

Decidable Subtyping of Existential Types for the Julia Language

Julia Belyakova

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts, USA


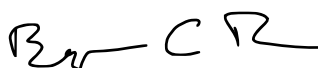
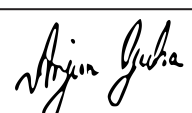

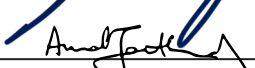
August 2023

Thesis Title: Decidable Subtyping of Existential Types for the Julia Language

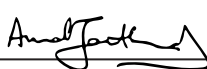
Author: Yulia Belyakova

PhD Program: X Computer Science Cybersecurity Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

 _____ Thesis Advisor	<u>08/02/2023</u> _____ Date
Benjamin C. Pierce  _____ Thesis Reader	<u>08/02/2023</u> _____ Date
Arjun Guha  _____ Thesis Reader	<u>08/02/2023</u> _____ Date
Giuseppe Castagna  _____ Thesis Reader	<u>08/02/2023</u> _____ Date
Amal Ahmed  _____ Thesis Reader	<u>08/02/2023</u> _____ Date

KHOURY COLLEGE APPROVAL:

 _____ Associate Dean for Graduate Programs	<u>08/11/2023</u> _____ Date
--	------------------------------------

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

 _____ Recipient's Signature	<u>14 August 2023</u> _____ Date
---	--

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

Abstract

Julia is a dynamic, high-performance programming language for scientific computing. To encourage a high level of code reuse and extensibility, Julia is designed around symmetric multiple dynamic dispatch, which allows functions to have multiple implementations tailored to different argument types. To resolve multiple dispatch, Julia relies on a subtype relation over a complex language of run-time types and type annotations, which include set-theoretic unions, distributive tuples, parametric invariant types, and impredicative existential types. Notably, subtyping in Julia is undecidable, which manifests with a run-time stack-overflow error when program execution encounters a subtyping query that causes the subtype checker to loop.

In this dissertation, I propose a decidable subtype relation for a restricted language of Julia types where existential types inside invariant constructors are limited to ones expressible with use-site variance. To estimate the migration effort that would be required for switching to the restricted type language, I analyze type annotations in the corpus of 9K registered Julia packages. Out of 2M statically identifiable type annotations in the corpus, 99.99% satisfy the restriction, making it a viable candidate for evolving Julia towards decidable subtyping.

ACKNOWLEDGEMENTS

I would not be able to do this without plenty of luck and so many good people who helped me along the way.

I thank my family, especially my mom, for all their love, support, and trust. Without you all, I would have never embarked on this journey.

I dedicate this thesis to my late grandparents. Research took me far away and I could not be with you, but I know you would be proud.

I thank my husband, Artem, for his endless love and care. Without you, I would not have finished this journey.

I thank my daughter, Sophia, for bringing me joy on tough days and giving me strength to go on.

I thank my advisor, Jan Vitek, not only for guiding me through this research journey, but also his kindness and support when it was most needed.

I thank my thesis committee, Amal Ahmed, Giuseppe Castagna, Arjun Guha, and Benjamin C. Pierce, for their helpful feedback, attention to detail, time, and flexibility.

I thank my first research advisor, Stanislav Mikhalkovich, for inviting me to programming languages research and supporting me.

I thank Ross Tate and Francesco Zappa Nardelli for their research mentorship.

I thank Vitaly Bragilevsky, Yana Demyanenko, and Jason Hemann for teaching with joy and making me feel that I belong.

I thank Matthias Felleisen for challenging and teaching me about teaching.

I thank Jane Kokernak and Mitch Wand for teaching me about writing.

I thank my labmates and friends, especially Ellen Arteca, Benjamin Chung, Aviral Goel, Celeste Hollenbeck, Alexi Turcotte, and Ming-Ho Yee, for making this journey fun and less stressful.

I thank Alexandra Silva for mentorship and support during the most challenging part of this journey.

I thank all graduate worker parents out there who made me believe that having a baby during a PhD is possible.

I thank Chelsea, Sherisse, Terry, and Yiliana for taking care of Sophia while I was writing this thesis.

CONTENTS

Acknowledgements	iv
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Subtyping in Julia	1
1.2 Thesis and Contributions	3
2 Background	5
2.1 Types	5
2.2 Multiple Dispatch	8
3 Julia Subtyping	11
3.1 Overview of Subtyping	11
3.2 Specification of Subtyping	14
3.3 Undecidability of Subtyping	23
4 Decidable Subtyping of Existential Types	27
4.1 Definition of Subtyping	29
4.1.1 Overview	30
4.1.2 Unification-Free Subtyping of Types	33
4.1.3 Constrained Subtyping of Types	35
4.1.4 Signature Subtyping	39
4.1.5 Validity of Types and Type Signatures	43
4.2 Properties of Subtyping	43
4.2.1 Decidability	44
4.2.2 Unification-Free Subtyping	49
4.2.3 Intersection	54
4.2.4 Constrained Subtyping	55
4.3 Extended Subtyping	58
4.3.1 Nominal subtyping	58
4.3.2 Diagonal Rule	62
5 Evaluation of Migration Effort	66
5.1 Methodology	67
5.2 Results	68
5.3 Migration Strategies	71
6 Related work	75
6.1 Decidability of subtyping	75
6.2 Semantic subtyping	77
6.3 Java wildcards	78
7 Conclusions and Future Work	80
Bibliography	81

A	Appendix	87
A.1	Properties of Subtyping	87
A.1.1	Decidable Subtyping	87
A.1.2	Unification-Free Subtyping	88
A.2	Semantic Model	91
A.2.1	Tuples and Unions	91
A.2.2	Invariant Constructors	91
A.2.3	Existential Types	92
A.3	Evaluation	95

