrier, E): pop v from E.curr and check that it is a barrier tag;

search($\mathtt{T}, E$): return the variable definition found for $\mathtt{T}$ in E.curr or E.past; fail if the variable definition is not found;

upd(${}^{R}\mathtt{T}^{ub}_{lb}, E$): update the lower and upper bounds of the variable definition $\mathtt{T}$ in E.curr; if the variable is found in E after a barrier then increase the invariant occurrence counter, and the covariant occurrence counter otherwise. Fail if the variable definition is not found;

$reset\_occ_{E'}(E)$ and $max\_occ_{E_1..E_n}(E)$: for each variable $\mathtt{T}$ defined in E, $reset\_occ_{E'}(E)$ updates its occurrence counting with the occurrence counting for $\mathtt{T}$ in E', while $max\_occ_{E_1..E_n}(E)$ updates its occurrence counting with the max of the occurrence counting for $\mathtt{T}$ in $E_1..E_n$;

consistent($\mathtt{T}, E$): search $\mathtt{T}$ in E. If the search returns ${}^{L}\mathtt{T}^{ub}_{lb}$, then return true if $E \vdash lb <: ub$ and false otherwise; while building this judgment recursive consistency checks are disabled. If the search returns ${}^{R}\mathtt{T}^{ub}_{lb\,(co,io)}$, then check if $E \vdash lb <: ub$ is derivable. If not, return false. If yes, additionally check the diagonal rule: if $co > 1$ and $io = 0$ then its lower-bound lb must be a concrete type, as checked by the *is_concrete*(lb) function. The definition of this function is non-trivial as a lower bound might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to a concrete type `Vector{Int}` only if both the upper and lower bounds of `T` equal `Int`. At the time of writing, Julia's implementation of *is_concrete* is heuristic and does not catch all possible concrete types. We omit its formalisation but our artifact includes a simple implementation. The shorthand *consistent*(E) checks the consistency of all variables in the environment E.

We assume that types appearing in a judgment are well-formed, as enforced by the *typeof* relation. We comment the subtyping rules. The rule ANY states that `Any` is the super-type of all types. The rule TUPLE_LIFT_UNION rewrites tuple types on the left-hand-side of the judgment in disjunctive normal forms, making the distributivity of unions with respect to tuples derivable. This rule can be invoked multiple times in a subtype derivation, enabling rewriting tuples in disjunctive normal form even inside invariant constructors. Rewriting is performed by the auxiliary function *lift_union*(t), which pulls union and where types out of tuples, anticipating syntactically the forall quantifications in a derivation. Symmetrically, the rule TUPLE_UNLIFT_UNION performs the opposite rewriting, delaying syntactically the exist quantifications, on union types appearing on the right-hand side of a judgment. The auxiliary function *unlift_union*(t) returns a type t' such that t = *lift_union*(t'). Finally, the rule TUPLE checks covariant subtyping of the tuple elements. The constraints generated by subtyping each element are assumed by subsequent checks, consistency is verified at the end.

The, perhaps surprising, need for the TUPLE_UNLIFT_UNION rule is due to the *complex interaction between invariant constructors, union types, and existentials*. The following judgment:

```
Ref{Union{Tuple{Int}, Tuple{Bool}}} <: Ref{Tuple{T}} where T
```

is valid because `T` can be instantiated with `Union{Int, Bool}`. However building a derivation without the TUPLE_UNLIFT_UNION rule fails. Initially the left-to-right check for invariant application generates the constraint `T >: Union{Int, Bool}`. Given the environment ${}^{R}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Union\{Int,Bool\}}}$, the right-to-left check `Tuple{T} <: Union{Tuple{Int}, Tuple{Bool}}` gets stuck trying to prove

$$\frac{}{E \vdash t \ <: \ \mathtt{Any} \ \vdash E}\ [\textsc{top}] \qquad \frac{}{E \vdash a \ <: \ a \ \vdash E}\ [\textsc{refl}] \qquad \frac{\begin{array}{l} E \vdash a_1 \ <: \ a_1' \ \vdash E_1 \ \ .. \ \ E_{n-1} \vdash a_n \ <: \ a_n' \ \vdash E_n \\ \mathit{consistent}(E_n) \end{array}}{E \vdash \mathtt{Tuple}\{a_1, \, .., a_n\} \ <: \ \mathtt{Tuple}\{a_1', \, .., a_n'\} \ \vdash E_n}\ [\textsc{tuple}]$$

$$\frac{\begin{array}{l} t' = \mathit{lift\_union}(\mathtt{Tuple}\{a_1, \, .., a_n\}) \\ E \vdash t' \ <: \ t \ \vdash E' \end{array}}{E \vdash \mathtt{Tuple}\{a_1, \, .., a_n\} \ <: \ t \ \vdash E'}\ [\textsc{tuple\_lift\_union}] \qquad \frac{\begin{array}{l} t' = \mathit{unlift\_union}(\mathtt{Union}\{t_1, \, .., t_n\}) \\ E \vdash t \ <: \ t' \ \vdash E' \end{array}}{E \vdash t \ <: \ \mathtt{Union}\{t_1, \, .., t_n\} \ \vdash E'}\ [\textsc{tuple\_unlift\_union}]$$

$$\frac{E \vdash t_1 \ <: \ t \ \vdash E_1 \ \ .. \ \ \mathit{reset\_occ}_E(E_{n-1}) \vdash t_n \ <: \ t \ \vdash E_n}{E \vdash \mathtt{Union}\{t_1, \, .., t_n\} \ <: \ t \ \vdash \mathit{max\_occ}_{E_1..E_n}(E_n)}\ [\textsc{union\_left}] \qquad \frac{\exists j. \, E \vdash t \ <: \ t_j \ \vdash E'}{E \vdash t \ <: \ \mathtt{Union}\{t_1, \, .., t_n\} \ \vdash E'}\ [\textsc{union\_right}]$$

$$\frac{\begin{array}{l} n \leq m \quad E_0 = \mathit{add}(\mathrm{Barrier}, E) \\ \forall\, 0 < i \leq n. \ \ E_{i-1} \vdash a_i \ <: \ a_i' \ \vdash E_i' \ \wedge \ E_i' \vdash a_i' \ <: \ a_i \ \vdash E_i \end{array}}{E \vdash \mathit{name}\{a_1, \, .., a_m\} \ <: \ \mathit{name}\{a_1', \, .., a_n'\} \ \vdash \mathit{del}(\mathrm{Barrier}, E_n)}\ [\textsc{app\_inv}] \qquad \frac{\begin{array}{l} \mathit{name}\{\mathsf{T}_1, \, .., \mathsf{T}_m, \, ..\} \ <: \ t'' \in \mathit{tds} \\ E \vdash t''[a_1/\mathsf{T}_1 \, .. \, a_m/\mathsf{T}_m] \ <: \ t' \ \vdash E' \end{array}}{E \vdash \mathit{name}\{a_1, \, .., a_m\} \ <: \ t' \ \vdash E'}\ [\textsc{app\_super}]$$