LIST OF FIGURES

Figure 1	Several method definitions of generic function $(-)$ in Julia	6
Figure 2	Examples of type declarations: concrete (left) and abstract (right)	7
Figure 3	Function <code>myzero</code> , which dispatches on type values using <code>Type{T}</code>	8
Figure 4	A REPL session demonstrating method specialization	9
Figure 5	Syntax of Julia types	12
Figure 6	Julia subtyping (excerpt from [Zappa Nardelli et al. 2018])	16
Figure 7	Encoding of F_{\leq}^N types in Julia	23
Figure 8	Grammar of type signatures and types	29
Figure 9	Auxiliary definitions	32
Figure 10	Subtyping of types (free from unification variables)	34
Figure 11	Constrained subtyping of types	36
Figure 12	Intersection (\sqcap_{Γ}) of types	39
Figure 13	Subtyping of type signatures	40
Figure 14	Constraints resolution algorithm $\mathbf{Solve}(\Gamma; \Delta; K)$	41
Figure 15	Validity of types and type signatures	42
Figure 16	Measure of types and type signatures	44
Figure 17	Syntactic size	45
Figure 18	Occurrence of a variable	48
Figure 19	Validity of substitution with respect to environment and constraints	54
Figure 20	Examples of inheritance in type declarations	58
Figure 21	Julia subtyping: nominal types	60
Figure 22	Additional subtyping rules to support inheritance	61
Figure 23	Variable discharge algorithm $\mathbf{Discharge}(\Gamma; l <: X <: u; K)$	61
Figure 24	Validity of type declarations and types in the presence of inheritance (extends Figure 8 and Figure 15)	63
Figure 25	Type concreteness	64
Figure 26	Subtyping with concrete variables	65
Figure 27	Type annotations flagged by the analysis: false positives (left) and true positive (right)	69

Figure 28	Muon.jl code fragment with unrepresentable types	73
Figure 29	Alicorn.jl code fragment with unrepresentable type	73
Figure 30	UnitfulEquivalences.jl code fragment with unrepresentable type (could not be migrated easily)	74
Figure 31	Semantic interpretation and decidable subtyping for simple types	92
Figure 32	Semantic interpretation and decidable subtyping for invariant constructors	93
Figure 33	Semantic interpretation for existential types	94
Figure 34	Test cases for the analysis of type annotations	95
Figure 35	An example of a parsing error	95
Figure 36	Type annotation processing errors	96

LIST OF TABLES

Table 1	Open/closed issues on the Julia bug tracker (March 2023)	3
Table 2	Corpus of Julia packages from General registry (May 2023)	67

INTRODUCTION

Julia is a dynamic, high-level, high-performance programming language [Bezanson et al. 2017], originally designed for scientific computing. At the core of Julia’s design is multiple dynamic dispatch, which relies on an expressive language of type annotations to tailor function implementations to different argument types. The type language includes set-theoretic unions, distributive tuples, parametric invariant types, and impredicative existential types. Multiple dispatch is resolved at run time with a complex subtyping algorithm. However, subtyping in Julia is undecidable. In practice, when an undecidable subtyping check is encountered, program execution fails with an uninformative `StackOverflowError`.

In this dissertation, I propose a decidable subtype relation for a restricted language of Julia types. Namely, the restriction limits existential types inside invariant constructors to existential types expressible with use-site variance, similar to Java wildcards. In a corpus of 9K registered Julia packages, 99.99% of 2M statically identifiable type annotations satisfy the proposed restriction. Thus, the restriction provides a viable approach for evolving the Julia language towards decidable subtyping.

1.1 SUBTYPING IN JULIA

To encourage a high level of code reuse and extensibility, Julia is designed around *symmetric multiple dynamic dispatch*. Multiple dispatch allows a function to have multiple implementations, called methods, tailored to different argument types; at run time, a function call is dispatched to the most specific method for the given arguments.

To deliver performance, Julia relies on a type-specializing just-in-time (JIT) compiler: with few exceptions¹, every time a method is called with a new set of argument types, it is specialized for those types, creating a new method definition [Pelenitsyn et al. 2021]. Therefore, even for functions with a single user-defined method, more

¹ <https://docs.julialang.org/en/v1/manual/performance-tips/#Be-aware-of-when-Julia-avoids-specializing>

methods can accrue during the program execution, adding work for the dispatch mechanism.

Multiple dispatch is resolved with the use of a subtype relation. Namely, for every function call, subtyping is checked between the argument types and method signatures, as well as between method signatures, to pick the most specific applicable method. While most run-time value types are nominal, the language of type annotations is more complex and expressive. In particular, inspired by semantic subtyping where types are interpreted as sets, Julia supports covariant tuples that distribute over unions. For example, according to the semantic approach, the type `Tuple{String, Union{Int64, UInt64}}` represents a set of binary tuples where the first component is a string, and the second component is either a signed or unsigned integer. Therefore, both `Tuple{String, Int64}` and `Tuple{String, UInt64}` are subtypes of the above type, and that type is equivalent to the union of the two tuples, `Union{Tuple{String, Int64}, Tuple{String, UInt64}}`. In addition to finite unions, Julia supports existential types, referred to as union-all types or iterated unions in the language. For example, `Vector{T}` where `T` represents an infinite union of vectors `Vector{t}` for all instantiations `t` of the type variable `T`. A more detailed account of Julia types and subtyping is provided in Chapter 2.

Despite intentional simplifications in the type language, e.g. the lack of recursive bounds² on type variables, subtyping in Julia is *undecidable* [Chung 2023]. Although relying on undecidable subtyping is not unprecedented for a statically typed language (see Scala [Hu and Lhoták 2019] for an example), the price of undecidability is higher in Julia, since it can manifest at almost any point during program execution. In particular, the run-time system relies on undecidable subtyping to resolve function calls, process new method definitions [Belyakova et al. 2020], manipulate data (e.g. when adding an element to a container), as well as JIT compile methods [Pelenitsyn et al. 2021]. In practice, the undecidability leads to a run-time crash with a `StackOverflowError`. Such an issue can be particularly hard to debug, because the error is uninformative: neither the problematic subtyping query nor stack trace is reported.