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

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

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

$$\frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{R}\mathsf{T}_{l}^{u} \\ E \vdash l \ <: \ t \ \vdash E' \end{array}}{E \vdash \mathsf{T} \ <: \ t \ \vdash \mathit{upd}({}^{R}\mathsf{T}_{l}^{t}, E')}\ [\textsc{R\_left}] \qquad \frac{\begin{array}{l} \mathit{search}(\mathsf{T}, E) = {}^{R}\mathsf{T}_{l}^{u} \\ (\mathit{is\_var}(t) \wedge \mathit{search}(t, E) = {}^{L}\mathsf{S}_{l_1}^{u_1}) \Longrightarrow \neg \mathit{outside}(\mathsf{T}, \mathsf{S}, E) \\ E \vdash t \ <: \ u \ \vdash E' \end{array}}{E \vdash t \ <: \ \mathsf{T} \ \vdash \mathit{upd}({}^{R}\mathsf{T}_{\mathtt{Union}\{l, t\}}^{u}, E')}\ [\textsc{R\_right}]$$

$$\frac{\begin{array}{l} \neg \mathit{is\_var}(a_1) \\ E \vdash \mathit{typeof}(a_1) \ <: \ t_2 \ \vdash E' \end{array}}{E \vdash \mathtt{Type}\{a_1\} \ <: \ t_2 \ \vdash E'}\ [\textsc{Type\_left}] \qquad \frac{\begin{array}{l} \mathit{is\_kind}(t_1) \quad \mathit{is\_var}(t_2) \\ E \vdash \mathtt{Type}\{\mathsf{T}\} \ \mathbf{where} \ \mathsf{T} \ <: \ \mathtt{Type}\{t_2\} \ \vdash E' \end{array}}{E \vdash t_1 \ <: \ \mathtt{Type}\{t_2\} \ \vdash E'}\ [\textsc{Type\_right}]$$

$$\frac{\mathit{add}(\mathrm{Barrier}, E) \vdash a_1 \ <: \ a_2 \ \vdash E' \quad E' \vdash a_2 \ <: \ a_1 \ \vdash E''}{E \vdash \mathtt{Type}\{a_1\} \ <: \ \mathtt{Type}\{a_2\} \ \vdash \mathit{del}(\mathrm{Barrier}, E'')}\ [\textsc{Type\_Type}]$$

Fig. 2. The subtype relation.

`T <: Int` or `T <: Bool`. Rule TUPLE_UNLIFT_UNION enables rewriting the right-to-left check into `Tuple{T} <: Tuple{Union{Int,Bool}}`, which is provable because the existential quantification due to the Union in the right-hand side is syntactically delayed.

Rules UNION_LEFT and UNION_RIGHT implement the forall and exist semantics for union types on the left and on the right of the subtyping judgment. In rule UNION_LEFT, the constraints generated by subtyping each element are assumed by each subsequent check and thus propagated into the final constraints. Discarding these constraints would allow proving that `Pair{Union{Int,Bool},Int}` is a subtype of `Pair{T,T} where T`, which is incorrect. However, to count correctly the occurrences of variables for the diagonal rule, each forall subderivation must reset the dynamic counting of the occurrences to that of its initial state, while the occurrences of the variables in the final state must be updated with the max of their occurrences in each intermediary state. From UNION_LEFT we immediately derive that `Union{ }` is subtype of all types, because its hypothesis is trivially validated by the forall quantification over an empty set. We conjecture that, given a type `Union{`$t_1, .., t_n$`}`, the order of the types t_i is irrelevant for subtyping, but a formal proof is non-trivial.

Type application is governed by APP_INV and APP_SUPER. When subtyping type applications with the same callee, the rule APP_INV pushes a barrier onto the environment and checks the invariance of the actual type parameters. Constraints are all propagated across all subtype checks. If all checks succeed, the latest barrier is deleted from the environment and the final constraints are passed on. A subtlety: the number of actual parameters on the right-hand side can be smaller than that on the left-hand side. It is indeed always the case that partial application gives rise to super-types; for example `Dict{Int,String} <: Dict{Int}` holds because `Dict{Int,String}` denotes all dictionaries associating integers to strings, while `Dict{Int}` denotes all dictionaries associating integers to arbitrary values: it is natural to consider the latter a supertype of the former. Rule APP_SUPER enables replacing a user-defined type by its supertype in the left-hand side of a judgment; while doing so, the rule also appropriately instantiates the type variables of the supertype.

Rules L_INTRO and R_INTRO add a **`where`** introduced variable to the current environment, specifying the relevant forall (L) or exist (R) semantics, and attempt to build a subtype derivation in this extended environment. Finally, since it gets out of scope, the introduced variable is deleted from the `curr` list and added to the `past` list. Variables with exist semantics might have had their bounds updated in unsatisfiable way; before discarding them, the consistency of their bounds is checked by the *consistent*(`T`, E) auxiliary function.

Subtyping for type variables is governed by rules L_LEFT, L_RIGHT, R_LEFT and R_RIGHT. Type variables with forall semantics are replaced with the hardest-to-satisfy bound: the upper bound if the variable is on the left of the judgment, and the lower bound if the variable is on the right. Variables with exist semantics are instead replaced with their easiest-to-satisfy bound, and, to keep track of the match, bounds of these variables are updated if a successful derivation is found, reflecting their new bound. By symmetry one would expect the rule R_LEFT to update `T` upper bound with $t \cap u$. Until recently, it was believed that, because of invariance, the explicit ordering of the checks performed by rule APP_INV or TYPE_TYPE would ensure that $t <: u$ had already been checked by rule R_RIGHT. Therefore it would always hold that $t = t \cap u$, avoiding the need to compute intersections of Julia types. To everybody surprise this turned out to be false. Consider:

```
Vector{Vector{Number}} <: Vector{Union{Vector{Number},Vector{S}}} where Int<:S<:Signed
```

This judgment contradicts the idea that `Vector{S}` can be subtype of `Vector{Number}` only if `S` is equivalent to `Number`, which is not possible here. However both Julia and our formalization can build a derivation for it: due to the existential on the right-hand side, the check that ought to ensure $t <: u$, that is `Number <: Signed`, is skipped when performing the left-to-right subtype check of the invariant constructor `Vector`. In the spirit of this work, our formalization faithfully mimic Julia behaviour. Consequences and possible mitigations to this design issue are discussed in Section 5.3.

To account for the exist/forall quantification inversion, the R_RIGHT does not apply if the type on the left is a variable with forall (that is, L) semantics and the variables are in the exists/forall quantification (the check $\neg outside(\mathtt{T}, \mathtt{S}, E)$ is responsible for this). Matching R-L variables is specially dealt by the R_L rule, which also performs the necessary outside check: if the R-variable is outside, then the bounds on the L-variable must constraint it to only one type. For this the check ub `<:` lb is sufficient, as the other direction is later verified by the environment consistency check.

Subtyping the `Type` construct is more subtle than expected. Recall that for each type or plain-bit value a, the singleton type `Type{a}` is an abstract type whose only instance is the object a. Subtyping two `Type{a}` is analogous to check invariance of constructors, as done by rule TYPE_TYPE. But there are additional cases to be considered. `Type{a}` is subtype of the type of a (e.g. `Type{1} <: Int`), as enforced by the rule TYPE_LEFT. Conversely, `Type{t}` has subtypes only if t is a type variable, and the only subtypes are kinds; the recursive check updates correctly the constraints for t.

Interestingly, reflexivity of subtyping is not derivable from these rules, due to the asymmetric treatment of L variables. Consider for instance ${}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{T}\ \mathtt{<:}\ \mathtt{T}$: the judgment ought to be true, but the subtyping rules will independently replace the left and right occurrences of `T` by upper and lower bounds, ignoring that the same variable was thus attempting to prove ${}^{L}\mathtt{T}^{\mathtt{Any}}_{\mathtt{Bot}} \vdash \mathtt{Any}\ \mathtt{<:}\ \mathtt{Bot}$. Julia 0.6.2 subtype algorithm systematically performs reflexivity checks on the fast path; reflexivity ought to hold. This is solved by explicitly adding the REFL rule to the system. Plain-bit values behave as singleton types; as such, the rule REFL is the only one that applies on plain-bit values.

We made the explicit choice of not baking transitivity into the subtype rules, expecting it to be derivable. This design choice allowed us to identify a bug in Julia 0.6.2, discussed in Sec. 4.4. More interestingly, it turned out that by exploiting empty tuples it is possible to build judgements for which transitivity does not hold, as discussed in Sec. 5.1. Although surprising, the programming practice is not affected because empty tuple types are not inhabited.

Unprovable judgments. Julia's subtype algorithm, and in turn our formalization, cannot prove all judgments expected to hold. For instance it cannot prove:

`(Tuple{T} where String<:T<:Int) <: Union{ }` or `Tuple{Union{ }} <: Union{ }`

despite all these types having no elements (the type on the left-hand side being a valid Julia type). Additionally, constraints on type variables that are declared in the type definitions, such as in `struct Foo{T<:Integer} end`, are not relied upon by subtyping; therefore it is not possible to prove judgments as `(Foo{T} where T) <: Foo{T} where T<:Integer`. For dispatch these are not issues, as similar examples do not occur in the programming practice.

Omitted features. Our work omits the `Vararg{T,N}` construct. This can be used as last parameter of a tuple to denote `N` trailing arguments of type `T`. Supporting it would add considerable boilerplate to the formalization to distinguish the case where a concrete integer has been supplied for `N` from the general case where it is left parametric, without adding interesting interactions between the type system features.

We mentioned that Julia type syntax allows to instantiate explicitly existential types, via the syntax `(`t **where** t_1`<:T<:`t_2`){`a`}`. These types are immediately rewritten by Julia frontend into their equivalent "beta-reduced" type $t[a/\mathtt{T}]$; this behavior can be modeled by a simple ahead-of-time rewrite step, which we omit for simplicity from our formalization, although it is performed by our reference implementation.

4 EMPIRICAL VALIDATION

Is the complexity of Julia's subtype relation motivated by real-world requirements? If not, then a simpler notion of subtyping may be devised. Is our specification a model of the reference implementation? Perhaps we have over-engineered a specification with unnecessary rules or missed some important corner cases.

To answer both questions we present data obtained by considering 100 popular Julia projects from GitHub. We show through static analysis that developers avail themselves to the full power of the Julia type annotation sublanguage, and dynamic analysis allows us to answer whether ours is a faithful model.

4.1 Type annotations in the real-world

The need for an expressive subtype relation must be motivated by real-world use cases. We analyzed a corpus of projects and extracted statistics on all type declarations. In Fig. 3(a) each bar depicts the total number of types declared by a project, how many of those type declarations have at least one type parameter, and how many of those apply a bound to one of their parameters (package names are reported in the paper web-page). The total number of types declared in the corpus is 2 717, with the `Merlin` package defining the per-package maximum of 204 types. The median number of types declared in a package is 15, with 60% of the packages defining at least 10 types. The total number of parametric types is 815 and the number of bound parametric types (where the bound is not trivial, i.e. `Any`) is 341.

Fig. 3(b) depicts four statistics for each project regarding type annotations on methods: of methods with at least one argument of a type other than `Any`, methods with a union or a where clause,

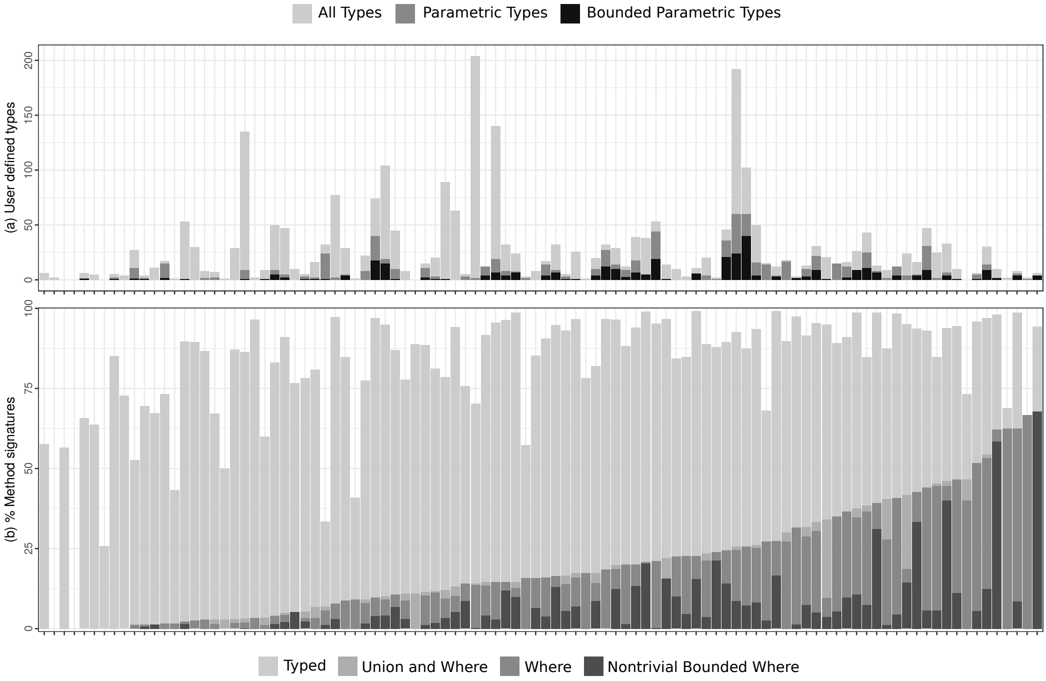

Fig. 3. (a) Type declarations (upper). (b) Proportion of complex types in methods (lower).

methods with a where clause, and methods with a where clause with at least one explicit bound. The mean proportion of union or where clauses is 18%, the mean proportion of where clauses is 16%, and the mean proportion of nontrivial where clauses is 6%. These numbers exclude the standard library: language implementors are more likely to use advanced features than end-users.

Overall these numbers suggest that programmers use the type declaration features of Julia and even the more complex where-bounded type annotations occur in most projects.

4.2 Re-implementing Subtyping

We wrote our own implementation of Julia subtyping in Julia. Our development comprises about 1,000 lines of code. For parsing types, we rely on Julia's parser but prevent front-end simplification. Our subtyping rules do not naturally define an algorithm. For instance, in the simple judgment `Union{Int,String}` <: `Union{String,Int}` two rules apply, namely UNION_LEFT and UNION_RIGHT, but a derivation requires UNION_LEFT to be used first. The challenge is thus to direct the search in the proof space. Our implementation applies the following strategy:

1. if the type on the right-hand side is `Any`, return true;
2. if UNION_LEFT applies, then it has higher priority than all other rules (including UNION_RIGHT);
3. if the left-hand side and right-hand side are both type variables, analyze their L or R annotations to check if the R_L rule applies;
4. TUPLE_LIFT_UNION and TUPLE_UNLIFT_UNION have higher priority than TUPLE;
5. replacing a type variable with its lower or upper bound is prioritary over decomposing **`where`** constructs;
6. rule APP_INV has higher priority than rule APP_SUPER.

Additionally, in rules L_LEFT, L_RIGHT, and R_L, we substitute all occurrences of the left variable `T`; this simplifies the search of the states where the REFL rule must be applied. As we mentioned, when checking consistency of the environment, nested calls to the function *consistent* are disabled and assumed to succeed. Our implementation can exhaustively explore the search space due to the UNION_RIGHT existential quantification; this is the only source of backtracking. A complete implementation of the auxiliary function *unlift_union* is complex, our implementation is heuristic.

Our implementation outputs XML traces of derivations, useful for manual inspection, and collects statistics about the rule usage. Comforting the above claims, rule usage statistics confirm that all rules are needed to validate real-world subtype judgments, including the perhaps surprising UNLIFT_UNION and R_L, used respectively 27 and 1163 times on a benchmark of 6 millions tests (full numbers on the paper web-page).

4.3 Subtyping validated

Our first benchmark is the test suites for the Julia subtype implementation internally used by Julia developers (`test/subtype.jl`): about 160 hand-picked tests inspired by bug-reports, and 335 097 subtype queries over a list of 150 types for properties of the subtype relation such as `Union{T,S} == T` implies `S <: T`. Our reference implementation passes all the tests from both test suites.

To further explore corner cases, we developed a fuzzer that generates pairs of types; it builds on the approach pioneered by Claessen and Hughes [2000] to fuzz-test a unification algorithm. The key idea is to randomly generate one term, and derive a second term by *mutation* of the first one. Our fuzzer relies on the FEAT library by Duregård et al. [2012] to enumerate exhaustively up to a certain size *pre-types* over a simplified algebra of type constructors:

$$p ::= \square \mid \texttt{Union}\{p_1, p_2\} \mid \texttt{Tuple}\{p_1, p_2\} \mid \triangle\{p\}$$

Every pre-type is then mutated by replacing the instances of the placeholder □ with either the concrete type `Int`, or the abstract type `Number`, or a type variable `T`. If a type variable is used, it is bound by a **`where`** clause at the top level. Additionally, the placeholder △ is instantiated by either the concrete type constructor `Val`, or by the abstract type constructor `Ref`. Mutating from a simplified algebra ensures that generated types satisfy the well-formedness conditions imposed by Julia. Pairs of mutated types from the same pre-type are then passed to Julia and compared with our implementation. The generated types explore many corner cases of the algorithm (e.g. type variables appearing in covariant/invariant positions, tuple and union types, and various combinations of concrete/abstract matching). The fuzzer discovered three previously unknown issues in the Julia implementation (reported in Appendix B).