A number of issues related to subtyping have been reported on the Julia bug tracker. For example, #41948³ reports a `StackOverflowError` caused by a function definition, which is likely linked to undecidability; #33137⁴ points out an inconsistency in subtyping; and #24166⁵ (now fixed) reports a problem with reflexivity and transitivity. Overall, there are 105 open/704 closed issues labeled with “types and dispatch”

² For example, `X implements Comparable<X>` in Java.

³ <https://github.com/JuliaLang/julia/issues/41948>

⁴ <https://github.com/JuliaLang/julia/issues/33137>

⁵ <https://github.com/JuliaLang/julia/issues/24166>

Table 1: Open/closed issues on the Julia bug tracker (March 2023)

	types and dispatch	codegen	GC	macros	<any label>
<any label>	92/414	56/246	24/51	27/36	3551/19238
bug	13/138	8/86	1/15	5/11	226/2664

Axes represent labels the issues are marked with: for example, the cell 24/51 in the row <any label> and column GC corresponds to all issues labeled with “GC”, whereas 1/15 in the row *bug* and column GC corresponds to “GC” issues that are also labeled with “bug”.

as of March 2023, with 13 open/138 closed being also labeled with “bug” (not every issue is properly labeled as a bug, e.g. the aforementioned #24166 is not). Table 1 provides a few more data points for comparison: for example, there are 8 open/86 closed “codegen” bugs and 1 open/15 closed “GC” bugs. Thus, type-related concerns, including the undecidability of subtyping, constitute a non-negligible portion of problems in the Julia implementation.

1.2 THESIS AND CONTRIBUTIONS

Due to Julia’s ubiquitous use of subtyping at run time, the undecidability of subtyping is consequential. It is unlikely, as I discuss in Section 3.3, that a compatible⁶ decidable subtype relation exists for the entirety of Julia types. However, my thesis is that

the Julia language can be evolved to provide for decidable subtyping while requiring minimal effort for migrating existing code.

Because the type language and its associated subtype relation impact the set of valid Julia programs and their dynamic semantics, replacing subtyping can have profound effects on programmer experience as well as the ability to migrate existing code. Thus, not every decidable subtyping would be a viable candidate for Julia’s evolution.

In this dissertation, I show that there is a restriction on Julia’s type language that allows for decidable subtyping and supports 99.99% of statically identifiable type annotations in registered Julia packages. Namely, the restriction limits existential types inside invariant constructors to existential types expressible with use-site variance, similar to Java wildcards [Torgersen et al. 2004].

The dissertation provides an overview of Julia (Chapter 2 and Section 3.1) and makes the following contributions:

- **Specification of Julia subtyping** [Zappa Nardelli, Belyakova, Pelenitsyn, Chung, Bezanson, and Vitek 2018] (Section 3.2). The

⁶ *Compatible* means *compatible with the current Julia subtyping*, i.e., produces the same answer as Julia subtyping whenever Julia subtyping terminates without an error.

specification covers most of Julia, with the only exception being variable-length tuples.

- **Decidable subtyping** for a restricted language of Julia types [Belyakova, Chung, Tate, and Vitek 2023] (Chapter 4). The restriction retains most of Julia types, limiting only existential types inside invariant constructors.
- **Evaluation of the migration effort** required to switch to the restricted type language (Chapter 5). In the corpus of all 9K registered Julia packages, 99.99% of source code type annotations satisfy the restriction.

Furthermore, Appendix A.2 discusses a semantic interpretation of types, extending [Belyakova 2019].

Related to this dissertation, I worked on formalizing several aspects of the Julia language:

- [Belyakova, Chung, Gelinas, Nash, Tate, and Vitek 2020] presents world-age semantics, which enables dispatch optimization in the presence of eval by delaying visibility of dynamically generated methods.
- [Pelenitsyn, Belyakova, Chung, Tate, and Vitek 2021] defines type stability—the property which allows for efficient compilation of code with multiple dispatch.

BACKGROUND

Julia is a high-level, dynamic programming language, which was originally designed for scientific computing [Bezanson et al. 2017] and released in 2012. It aims to solve the so-called “two-language problem” by providing both good performance and productivity features [Bezanson et al. 2018]. For performance, Julia relies on an optimizing, type-specializing JIT compiler. For productivity, the language provides garbage collection, dynamic typing, and multiple dynamic dispatch.

Multiple dynamic dispatch [Bobrow et al. 1986; Chambers 1992] is at the core of the Julia language. The dispatch mechanism allows a function, called a *generic function*, to have multiple implementations, called *methods*, that are tailored to different argument types. For example, Figure 1 shows several method definitions of the function `(-)`, which is used as both unary negation (lines 1 and 5) and binary subtraction (lines 2–4).

Unlike static method overloading (which is resolved at compile time, using static types of the arguments), multiple dynamic dispatch is resolved at *run time*: every function call in the program is dispatched to the *most specific applicable* method based on the run-time types of the arguments. If such a single best method does not exist, an error is raised. Section 2.2 describes this process in more detail, but for now, it suffices to know that dispatch resolution relies on **subtyping**. Notably, Julia does not allow the user to call a method of a generic function directly: the only way to reach a method is via the dispatch mechanism. Thus, whenever a method is called, its arguments are guaranteed to be subtypes of the declared method signature types.

2.1 TYPES

The expressive power of Julia’s multiple dispatch stems from its unique **language of type annotations**, as demonstrated in Figure 1. Briefly, the type language supports:

- the top type `Any`, which is a supertype of all types;
- nominal non-parametric types, e.g. `BigInt` and `Number`;

```

1 - (x::BigInt) = MPZ.neg(x)
2 - (x::BigInt, y::BigInt) = MPZ.sub(x, y)
3 - (x::T, y::T) where T<:Union{Int16, Int32, ..., UInt128} = sub_int(x, y)
4 - (m::Missing, n::Number) = missing
5 - (A::AbstractArray{T,N}) where {T,N} = broadcast_preserving_zero_d(-, A)

```

Figure 1: Several method definitions of generic function `(-)` in Julia. Methods in lines 1–2 call library functions. Method in line 3 calls a built-in function. Method in line 4 returns a singleton value `missing` of the `Missing` type. Method in line 5 calls a higher-order function that applies `(-)` to the elements of the array argument `A`.

- nominal parametric types, which are invariant in the type parameters, e.g. `AbstractArray{T,N}`¹ in line 5;
- set-theoretic union types, e.g. `Union{Int16, ..., UInt128}` in line 3; the empty `Union{}` is the bottom type, i.e. a subtype of all types;
- covariant tuple types, e.g. `Tuple{String, Number}`;
- so-called union-all types, written with `where` (lines 3 and 5): intuitively, a union-all type `t where T` represents a union of types `t[t'/T]` for all valid instantiations `t'` of the type variable `T`.

Nominal types are induced by user-defined datatype declarations and constitute a single-parent inheritance hierarchy. Julia’s support for inheritance is limited compared to mainstream object-oriented languages such as C# and Java: only *abstract* types, which cannot be used to construct values, can be inherited, i.e., used as declared supertypes of other nominal types. Datatypes that are used to construct values are referred to as *concrete*. Concrete nominal types include primitive types such as `Int128`, and composite `struct` types (either mutable or immutable). Figure 2 provides several examples of user-defined concrete (on the left) and abstract (on the right) types. Parametric types (both abstract and concrete) can declare non-recursive lower and upper bounds on type variables; for example, type `Rational{T}` requires `T` to be a subtype of abstract `Integer`. Supertypes in type declarations are declared to the right of `<:` and cannot refer to the type being declared. For example, `Int128` is a subtype of `Signed`, and `Signed` is a subtype of `Integer`. If the supertype declaration is omitted, like in `AbstractSet{T}`, the supertype defaults to `Any`.

The distinction between *concrete* and *abstract* types is important to Julia’s runtime. *Concrete* types are run-time types of values, which can

¹ Julia allows unbounded type variables such as the dimensionality parameter `N` of `AbstractArray{T,N}` to be instantiated with “plain bits” values such as numbers or booleans, for which `isbits` function returns `true`. Thus, `AbstractArray{Int64,1}` is a valid nominal type. For value arguments, the invariance check succeeds if they are bitwise equal.

```

1 primitive type Int128 <: Signed 128
2 end
3
4 struct Rational{T<:Integer} <: Real
5     num::T
6     den::T
7 end
8
9 mutable struct
10     BitSet <: AbstractSet{Int}
11     ...
12 end

```

```

1 abstract type Signed <: Integer
2 end
3
4 abstract type AbstractSet{T}
5 end

```

Figure 2: Examples of type declarations: concrete (left) and abstract (right)

`Int128` is a primitive, 128-bit type. `Rational` is an immutable parametric composite type. `BitSet` is a mutable composite type. `Signed` is an abstract type. `AbstractSet` is an abstract parametric type.

be obtained with the `typeof` operator. Because concrete types are final, i.e., cannot be inherited, the compiler knows their size and memory layout and can thus optimize code efficiently. In Julia, concrete types include concrete nominal types and their tuples. *Abstract* types, on the other hand, are widely used as type annotations in method definitions. They also account for heterogeneous data when used as field type annotations or type arguments of parametric constructors, e.g. in `Vector{Any}`. Values of abstract types are stored as references and are not conducive to optimizations. Besides abstract nominal types, abstract types include union types such as `Union{Int16, ..., UInt128}`, and union-all types such as `Vector{T}` where `T` (a shorthand for this type is simply `Vector`). Note, however, that every particular instantiation of `Vector`, e.g. `Vector{Number}`, is a concrete type regardless of the concreteness of the type argument. The bottom type represented with the empty union, `Union{}`, is neither concrete nor abstract: it has no values and is a subtype of all types.

It is worth noting that the Julia language does not have a structural function type, which would typically be written as $\tau \rightarrow s$. As mentioned above, all function calls are dynamically dispatched to a method of that generic function, and there is no way to directly call a particular method or pass it as an argument. However, generic functions are first-class values and can be used as method arguments: for example, `(-)` is passed to a function call in line 5 of Figure 1. Every generic function `f` has the concrete type `typeof(f)`, which is a subtype of the type of all functions `Function`.

In Julia, types are also values: they can be stored in variables, passed as function arguments, and introspected. Depending on the kind of a type, the `typeof` operator returns one of four values: `DataType` for


```

1 myzero(::Type{Int})           = 0
2 myzero(::Type{Float64})       = 0.0
3 myzero(::Type{Vector{T}}) where T = Vector{T}()
4
5 myzero(Int)                   # returns 0
6 myzero(Vector{Bool})          # returns Bool[]

```

Figure 3: Function `myzero`, which dispatches on type values using `Type{T}`

fully instantiated nominal types and tuples, `Union` for finite union types, `UnionAll` for union-all (where) types, and `Core.TypeofBottom` for the bottom type `Union{}`. There is also a special abstract parametric type `Type{T}` such that `t` is considered an instance² of `Type{t}`: as exemplified by Figure 3, `Type{T}` facilitates multiple dispatch on type values.

2.2 MULTIPLE DISPATCH

Argument type annotations are optional: annotations are the key to defining multiple distinct methods, but if the user omits some or all annotations in a method, those annotations default to `Any`—a supertype of all types. For example, in the first `sub5` definition below, argument `x` does not have an explicit type annotation:

```

1 sub5(x) = x - 5           # same as sub5(x::Any) = x - 5
2 sub5(x::String) = x * "-5"

```

If `sub5` is called with a string, the call will dispatch to the method definition in line 2 and always succeed. If `sub5` is called with anything else, the call will dispatch to the method definition in line 1. In the latter case, the execution will succeed as long as there is an implementation of `(-)` for the type of `x` and `Int64`, which is the type of `5`. For example, `sub5(10.0)` evaluates to `5.0`, whereas `sub5([])` fails:

```

1 julia> sub5([])
2 ERROR: MethodError: no method matching -(::Vector{Any}, ::Int64)

```

To achieve good performance, Julia generates method instances specialized to concrete argument types the function was called with [Pelenitsyn et al. 2021]. Thus, next time a function is called with the same argument types, method dispatch will have more definitions to choose from. Figure 4 shows an example REPL session for `sub5`.