Finally, to stress test our implementation on realistic workloads, we instrumented the Julia C runtime to log all the calls to the subtype function. We traced the execution of the commands `using PkgName` and `Pkg.test(PkgName)` for 100 packages. The former builds the dispatch table for the methods defined in the imported package, calling subtype to sort their types. The latter executes tests, allowing us to explore the calls to subtype performed during execution. To reduce noise in the logs, we filter duplicates and precompile all dependencies of packages before logging anything. Our implementations of `typeof` and `subtype` require that all the declarations of user-defined types (denoted *tds* in the formalization) are passed as an argument. We wrote a tool that dumps the whole subtype hierarchy loaded in a Julia instance, by starting from the type `Any` and recursively walking the type hierarchy. We compare the outcome of each logged subtype test (ignoring those for which at least one type is not well-formed) with the result returned by our implementation.

Our `subtype` implementation differs from Julia's algorithm on 122 tests out of 7,612,469 (of which 6,014,476 are non-trivial, that is they are not instances of the top or refl rule, or subtype tuples with different number of arguments). Per-package numbers are reported on the paper web-page. We have manually inspected and analyzed the failures: 120 are instances a Julia 0.6.2 bug described below, which we reported and has been acknowledged and fixed. The remaining 2 cases are also suspected to be Julia bugs and are under examination.

4.4 Julia Bugs

Since the inception of this project, we have discovered and reported several issues affecting Julia 0.6.2 subtyping. Appendix B lists all our bug reports with links to GitHub issues. Here is a discussion of the most interesting ones.

The majority of discrepancies on the realistic workload of the previous section, 120 differences out of 122, can be reduced to judgments of this form: `Tuple{Ref{Ref{T}} where T,Ref{T} where T}` <: `Tuple{Ref{S},S} where S`. Such judgments hold in Julia 0.6.2, though they should not. We reported this issue to the Julia developers and it has been promptly fixed; it was due to the incorrect treatment of variables that go out of scope.

While developing our system, we also identified corner cases of Julia subtype design which were not covered by the reference test suite. These include subtle counterexamples to reflexivity and transitivity of subtyping in Julia 0.6.2 design. The transitivity one is interesting. These two judgments hold:

$$\texttt{Tuple\{Number,Number,Ref\{Number\}\}} <: \texttt{Tuple\{T,T,Ref\{T\}\}}\ \textbf{where}\ \texttt{T} \tag{2}$$

$$(\texttt{Tuple\{T,T,Ref\{T\}\}}\ \textbf{where}\ \texttt{T}) <: \texttt{Tuple\{S,S,Ref\{Q\}\}}\ \textbf{where}\ \texttt{Q}\ \textbf{where}\ \texttt{S} \tag{3}$$

but their transitive closure does not hold:

$$\texttt{Tuple\{Number,Number,Ref\{Number\}\}} \not<: \texttt{Tuple\{S,S,Ref\{Q\}\}}\ \textbf{where}\ \texttt{Q}\ \textbf{where}\ \texttt{S}$$

Type variable `S` is diagonal, so its lower bound cannot be the abstract type `Number`. This is a design issue. In the judgment (3), it is incorrect to allow the diagonal variable `S` to be instantiated with the variable `T` because there is no guarantee that `T` itself is instantiated with a concrete type. Indeed in judgment (2) the variable `T` is not diagonal, and gets instantiated with the abstract type `Number`. Our formalization makes the problem evident: occurrence counters to identify diagonal variables are only kept and updated for right variables. The issue will be fixed in the next revision of Julia by keeping occurrence counters for left variables too.

One more surprising issue affects Julia's frontend. We mentioned that the frontend implicitly performs several simplifications when parsing and building the internal representation of types; for instance, `T where T` is replaced by `Any`, or `Union{Int,Int}` is replaced by `Int`. In general these transformations are sound, with one notable exception: simplification of types under explicit type instantiation is incorrect. In our formalization we have:

$$\texttt{(Vector\{T\} }\mathbf{where}\texttt{ T }\mathbf{where}\texttt{ S)\{Int\}} \;\not<:\; \texttt{Vector\{Int\}}$$

Julia incorrectly simplifies the type on the left-hand side as `Vector{Int}` (while it would be correct to rewrite it as `Vector{T} where T`) and concludes that subtyping holds. We have reported this arguably incorrect behavior.

5 DESIGN TRADEOFFS

In this work we set out to formalize the subtype relation as implemented by Julia 0.6.2; while doing so we have contributed to identifying both bugs in the current implementation and issues in the design. Most of these have been addressed by developers and are fixed in Julia 0.7dev, the development branch of Julia. In this section we briefly review alternative proposals to address potentially unsatisfactory design points, and discuss their implementation drawbacks.

5.1 Transitivity and uniqueness of the bottom type

In Julia, types `Tuple{Vector{T},Union{ }} where T` and `Tuple{Vector{T} where T,Union{ }}` are equivalent, and the following judgments hold:

$$\begin{aligned}\texttt{Tuple\{Vector\{T\} }\mathbf{where}\texttt{ T,Union\{ \}\}} &\;<:\; \texttt{Tuple\{Vector\{S\},Vector\{S\}\} }\mathbf{where}\texttt{ S}\\ \texttt{(Tuple\{Vector\{S\},Vector\{S\}\} }\mathbf{where}\texttt{ S)} &\;<:\; \texttt{Tuple\{Q,Q\} }\mathbf{where}\texttt{ Q}\end{aligned} \tag{4}$$

However, in Julia 0.6.2 their transitive closure does not, as we have:

$$\texttt{Tuple\{Vector\{T\} }\mathbf{where}\texttt{ T,Union\{ \}\}} \;\not<:\; \texttt{Tuple\{Q,Q\} }\mathbf{where}\texttt{ Q} \tag{5}$$

The judgment (4) holds because for all instantiations of `S` the type `Vector{S}` is concrete and `Union{ }` is a subtype of any type, while in judgment (5) the diagonal rule prevents subtyping because the type `Vector{T} where T` is abstract. This has been fixed in Julia 0.7dev by making (4) false. We argue that this solution in unsatisfactory. It is odd to have

$$\texttt{(Tuple\{Vector\{S\},Vector\{S\}\} }\mathbf{where}\texttt{ S)} \;\not<:\; \texttt{Tuple\{Q,Q\} }\mathbf{where}\texttt{ Q,}$$

while any instantiation of `S` with a concrete type, e.g. `Int`, leads to valid subtyping, e.g.

$$\texttt{Tuple\{Vector\{Int\},Vector\{Int\}\}} \;<:\; \texttt{Tuple\{Q,Q\} }\mathbf{where}\texttt{ Q.}$$

We propose an alternative design. The type `Tuple{Union{}}` (or, more generally, any tuple type containing `Union{}` as one of its components) is not inhabited by any value, and dispatch-wise it behaves as `Union{}`. However, neither Julia 0.6.2 nor our formalization can prove it equivalent to `Union{}` because the judgment `Tuple{Union{}}<:Union{}` is not derivable: following Julia 0.6.2 semantics, the *lift_union* function does not lift empty unions out of tuples. Extending *lift_union* to lift empty unions, thus rewriting types such as `Tuple{Union{}}` into `Union{}`, is straightforward;

the resulting subtype relation is not affected by the transitivity problem described above. We have modified our reference implementation along these lines. Testing over the real-world workloads does not highlight differences with the standard subtype relation, suggesting that this change does not impact the programming practice. However, this alternative design has implementation drawbacks. Assuming that a $\texttt{Tuple}\{t\}$ type in a program where t is unknown yields a 1-element tuple type becomes incorrect, making dataflow analyses more complex. Also, uniqueness of the bottom type is lost, and testing if a type is bottom (a test that occurs surprisingly often in Julia code-base) becomes slower. These tradeoffs are being investigated by Julia developers.