² `v isa t` checks if value `v` is an instance of type `t`

```

1 julia> sub5(x) = x - 5
2 julia> sub5(x::String) = x * "-5"
3
4 julia> methods(sub5)
5 # 2 methods for generic function "sub5":
6 [1] sub5(x::String) in Main at REPL[2]:1
7 [2] sub5(x) in Main at REPL[1]:1
8
9 julia> methods(sub5)[2].specializations
10 svec()
11
12 julia> sub5(10.0)
13 5.0
14
15 julia> methods(sub5)
16 # 2 methods for generic function "sub5":
17 [1] sub5(x::String) in Main at REPL[2]:1
18 [2] sub5(x) in Main at REPL[1]:1
19
20 julia> methods(sub5)[2].specializations
21 svec(MethodInstance for sub5(::Float64), nothing, ...)

```

Figure 4: A REPL session demonstrating method specialization

Function `methods` lists all methods of a generic function. Field `specializations` of a method stores compiled method specializations.

Method resolution for a dispatched function call $f(a_1, a_2, \dots)$ relies on tuple subtyping [Leavens and Millstein 1998] between run-time argument types and method signatures. Namely, argument types are combined into a tuple of concrete types

$$\sigma = \text{Tuple}\{\text{typeof}(a_1), \text{typeof}(a_2), \dots\},$$

and every method signature is either a tuple of declared argument types if the method definition is not parametric, or a union-all of such tuple if the definition is parametric. For example, consider Figure 1: method in line 1 is represented by the signature `Tuple{BigInt}`, line 4 by `Tuple{Missing, Number}`, and line 3 by `Tuple{T, T}` where $T <: \text{Union}\{\dots\}$. Because tuple subtyping gives all elements equal consideration, multiple dispatch in Julia is *symmetric*.

Dispatch resolution, if successful, returns the *most specific applicable method* for the given arguments. Namely, for a call to a generic function f with a concrete argument type σ , the process can be described as follows:

1. Find all applicable methods, i.e., all method signatures τ of the function f that are supertypes of the argument type $\sigma <: \tau$. If no methods are applicable, a method-not-found error is raised.
2. Among the applicable methods $\{\tau_1, \dots, \tau_n\}$, pick the method with the most specific type signature, i.e., τ_k such that $\forall i \in$

$1..n, \tau_k <: \tau_i$. If there is no such single best method, a method-is-ambiguous error is raised. Requested by language users over the years, Julia has additional specificity rules to resolve ambiguities in some cases, but those are not relevant to this work.

Thus, subtyping is an integral part of Julia's *dynamic semantics*, and due to step 2, it is used with arbitrarily complex types.

JULIA SUBTYPING

In this chapter, I provide a detailed account of Julia subtyping (Sections 3.1 and 3.2) and discuss its undecidability (Section 3.3).

3.1 OVERVIEW OF SUBTYPING

The syntax of Julia types is given in Figure 5.

Subtyping in Julia largely follows the combination of **nominal subtyping** for user-defined nominal types and **semantic subtyping** for covariant tuple and union types. For example, for datatype declarations from Figure 2, `Int128` is a subtype of `Signed` and, transitively, of `Integer`; `BitSet` is a subtype of `AbstractSet{Int}`. A union type `Union{t1, t2, ...}` describes a set-theoretic union of types `t1`, `t2`, `...`; for example, `Int` is a subtype of `Union{Signed, String}`, and `Union{t1, t2, ...} <: t` if all components `t1 <: t`, `t2 <: t`, `...`. Tuples in Julia are immutable, and tuple types are covariant: `Tuple{t1, t2, ...}` is a subtype of `Tuple{t1', t2', ...}` if their corresponding components are subtypes, i.e., `t1 <: t1'`, `t2 <: t2'`, `...`. Following semantic subtyping [Frisch et al. 2008] where types are interpreted as sets, tuple types distribute over unions, so types `Tuple{Union{Int, String}}` and `Union{Tuple{Int}, Tuple{String}}` are equivalent.

User-defined parametric datatypes are invariant in the type parameter regardless of whether the datatype is mutable or immutable, meaning that `Name{t1, t2, ...}` is a subtype of `Name{t1', t2', ...}` only if the corresponding type arguments are equivalent, i.e., `t1 <:= t1'`, `t2 <:= t2'`, `...`. Thus, the invariant type `Rational{Int}` is *not* a subtype of `Rational{Signed}`.

Abstract union-all types `t where l <: T <: u` are better known in the literature as **bounded existential types**, which also model Java wildcards¹ [Torgersen et al. 2004]; I will call them as such in the remainder of the dissertation. If lower (upper) bound on the type variable is omitted, it defaults to the bottom type `Union{}` (top type `Any`). Intuitively, an existential type denotes a union of `t[t'/T]` for all instantiations of the

¹ In Julia syntax, a Java wildcard type `Foo<?>` can be written as `Foo{T} where T or Foo{<:Any}`.

$t ::=$		
	<code>Any</code>	top
	<code>Union</code> $\{t_1, \dots, t_n\}$	union
	<code>Tuple</code> $\{t_1, \dots, t_n\}$	tuple
	<code>name</code> $\{t_1, \dots, t_n\}$	user-defined type
	<code>t where l <: T <: u</code>	union-all
	<code>T</code>	type variable

Figure 5: Syntax of Julia types

type variable T such that $l <: t' <: u$. Similarly to subtyping of finite union types, the intent is that:

- $(t \text{ where } l <: T <: u) <: t_2$ if, for all valid instantiations $l <: t' <: u$, it holds that $t[t'/T] <: t_2$, and
- $t_1 <: (t \text{ where } l <: T <: u)$ if there exists at least one $l <: t' <: u$ such that $t_1 <: t[t'/T]$.

For example, `Vector{Int}` is a subtype of `Vector{T} where T <: Integer` because T can be instantiated with `Int`, and `Vector{T} where T <: Integer` is a subtype of `Vector{S} where S` because for all valid instantiations t' of T , type variable S can be instantiated with the same type t' . Just like with unions, tuples distribute over existential types: for example, types `Tuple{Vector{T} where T}` and `Tuple{Vector{T}} where T` are equivalent.

Existential types in Julia are *impredicative*: existential quantifiers can appear anywhere in a type, and type variables can be instantiated with arbitrary existential types. For example, type `Vector{Matrix{T} where T}` denotes a vector of matrices with arbitrary element types. In contrast, `Vector{Matrix{S}} where S` denotes a set of vectors where elements are matrices with the same element type. Thus, a vector of integer matrices `Vector{Matrix{Int}}` is a subtype of the latter—existential—type, because S can be instantiated with `Int`. But it is not a subtype of the former—invariant parametric—type `Vector{Matrix{T} where T}`, because type arguments `Matrix{Int}` and `Matrix{T} where T` are not equivalent. Note how type arguments of invariant type constructors such as `Vector` can be arbitrarily complex: because of that, subtyping between two concrete types is no easier than subtyping between arbitrary types.

Existential types serve at least two distinct purposes in the Julia language. First, parametric types instantiated with existential type arguments, such as

`Vector{Matrix{T} where T},`

are useful for representing heterogeneous data. Second, top-level existential types, such as

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

represent type signatures of parametric method definitions. It may be surprising that for method signatures, Julia uses existential rather than universal types, but recall that the primary purpose of types is to serve multiple dispatch. In Julia, it is impossible to directly invoke a parametric method definition and provide it with a type argument. Instead, the method is being dispatched to if subtyping for the corresponding existential type succeeds. Then, in the body of the method, the existential type is implicitly unpacked, with the witness type being some valid instantiation induced by subtyping. Consider the following code snippet as an example:

```

1  f(v :: Vector{T}) where T =
2      Set{T}(v)
3
4  f([5, 7, 5]) # returns Set{Int} with 2 elements: 5, 7

```

Because `[5, 7, 5]` is a `Vector{Int}`, and `Tuple{Vector{Int}}` is a subtype of the existential `Tuple{Vector{T}} where T`, as witnessed by the instantiation of `T` with `Int`, the call `f([5, 7, 5])` dispatches to the method in line 1, and `T` in the body of the method becomes `Int`. However, when multiple instantiations of the variable are possible, Julia sometimes gives up on assigning the witness type. In the example below, subtyping succeeds for the call `g(true)`, because `Tuple{Bool}` is a subtype of `Tuple{T} where T>:Int`: there are, in fact, multiple possible instantiations `t` of `T` such that `Tuple{Bool} <: Tuple{t}`, e.g. `Any` or `Union{Int, Bool}`. But rather than pick one instantiation, Julia throws an error as soon as the program tries to access type variable `T`:

```

1  > g(x::T) where T>:Int = begin
2      println(x)
3      println(T)
4  end
5
6  > g(true)
7  true
8  ERROR: UndefVarError: T not defined

```

On the other hand, for the call `g(5)`, `T` is assigned the smallest possible type, `Int`:

```

1  > g(5)
2  5
3  Int

```

Subtyping of existential types includes a special case, called the **diagonal rule**, which provides the support for a generic programming pattern where method arguments are expected to be of the same concrete type. Consider the following method definition, which defines equality (`==`) in terms of the built-in equality of bit representations (`===`):

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

If it were possible to instantiate τ with an abstract type such as `Integer`, the method could be called with a pair of a signed and unsigned integer. This would be an incorrect implementation of equality, for the same bit representation corresponds to different numbers when interpreted with and without the sign. To prevent such behavior, the diagonal rule states: if a type variable appears in the type (1) only covariantly and (2) more than once, it can be instantiated only with concrete types.² Thus, the type signature of (`==`) above, `Tuple{T, T} where T<:Number`, represents a concretely instantiatable existential type: it is a union of tuples `Tuple{t, t}` where `t` is a concrete subtype of `Number`. The same rule applies in line 3 of Figure 1: built-in integer subtraction `sub_int` is guaranteed to be called only with primitive integers of the same concrete type.

3.2 SPECIFICATION OF SUBTYPING

For Julia programmers, reasoning about the subtype relation is necessary to understand and correctly use multiple dynamic dispatch. However, the language documentation does not provide a specification of subtyping: instead, subtyping is mentioned in multiple sections related to type constructors and multiple dispatch. Initially, an incomplete (and soon outdated) definition of subtyping existed only in [Bezanson 2015], with the only reference point being the actual implementation of subtyping (~3000 lines of heavily optimized C code as of 2018).

To build intuition about Julia subtyping, this section presents a fragment of the specification of Julia 0.6.2 subtyping from [Zappa Nardelli et al. 2018]. In that work, my collaborators and I defined, implemented, and empirically validated a subtype relation for Julia's type language, thus providing the first complete specification of Julia subtyping. The only omitted feature is the `Vararg{T, N}` construct, which can be used as the last parameter of a tuple to denote `N` trailing arguments of type `T`. Having reconstructed the subtype relation,

² A similar rule applies to static resolution of method overloading in C#. An example can be found on this page: <https://fzn.fr/projects/lambdajulia/diagonalcsharp.pdf>

we were able to identify multiple issues such as counterexamples to reflexivity and transitivity, and suspected the undecidability of subtyping, which was later proved by Chung [2023]. Most of the bugs we reported have been fixed by Julia maintainers. Furthermore, the treatment of union types in the diagonal rule has changed based on our proposal.

The subtyping fragment is presented in Figure 6. It includes features relevant to the undecidability of subtyping, namely: unions, tuples, invariant constructors, and existential types. Following the paper, types are written as in the Julia language: for example, existential types are represented with `where`. “Plain bits” values, nominal subtyping, subtyping of `Type{t}`, and the diagonal rule³ are omitted, but they can be found in [Zappa Nardelli et al. 2018].