5.2 Ahead-of-time normalization and static detection of diagonal variables

We have seen in Sec. 3.1 that the interactions between subtyping invariant constructors, union types, and existential types make dynamic lifting and unlifting of union and existential types with respect to tuples necessary to specify a complete subtype relation. It is, however, interesting to explore if an ahead-of-time normalization phase has any benefit.

Since lifting unions across invariant constructors is unsound, our *normalization* phase rewrites tuples of unions into unions of tuples, pulling wheres out of tuples and pushing wheres inside unions, both at top-level and inside all invariant constructors. Additionally, it rewrites tuples with a bottom element into the bottom type, as suggested in Sec. 5.1.

Search over normalized types does not require the rule TUPLE_LIFT_UNION anymore, but rule TUPLE_UNLIFT_UNION is still crucial (even more so) for completeness. Despite this, ahead-of-time normalization may have benefits. At the end of Sec. 3.1 we explained that in a type such as

$$\texttt{Tuple\{Union\{Bool, T\}, T\}}\ \textbf{where}\ \texttt{T}$$

it is not possible to determine statically if the variable $\texttt{T}$ is diagonal as this depends on the type on the left-hand side of the judgment and on the derivation. However, if this type is normalized into the equivalent type

$$\texttt{Union\{Tuple\{Bool, T}_1\texttt{\}}\ \textbf{where}\ \texttt{T}_1\texttt{, Tuple\{T}_2\texttt{, T}_2\texttt{\}}\ \textbf{where}\ \texttt{T}_2\texttt{\}},$$

the confusion about the variable $\texttt{T}$ is removed: the variable $\texttt{T}_2$ is diagonal, while the variable $\texttt{T}_1$ is not. It is then straightforward to write a function *mark_diagonal* that marks variables over normalized types as diagonal whenever they occur more than once in a covariant context and never in an invariant context. In the general case, static marking can only *approximate* the dynamic counting of occurrences, for variables that appear in bounds get expanded only while building a complete derivation. However, the static counting has some nice properties.

First, a syntactic separation between diagonal and non-diagonal variables avoids subtle interactions of unions and type variables. Both Julia 0.6.2 and Julia 0.7dev, as well as our formalisation, state that the judgment below is correct:

$$(\texttt{Tuple\{Q, Int\}}\ \textbf{where}\ \texttt{Q<:Union\{Bool, S\}}\ \textbf{where}\ \texttt{S})\ \ <:\ \ \texttt{Tuple\{Union\{Bool, T\}, T\}}\ \textbf{where}\ \texttt{T}.$$

We argue that this judgment should not hold. The variable $\texttt{T}$, when matched with $\texttt{S}$ and $\texttt{Int}$, should be considered diagonal. This becomes explicit if the right-hand side is normalized into the type $\texttt{Union\{Tuple\{Bool, T}_1\texttt{\}}\ \textbf{where}\ \texttt{T}_1\texttt{, Tuple\{T}_2\texttt{, T}_2\texttt{\}}\ \textbf{where}\ \texttt{T}_2\texttt{\}}$; Julia 0.6.2 and Julia 0.7dev confirm that

$$\begin{array}{l}(\texttt{Tuple\{Q, Int\}}\ \textbf{where}\ \texttt{Q<:Union\{Bool, S\}}\ \textbf{where}\ \texttt{S}) \qquad \not<: \\ \texttt{Union\{Tuple\{Bool, T}_1\texttt{\}}\ \textbf{where}\ \texttt{T}_1\texttt{, Tuple\{T}_2\texttt{, T}_2\texttt{\}}\ \textbf{where}\ \texttt{T}_2\texttt{\}}.\end{array}$$

Additionally, it might be argued that static marking of diagonal variables makes subtyping more *predictable*. As we briefly mentioned, to address the transitivity problem of Sec. 4.4, Julia 0.7dev identifies covariant and invariant occurrences of each variable also on types that appear on the left-hand side of the judgment. Diagonal variables are not allowed to have non-diagonal variables as lower-bounds. With this in mind, consider the judgments below:

$$(\texttt{Tuple\{T, S, S\}}\ \textbf{where}\ \texttt{T <: Ref\{S\}}\ \textbf{where}\ \texttt{S})\ <:\ \texttt{Tuple\{Any, Q, Q\}}\ \textbf{where}\ \texttt{Q} \tag{6}$$

$$(\texttt{Tuple\{T, S, S\}}\ \textbf{where}\ \texttt{T <: Ref\{S\}}\ \textbf{where}\ \texttt{S})\ \not<:\ \texttt{Tuple\{Ref\{R\}, Q, Q\}}\ \textbf{where}\ \texttt{R}\ \textbf{where}\ \texttt{Q} \tag{7}$$

Both judgments exhibit the same type on the left-hand side, matched against different types. In the former the variable `S` is diagonal: it occurs twice in covariant position, and, since the subtype derivation does not use the constraint `T <: Ref{S}`, its invariant occurrence in `Ref{S}` is not counted. In the latter the derivation does use the information on the upper bound of `T`; the variable `S` is no longer diagonal, and `Q` (which is diagonal) cannot be instantiated with `S`. Programmers implement the methods of a function one at a time, possibly in different files; the lack of *predictability* of which variables are diagonal might lead to confusing dispatch behaviors. Static diagonal variable marking identifies the variable `S` as non-diagonal in both judgments: judgment (6) no longer holds and the behavior of the type on the left-hand-side thus becomes consistent with (7). We have modified our reference implementation of the subtyping algorithm to support ahead-of-time normalization and static marking of diagonal variables. This version passes the Julia regression test suites, and comparing the two algorithms over the real-world workload highlights only 41 differences. Of these, 35 of them reduce to 3 cases in which our modified algorithm cannot prove judgments due to our incomplete implementation of the *unlift_union* function. Since Julia relies on a different search mechanism, it would not be affected by these. The remaining 6 are interesting: the *typeof* function sometimes behaves differently on a normalized type, affecting the subtyping of `Type` types. For instance, *typeof*(`Tuple{Ref{T} where T}`) returns `DataType`, but if the argument is normalized, *typeof*(`Tuple{Ref{T}} where T`) returns `UnionAll`.

Summarizing: in theory subtyping based on ahead-of-time normalization and static marking of diagonal variables might constitute an alternative design of Julia subtyping. In practice the tradeoff is less clear: normalization may result in explosion of types' size, which is unacceptable for the actual implementation of Julia. It is an open question, and a future research project, to determine if a space-efficient subtyping algorithm can implement this revised relation.

5.3 Intersection types and symmetric UNION_LEFT

We have seen in Sec. 3.2 that rule R_LEFT allows arguably incorrect judgments to be derived because it propagates the new upper bound for the right variable, instead of the intersection of the old and new upper bounds. An hypothetical correct rule appears in the inset. Without native support for intersection types, computing the intersection of two arbitrary Julia types is a hard problem in itself. Julia code-base already includes a complex algorithm that computes an approximation of intersection of two types, which is used internally to compute dataflow informations, but this algorithm is too slow (and bug-ridden) to be integrated in the subtype algorithm. It should be noted that our counterexample is artificial and is unlikely to appear in the programming practice (e.g. it did not appear in the subtype calls we logged on real-world workloads, and it was not reported before), so there is a tradeoff between the extra complexity added to the implementation and the benefit of a more correct relation. In reply to our call, Julia developers have introduced a `simple_meet` function which computes intersections for simple cases; our counterexample has not been addressed yet.

$$\frac{\begin{array}{l} \mathit{search}(\mathtt{T}, E) = {}^{R}\mathtt{T}_{l}^{u} \\ E \vdash l\ <:\ t \vdash E' \end{array}}{E \vdash \mathtt{T}\ <:\ t \vdash \mathit{upd}({}^{R}\mathtt{T}_{l}^{t \cap u}, E')}$$

An ambitious redesign of Julia's *internal* type language, that would include native intersection types, has been considered, but no steps have been undertaken in this direction. This is an ambitious research project on its own.