In Figure 6, subtyping is defined in the form of a judgment

$$E \vdash t <: t' \vdash E'$$

which should be read as: *in the environment E , type t is a subtype of t' , with updated constraints E' . E is a type variable environment that contains two kinds of variables: forall (also called left, introduced by existential types on the left and recorded as $^L T$) and exist (also called right, introduced by existential types on the right and recorded as $^R T$). Variables are recorded with their declared lower and upper bounds l and u as T_l^u ; bounds on exist (right) variables can get tighter during subtype checking, which is reflected in the updated environment E' . For example, consider the following judgment:*

$$\text{Tuple}\{\text{Int}\} <: \text{Tuple}\{T\} \text{ where } \text{Union}\{ \} <: T <: \text{Any}.$$

First, the right variable T is introduced into the environment with its declared bounds `Union{}` and `Any`:

$$^R T_{\text{Union}\{ \}}^{\text{Any}} \vdash \text{Tuple}\{\text{Int}\} <: \text{Tuple}\{T\} \vdash ?$$

Next, since tuples are covariant, it should be the case that `Int` is a subtype of T . Recall the intuition that for existential types on the right, subtyping holds if there exists a valid instantiation of the variable that would satisfy the judgment. In the example, subtyping `Int <: T` holds if the variable T is instantiated with a type that is a supertype of `Int`. Therefore, this very constraint is recorded in an updated environment as $^R T_{\text{Int}}^{\text{Any}}$ (note the larger lower bound):

$$^R T_{\text{Union}\{ \}}^{\text{Any}} \vdash \text{Int} <: T \vdash ^R T_{\text{Int}}^{\text{Any}},$$

³ The treatment of union types in the diagonal rule has changed since the paper was published, but the paper discusses the issue.

$$\boxed{E \vdash t <: t \vdash E}$$

$$\begin{array}{c}
\text{TOP} \\
\hline
E \vdash t <: \text{Any} \vdash E
\end{array}
\qquad
\begin{array}{c}
\text{REFL} \\
\hline
E \vdash T <: T \vdash E
\end{array}$$

$$\begin{array}{c}
\text{TUPLE} \\
\hline
\frac{E \vdash t_1 <: t'_1 \vdash E_1 \quad \dots \quad E_{n-1} \vdash t_n <: t'_n \vdash E_n}{E \vdash \text{Tuple}\{t_1, \dots, t_n\} <: \text{Tuple}\{t'_1, \dots, t'_n\} \vdash E_n}
\end{array}$$

$$\begin{array}{c}
\text{TUPLE_LIFT_UNION} \\
\hline
\frac{t' = \text{lift_union}(\text{Tuple}\{t_1, \dots, t_n\}) \quad E \vdash t' <: t \vdash E'}{E \vdash \text{Tuple}\{t_1, \dots, t_n\} <: t' \vdash E'}
\end{array}$$

$$\begin{array}{c}
\text{TUPLE_UNLIFT_UNION} \\
\hline
\frac{t' = \text{unlift_union}(\text{Union}\{t_1, \dots, t_n\}) \quad E \vdash t' <: t \vdash E'}{E \vdash t <: \text{Union}\{t_1, \dots, t_n\} \vdash E'}
\end{array}$$

$$\begin{array}{c}
\text{UNION_LEFT} \\
\hline
\frac{E \vdash t_1 <: t \vdash E_1 \quad \dots \quad E_{n-1} \vdash t_n <: t \vdash E_n}{E \vdash \text{Union}\{t_1, \dots, t_n\} <: t \vdash E_n}
\end{array}
\qquad
\begin{array}{c}
\text{UNION_RIGHT} \\
\hline
\frac{\exists j. E \vdash t <: t_j \vdash E'}{E \vdash t <: \text{Union}\{t_1, \dots, t_n\} \vdash E'}$$

$$\begin{array}{c}
\text{APP_INV} \\
\hline
\frac{E_0 = \text{add}(E, \text{Barrier}) \quad \forall 0 < i \leq n. E_{i-1} \vdash t_i <: t'_i \vdash E'_i \wedge E'_i \vdash t'_i <: t_i \vdash E_i}{E \vdash \text{name}\{t_1, \dots, t_n\} <: \text{name}\{t'_1, \dots, t'_n\} \vdash \text{del}(\text{Barrier}, E_n)}
\end{array}$$

$$\begin{array}{c}
\text{L_INTRO} \\
\hline
\frac{\text{add}({}^L T_l^u, E) \vdash t <: t' \vdash E'}{E \vdash (t \text{ where } l <: T <: u) <: t' \vdash \text{del}(T, E')}
\end{array}$$

$$\begin{array}{c}
\text{R_INTRO} \\
\hline
\frac{\text{add}({}^R T_l^u, E) \vdash t <: t' \vdash E' \quad \text{consistent}(T, E')}{E \vdash t <: (t' \text{ where } l <: T <: u) \vdash \text{del}(T, E')}
\end{array}$$

$$\begin{array}{c}
\text{L_LEFT} \\
\hline
\frac{\text{search}(T, E) = {}^L T_l^u \quad E \vdash u <: t \vdash E'}{E \vdash T <: t \vdash E'}
\end{array}
\qquad
\begin{array}{c}
\text{L_RIGHT} \\
\hline
\frac{\text{search}(T, E) = {}^L T_l^u \quad E \vdash t <: l \vdash E'}{E \vdash t <: T \vdash E'}$$

$$\begin{array}{c}
\text{R_LEFT} \\
\hline
\frac{\text{search}(T, E) = {}^R T_l^u \quad E \vdash l <: t \vdash E'}{E \vdash T <: t \vdash \text{upd}({}^R T_l^f, E')}
\end{array}$$

$$\begin{array}{c}
\text{R_RIGHT} \\
\hline
\frac{\text{search}(T, E) = {}^R T_l^u \quad (\text{is_var}(t) \wedge \text{search}(t, E) = {}^L S_l^u) \implies \neg \text{outside}(T, S, E) \quad E \vdash t <: u \vdash E'}{E \vdash t <: T \vdash \text{upd}({}^R T_{\text{Union}\{l, t\}}^u, E')}
\end{array}$$

$$\begin{array}{c}
\text{R_L} \\
\hline
\frac{\text{search}(T_1, E) = {}^R T_{1l_1}^{u_1} \quad \text{search}(T_2, E) = {}^L T_{2l_2}^{u_2} \quad \text{outside}(T_1, T_2, E) \implies E \vdash u_2 <: l_2 \vdash E' \quad E \vdash u_1 <: l_2 \vdash E''}{E \vdash T_1 <: T_2 \vdash \text{upd}({}^R T_{\text{Union}\{T_1, l_1\}}^{u_1}, E')}
\end{array}$$