As an aside note, we remark that support for intersection types would enable the alternative

$$\frac{E \vdash t_1 \ <:\ t \vdash E_1 \quad .. \quad E \vdash t_n \ <:\ t \vdash E_n}{E \vdash \mathtt{Union}\{t_1,\ ..,\ t_n\} \ <:\ t \vdash \mathit{merge}(E_1 \, .. \, E_n)}$$

formulation of rule UNION_LEFT in the inset. The *merge* function returns an environment in which, for each variable, the lower bound is the union of all the lower bounds for the variable in $E_1..E_n$, and the upper bound is the intersection of all the upper bounds for the variable in $E_1..E_n$. In this formulation the order of the types in the list $t_1..t_n$ is obviously irrelevant for subtyping, a property non-trivial to prove in the current formulation.

6 RELATED WORK

Surprisingly for a dynamic language, Julia's subtype relation is defined over a rich grammar of types, which often is the prerogative of statically-typed programming languages.

Languages with multimethods differ on whether parametric polymorphism is supported or not. Most previous efforts focused on non-polymorphic types, such as Cecil [Chambers and Leavens 1994], Typed Clojure [Bonnaire-Sergeant et al. 2016], and MultiJava [Clifton et al. 2000]. Subtyping is used to check that classes implement all of the required methods of their supertypes. The subtype relations are defined over covariant tuples and discrete unions. Approaches that combine multimethods with parametric polymorphism are more involved. The earliest work, ML<: [Bourdoncle and Merz 1997], extends ML with subtyping and multimethods and shows that the type system is sound and decidable by showing that the constraint solving system that it uses to handle both subtyping and pattern matching is decidable. Following the ML polymorphism, types have only top-level quantifiers (for example, $\forall\alpha.\mathtt{list}[\alpha]$ is allowed but not $\mathtt{list}[\forall\alpha.\mathtt{list}[\alpha]]$), with subtyping being defined over monotypes. Constraints on type variables partially model union types: for instance, the type $\forall\alpha.(\mathtt{int}{<:}\alpha, \mathtt{bool}{<:}\alpha).\alpha$ can be understood as the set union of `int` and `bool`. Due to the lack of nesting quantifiers, this does not equate to Julia's union types.

Universal polymorphism and parametric multimethods have been proposed in Mini-Cecil [Litvinov 1998, 2003]. Similar to ML<:, universal types have only top-level quantifiers. Fortress [Allen et al. 2011], in addition, supports arrow types, and internally uses both universal and existential types, with top-level quantifiers. Mini-Cecil and Fortress both use a constraint generation strategy to resolve subtyping; they support union and intersection types but do not provide distributivity "in order to simplify constraints solving" [Litvinov 2003]. For Mini-Cecil typechecking is decidable. Fortress argued decidability based on [Castagna and Pierce 1994], though no proof is provided.

Frisch et al. [2002, 2008] studies the semantic interpretation of subtyping with union, intersection, negation types, and function types. Types are interpreted as sets of values; base types have their own denotation, and all the operators on types correspond to the equivalent set theoretical operations. Subtyping is defined semantically as inclusion of the sets denoted by types. The main challenge of the approach is due to the function types. However, only one type operator, namely, reference types, is described as an extension. An important contribution was a sound and complete algorithm to decide semantic subtyping. The algorithm crucially relies on semantic properties of their domain, in particular that types can be rewritten in disjunctive-normal form. As we have seen, Julia does not fully embrace semantic subtyping, and due to interactions between union types, invariant constructors, and existential types, search is considerably more complex. [Castagna et al. 2015, 2014] extended their system with parametric polymorphism: terms with type variables are first compiled away to a variable-free core language with a type-case construct. Similar to [Frisch et al. 2002, 2008], their interpretation of types differs from Julia's.

Subtyping of union types in Julia builds on [Vouillon 2004], which proposes an algorithm to decide subtyping of union types in a statically typed language with functions and pairs but without union/pair distributivity. The same paper also considers an extension of the language with ML-style polymorphism and co-/contravariant type constructors, but not invariant ones.

Bounded existential types have been used to model Java wildcards [Cameron et al. 2008; Tate et al. 2011]: for instance, the wildcard type `List<?>` can be represented as an existential type `∃T.List<T>`. Wildcards are less expressive than where-types in Julia, because they cannot denote types such as `∃T.List<List<T>>` while `List<List<?>>` corresponds to `List<∃T.List<T>>`. Nevertheless, inexpressible types may appear during typechecking, and therefore the formal models use the full grammar of existential types, and so does subtyping. Java wildcards do not have to deal with structural type constructors of Julia, such as union types, covariant tuples, and their distributivity. This allows for simpler subtyping rules for existential types that rely on well-developed machinery of unification and explicit substitution for type variables. Subtyping of wildcards with union and intersection types are studied in [Smith and Cartwright 2008]. Though the paper mentions that a distributivity rule is a desired extension of subtyping, the rule is omitted due to a "tedious normalization steps" that would have been needed. As our experience shows, in presence of invariant constructors normalization does not solve all the problems, and should be accompanied by "unlifting unions" (recall the example `Ref{Union{Tuple{Int}, Tuple{Bool}}} <: Ref{Tuple{T}} where T`).

In type systems with bounded existential types, as well as type systems with nominal subtyping and variance, decidability of subtyping has been a major concern [Kennedy and Pierce 2007; Wehr and Thiemann 2009]. By design, Julia lacks language features that are known to cause undecidability. Firstly, unlike in traditional existential types [Pierce 1992], types such as $\exists \mathtt{T}_{t_1}^{t_2}.\mathtt{T}$ are instantaneously rewritten into the upper bound of `T` by the frontend and do not appear in subtyping. Secondly, unlike in Java, where subtyping has been proved undecidable [Grigore 2017], neither of the following is allowed in Julia: recursive bounds on type variables (e.g. `Ref{T} where T<:Foo{T}`) [Wehr and Thiemann 2009], contravariant type constructors [Kennedy and Pierce 2007], existential types in type definitions (e.g. `struct Foo{T} <: Bar{S>:T} where S`) [Tate et al. 2011]. An unpublished manuscript on decidability of type checking for Java wildcards [Mazurak and Zdancewic 2005], while failing on modeling of a particular language feature [Zdancewic 2006], develops a formal machinery for updating environments which resembles ours.

A simplified kernel of the subtype relation was documented in Bezanson PhD thesis [Bezanson 2015], together with a minimal Julia implementation of the algorithm. This effort introduced some ideas: for example, it sketches the strategy to update the bounds on type variables. But it was neither complete nor correct, and reflected an older version of Julia's type system. In particular, it ignored the subtle rules that govern propagation of the constraints, and the exist/forall quantifier inversion; it did not support user-defined parametric types or the diagonal rule.

7 CONCLUSION

We have provided a specification of the subtype relation of the latest release of the Julia language. In many systems answering the question whether $t_1 <: t_2$ is an easy part of the development. It was certainly not our expectation, approaching Julia, that reverse engineering and formalizing the subtype relation would prove to be the challenge on which we would spend our time and energy. As we kept uncovering layers of complexity, the question whether all of this was warranted kept us looking for ways to simplify the subtype relation. We did not find any major feature that could be dropped. Indeed, we carried out static and dynamic analysis of the core libraries, as well as of 100 popular Julia packages, and found that both language implementors and end-users avail themselves of the full power of the type sublanguage. The usage of the advanced features of Julia type system is widespread in both groups. Our formalization enables the study of the metatheory of the subtype relation; the system is intricate and even simple properties require complex reasoning. Additionally, it is not *a priori* clear if it is possible to define the subtype relation in a more declarative style. Arguably these would be research contributions in their own right.

As a separate contribution, to validate our formalization and to explore the implementation challenges of the subtype algorithm, we have provided a proof-of-concept implementation that closely mirrors our specification and relies on a simple search strategy. Our experimental results show that this implementation captures the real subtype relation, and is useful in identifying issues in the Julia implementation.

At the outset, our ambition was to simplify the subtype relation, as we felt that it had too many corner cases. While our specification is more elegant and easier to understand than C code, we were not able to identify major features of the algorithm that could be eliminated. The design of the relation is motivated by seemingly reasonable end-user expectations and requirements.

REFERENCES

Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy Steele. 2011. Type Checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2048066.2048140

Jeff Bezanson. 2015. *Abstraction in technical computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/99811

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical optional types for Clojure. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-49498-1_4

François Bourdoncle and Stephan Merz. 1997. Type Checking Higher-order Polymorphic Multi-methods. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263743

Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-540-70592-5_2

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676991

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-theoretic Types: Part 1: Syntax, Semantics, and Evaluation. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2535838.2535840

Giuseppe Castagna and Benjamin C. Pierce. 1994. Decidable Bounded Quantification. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/174675.177844

Craig Chambers and Gary T. Leavens. 1994. Typechecking and Modules for Multi-methods. In *Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/191080.191083

John Chambers. 2014. Object-Oriented Programming, Functional Programming and R. *Statist. Sci.* 2 (2014). Issue 29. https://doi.org/10.1214/13-STS452

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/353171.353181

Linda DeMichiel and Richard Gabriel. 1987. The Common Lisp Object System: An Overview. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/3-540-47891-4_15

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364506.2364515

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.2002.1029823

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008). https://doi.org/10.1145/1391289.1391293

Radu Grigore. 2017. Java Generics Are Turing Complete. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3009837.3009871

Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*. https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/

Vassily Litvinov. 1998. Constraint-based Polymorphism in Cecil: Towards a Practical and Static Type System. In *Addendum to the Conference on Object-oriented Programming, Systems, Languages, and Applications.* https://doi.org/10.1145/346852.346948

Vassily Litvinov. 2003. *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages.* Ph.D. Dissertation. University of Washington.

Karl Mazurak and Steve Zdancewic. 2005. Type Inference for Java 5: Wildcards, F-Bounds, and Undecidability. (2005). https://pdfs.semanticscholar.org/a73a/aace3ecafb9f8f4f627705647115c29ef63e.pdf unpublished.

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/143165.143228

Allison Randal, Dan Sugalski, and Leopold Toetsch. 2003. *Perl 6 and Parrot Essentials.* O'Reilly.

Daniel Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It?. In *Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA).* https://doi.org/10.1145/1449764.1449804

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/1993498.1993570

The Julia Language. 2018. Manual: Diagonal Types. (2018). Retrieved 2018-07-24 from https://docs.julialang.org/en/v0.6.1/devdocs/types/#Diagonal-types-1

Jerome Vouillon. 2004. Subtyping Union Types. In *Computer Science Logic (CSL).* https://doi.org/10.1007/978-3-540-30124-0_32

Stefan Wehr and Peter Thiemann. 2009. On the Decidability of Subtyping with Bounded Existential Types. In *Programming Languages and Systems (ESOP).*

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: a Rational Reconstruction — Project Web-Page. (2018). Retrieved 2018-07-24 from https://www.di.ens.fr/~zappa/projects/lambdajulia/

Steve Zdancewic. 2006. A Note on "Type Inference for Java 5". (2006). https://web.archive.org/web/20060920024504/http://www.cis.upenn.edu/~stevez/note.html

A THE *typeof*(t) FUNCTION

Julia's *typeof* function returns the concrete type of each value. Since types are themselves values, it is legitimate to invoke *typeof* on them, and the types `DataType`, `Union`, and `UnionAll` play the role of kinds. Indeed, the auxiliary function *is_kind*(t) returns true if t is `DataType`, or `Union`, or `UnionAll`). Since the *typeof* function plays a role in the definition of subtyping, we provide its formalization in Fig. 4. We use an environment G to store the type variables in scope. We write *typeof*(t, G) as a shorthand for there exists t' such that *typeof*$(t, G) = t'$.

We have seen that Julia's frontend implicitly performs several simplifications when processing types; for instance, `T where T` is replaced by `Any`, or `Union{Int, Int}` is replaced by `Int`; these simplifications must be taken explicitly into account when formalizing the *typeof* relation. The auxiliary function *simplify*(t) performs the following simplification steps:

- simplify trivial **where** constructs, e.g.: replace `T` **where** `T <:` t_2 by t_2 and replace t **where** `T` by t whenever `T` $\notin fv(t)$;
- remove redundant `Union` types, e.g.: replace `Union{`t`}` by t;
- remove duplicate and obviously redundant types from `Union{`$t_1, .., t_n$`}` types.

Given a list of types $t_1, .., t_n$, a type t_i is obviously redundant whenever there exists a type t_j which is its supertype given an empty variable environment. These simplifications are guided by pragmatic considerations rather than semantic issues. As such they tend to vary between Julia versions, and we do not explicitly formalize them; our reference implementation mimics the simplification behavior of Julia 0.6.2, apart for the issue described in Sec. 4.4.

The function *typeof*(t) returns `Union` if the type t, after simplification or instantiation of trailing **where** constructs, is a union. The case for `UnionAll` is similar, except that trailing **where** constructs and the instantiation of user defined parametric types must be taken into account. In all other cases, a type has the `DataType` kind.

$$
\begin{array}{rlll}
\mathit{typeof}(a) = \mathit{typeof}(\mathit{simplify}\,(a), \emptyset) & & \\[1em]
\mathit{typeof}(t, G) = \texttt{DataType} & \textbf{if} & \mathit{is\_kind}(t) \\
& \textbf{or} & t = \texttt{Any} \\
& \textbf{or} & t = \texttt{Tuple}\{t_1,\ ..\, , t_n\} \ \textbf{and}\ \forall i, \mathit{typeof}(t_i, G) \\
& \textbf{or} & t = \texttt{T} \ \textbf{and}\ \texttt{T} \in G \\
& \textbf{or} & t = (t\ \textbf{where}\ t_1 \mathrel{<:} \texttt{T} \mathrel{<:} t_2)\{t'\} \ \textbf{and}\ \mathit{typeof}(t', G) \\
& & \quad \textbf{and}\ \mathit{typeof}(t_1, G) \ \textbf{and}\ \mathit{typeof}(t_2, G) \ \textbf{and}\ t_1 \mathrel{<:} t' \mathrel{<:} t_2 \\
& & \quad \textbf{and}\ \mathit{typeof}(t[t'/\texttt{T}], G) = \texttt{DataType} \\
& \textbf{or} & t = \mathit{name} \ \textbf{and}\ \mathit{attr\,name}\{\,\} \mathrel{<:} t' \in \mathit{tds} \\
& \textbf{or} & t = \mathit{name}\{t_1,\ ..\, , t_n\} \ \textbf{and}\ \forall i, \mathit{typeof}(t_i, G) \\
& & \quad \textbf{and}\ \mathit{attr\,name}\{t'_1 \mathrel{<:} \texttt{T}_1 \mathrel{<:} t''_1,\ ..\, , t'_n \mathrel{<:} \texttt{T}_n \mathrel{<:} t''_n\} \mathrel{<:} t''' \in \mathit{tds} \\
& & \quad \textbf{and}\ \forall i, \mathit{fv}(t'_i, t_i, t''_i) = \emptyset \implies t'_i \mathrel{<:} t_i \mathrel{<:} t''_i \\[1em]
\mathit{typeof}(t, G) = \texttt{Union} & \textbf{if} & t = \texttt{Union}\{t_1,\ ..\, , t_n\} \ \textbf{and}\ \forall i, \mathit{typeof}(t_i, G) \\
& \textbf{or} & t = (t\ \textbf{where}\ t_1 \mathrel{<:} \texttt{T} \mathrel{<:} t_2)\{t'\} \ \textbf{and}\ \mathit{typeof}(t', G) \\
& & \quad \textbf{and}\ \mathit{typeof}(t_1, G) \ \textbf{and}\ \mathit{typeof}(t_2, G) \ \textbf{and}\ t_1 \mathrel{<:} t' \mathrel{<:} t_2 \\
& & \quad \textbf{and}\ \mathit{typeof}(t[t'/\texttt{T}], G) = \texttt{Union} \\[1em]
\mathit{typeof}(t, G) = \texttt{UnionAll} & \textbf{if} & t = t_1\ \textbf{where}\ \texttt{T} \ \textbf{and}\ \mathit{typeof}(t_1, \texttt{T}; G) \\
& \textbf{or} & t = (t\ \textbf{where}\ t_1 \mathrel{<:} \texttt{T} \mathrel{<:} t_2)\{t'\} \ \textbf{and}\ \mathit{typeof}(t', G) \\
& & \quad \textbf{and}\ \mathit{typeof}(t_1, G) \ \textbf{and}\ \mathit{typeof}(t_2, G) \ \textbf{and}\ t_1 \mathrel{<:} t' \mathrel{<:} t_2 \\
& & \quad \textbf{and}\ \mathit{typeof}(t[t'/\texttt{T}], G) = \texttt{UnionAll} \\
& \textbf{or} & t = \mathit{name} \ \textbf{and}\ \mathit{attr\,name}\{\texttt{T}_1,\ ..\, , \texttt{T}_n\} \in \mathit{tds} \\
& \textbf{or} & t = \mathit{name}\{t_1,\ ..\, , t_n\} \ \textbf{and}\ \forall i, \mathit{typeof}(t_i, G) \\
& & \quad \textbf{and}\ \mathit{name}\{t'_1 \mathrel{<:} \texttt{T}_1 \mathrel{<:} t''_1,\ ..\, , t'_n \mathrel{<:} \texttt{T}_n \mathrel{<:} t''_n,\ ..\} \mathrel{<:} t''' \in \mathit{tds} \\
& & \quad \textbf{and}\ \forall i, \mathit{fv}(t'_i, t_i, t''_i) = \emptyset \implies t'_i \mathrel{<:} t_i \mathrel{<:} t''_i \\[1em]
\mathit{typeof}(v, G) = \ldots & & \textit{return the type tag of the value } v
\end{array}
$$

Fig. 4. The *typeof*(t) function.

The *typeof* function additionally checks that a type is well-formed, and in particular that, for all type variable instantiations, all actual types respect the bounds of binding variable. This check cannot be performed if the bounds or the variable itself have free variables, and in some cases Julia allows unsatisfiable bounds to be defined. For instance, the type `Foo{S}` **where** `S >: String` is accepted even assuming that `Foo` is declared as `abstract type Foo{T<:Int} end`. Since there is no type that is subtype of `Int` and supertype of `String`, this type denotes an empty set of values. Well-formedness of type-definitions can be checked along similar lines, keeping in mind that all parameters bind in the supertype, and each parameter binds in all bounds that follow.

B ISSUES REPORTED TO JULIA DEVELOPERS

The complete list of the issues we reported to the Julia bug tracker since starting this project follows. For each report, the number (and active link) in parentheses is the issue id's in Julia's github database. We distinguish between bug reports that have been fixed, bug reports that have been acknowledged and for which a solution is currently being investigated, and other design improvement proposals.

B.1 Fixed Bugs

(1) **Reflexivity and transitivity broken due to buggy diagonal rule** (#24166)
Flaws in the implementation of the diagonal rule check lead invalidate expected properties of the subtype relation, as discussed in Sec. 4.4. These flaws are observable in Julia 0.6.2 but have been fixed in the development version.

(2) **Propagation of constraints when subtype unions** (#26654)
The order of types inside a Union constructor should not affect the subtype relation (a property we call symmetry of Union). The subtype algorithm however traverses the types inside a Union constructor in a precise order. Incorrect propagation of constraints during subtyping made subtyping dependent on the order of types inside a Union constructor, as highlighted by the Julia 0.6.2 behavior below:

```
julia> Ref{Union{Int, Ref{Number}}} <: Ref{Union{Ref{T}, T}} where T
true
```

This issue was found by our fuzz tester. It has been fixed in the development version.

(3) **`Union{Ref{T}, Ref{T}}` and `Ref{T}` behave differently** (#26180)
This bug was introduced after the Julia 0.6.2 release:

```
julia> Ref{Union{Ref{Int}, Ref{Number}}} <: Ref{Ref{T}} where T
false

julia> Ref{Union{Ref{Int}, Ref{Number}}} <: Ref{Union{Ref{T}, Ref{T}}} where T
true
```

The second check should return `false`, as the first one, because the two types on the right-hand side are equivalent. This bug was found by our fuzz tester. It has been fixes in the development version (with the same commit that fixes the previous bug report).

B.2 Open Issues

(1) **Missing intersection types** (#26131)

```
julia> Vector{Vector{Number}}  <:
        Vector{Union{Vector{Number}, Vector{S}}} where S<:Integer
true
```

As discussed in Sec. 3.2, this query should return `false` because `Vector{S}` is not a subtype of `Vector{Number}` when `Vector{S}<:Integer`. To correctly derive similar judgments, the subtype algorithm must be able to compute the intersection of types. This is a hard problem in itself. As a temporary band-aid, in reply to our call, Julia developers have introduced a `simple_meet` function which computes intersections for simple cases. The current implementation is still too weak to handle this particular case. The fact that not computing the intersection of the upper bounds in rule R_LEFT might be source of problems in presence of union types was suggested by an anonymous reviewer; our example is built on top of reviewer's remark.

(2) **Stack overflows / Loops in subtype.c `subtype_unionall`**
Unexpected looping inside the subtype algorithm, or large computer-generated types, can make Julia subtype algorithm to exceed the space allocated for the recursion stack. We reported this issue on a large computer-generated type (#26065). We discovered later that other reports address a similar issue; some are referenced in the ticket above, but some are more recent (#26487).

(3) **Inconsistent constraints are ignored** (#24179)
Frontend simplification rewrites types of the form `T where lb<:T<:ub` into the upper bound `ub`, without checking first if the user-specified bounds are inconsistent, as in:

```
julia> T where String<:T<:Signed
Signed
```

This may lead to unexpected results in subtype queries, and the type above is not considered equivalent to the `Union{}`. Julia developers agree this behavior is incorrect.

(4) **Diagonality is ignored and constraints are missing when matching with union** (#26716)
Both Julia 0.6.2 and 0.7-dev incorrectly return `true` on these judgments (on the left types are equivalent, on the right it is the same type):

```
julia> (Tuple{Q,Bool} where Q<:Union{Int,P} where P) <: Tuple{Union{T,Int}, T} where T
true

julia> (Tuple{Union{Int,P},Bool} where P) <: Tuple{Union{T,Int}, T} where T
true

julia> (Union{Tuple{Int,Bool}, Tuple{P,Bool}} where P) <: Tuple{Union{T,Int}, T} where T
true
```

The correct answer is `false` because the variable `T` should be considered diagonal and gets matched both with `P` and `Bool`, and as such it cannot be concrete. This is confirmed by rewriting into an equivalent type by the *lift_union* function, thus making the diagonal variable explicit. In this case Julia returns the correct answer.

B.3 Proposals we made

(1) **Interaction of diagonal rule and lower bounds** (#26453)
Whenever the programmer specifies explicitly a lower bound for a type-variable, as in `Tuple{T,T}where T>:t`, it is not always easy to decide if `T` should be considered diagonal or not. This depends on whether the lower bound, `t`, is concrete, but in general deciding concreteness is hard and Julia implementation approximates it with an heuristic. We proposed that the variables should be considered diagonal only if their concreteness is obvious. The proposal was approved, implemented and merged into the `master` branch.

(2) **Another approach to fix problem with concreteness of `Vector{T}` / transitivity** (comment #372746252).
A subtle interaction between the bottom type and the diagonal rule can break transitivity of the subtype relation. We propose an alternative approach to fix the issue, as the solution to the problem applied in Julia seems unsatisfactory.