Figure 6: Julia subtyping (excerpt from [Zappa Nardelli et al. 2018])

and the new environment is propagated back to the judgment for tuples:

$$^R T_{\text{Union}\{\}}^{\text{Any}} \vdash \text{Tuple}\{\text{Int}\} <: \text{Tuple}\{T\} \vdash ^R T_{\text{Int}}^{\text{Any}}.$$

With the exception of `TUPLE_LIFT_UNION` and `TUPLE_UNLIFT_UNION`, which are discussed later, the rules for subtyping tuples, unions, and invariant constructors are mostly straightforward. The rule `TUPLE` checks covariant subtyping of the tuple elements. The 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; furthermore, the rule `UNION_LEFT` allows for deriving that `Union{}` is a subtype of all types, because its hypothesis is trivially validated by the forall quantification over an empty set. In the rule `APP_INV` for invariant constructors, the equality of corresponding type arguments t_i and t'_i is ensured by checking both $t_i <: t'_i$ and $t'_i <: t_i$, referred to as left-to-right and right-to-left checks, respectively; the Barrier construct is discussed later.

Because the same right variable can appear in multiple positions, environment updates E_i^l need to be propagated across all the elements of a tuple/union/invariant constructor. For instance, consider the following example:

$$^R T_{\text{Union}\{\}}^{\text{Any}} \vdash \text{Tuple}\{\text{Any}, \text{Ref}\{\text{Int}\}\} \not<: \text{Tuple}\{T, \text{Ref}\{T\}\}.$$

After the first recursive call to subtyping,

$$^R T_{\text{Union}\{\}}^{\text{Any}} \vdash \text{Any} <: T \vdash ^R T_{\text{Any}}^{\text{Any}}$$

the variable T is known to be a supertype of `Any`. Thus, subtyping between `Ref{Int}` and `Ref{T}` is invoked in the updated environment:

$$^R T_{\text{Any}}^{\text{Any}} \vdash \text{Ref}\{\text{Int}\} \not<: \text{Ref}\{T\}.$$

The left-to-right check succeeds, but the right-to-left check fails because the lower bound of T —`Any`—is not a subtype of `Int`:

$$^R T_{\text{Any}}^{\text{Any}} \vdash \text{Int} <: T \vdash ^R T_{\text{Any}}^{\text{Any}} \quad ^R T_{\text{Any}}^{\text{Any}} \vdash T \not<: \text{Int}.$$

Without environment propagation, the constraint $\text{Any} <: T$ would have been lost and the subtyping check would have succeeded.

LEFT AND RIGHT VARIABLES. Left and right variables are treated differently by the subtyping algorithm.

- Left variables never change in the environment, and subtyping should hold with respect to their declared bounds. Thus, if a

left variable T appears on the left of $<$, then the judgment can be satisfied only if the upper bound of T is smaller than t :

$${}^L T_l^u \vdash T <: t \vdash {}^L T_l^u \quad \text{only if} \quad {}^L T_l^u \vdash u <: t \vdash {}^L T_l^u.$$

If ${}^L T$ appears on the right, then it is the lower bound of T that must be a supertype of t :

$${}^L T_l^u \vdash t <: T \vdash {}^L T_l^u \quad \text{only if} \quad {}^L T_l^u \vdash t <: l \vdash {}^L T_l^u.$$

This corresponds to the intuition that for a left-hand side existential type, subtyping should hold for all possible instantiations of the type variable.

- Right variables, on the other hand, may accrue subtype constraints in addition to their declared bounds; updated bounds are recorded in the output environment E' . For example, if a right variable T appears on the right of the judgment, its lower bound can become larger (but not larger than the upper bound):

$${}^R T_l^u \vdash t <: T \vdash {}^R T_{\text{Union}\{l,t\}}^u \quad \text{only if} \quad t <: u.$$

If the resulting constraints on a right variable are consistent, as checked by $\text{consistent}(T, E)$, subtyping succeeds. This corresponds to the intuition that for a right-hand side existential type, subtyping holds if there exists a valid instantiation of the type variable.

ENVIRONMENT STRUCTURE. The environment has a non-trivial structure. First, an environment E is composed of two stacks, denoted by $E.\text{curr}$ and $E.\text{past}$. The former, $E.\text{curr}$, is a stack of variables currently in scope (growing on the right), reflecting the order in which variables have been added to the scope. In addition to variables, $E.\text{curr}$ records *barriers*: tags pushed to the environment whenever the subtype check encounters an invariant constructor. Barriers will be discussed later. In the examples presented in the chapter, $E.\text{curr}$ is displayed in place of E for readability. The second list, $E.\text{past}$, keeps track of variables that are not any longer in scope. Consider the judgment:

$$\text{Tuple}\{\text{Ref}\{S\} \text{ where } S <: \text{Int}\} <: \text{Tuple}\{\text{Ref}\{T\}\} \text{ where } T.$$

In the derivation variable T is introduced by R_INTRO rule—before the variable S , but T 's bounds refer to S in the judgment

$${}^R T_{\text{Union}\{\}}^{\text{Any}}, {}^L S_{\text{Union}\{\}}^{\text{Int}} \vdash \text{Ref}\{S\} <: \text{Ref}\{T\} \vdash {}^R T_S^S, {}^L S_{\text{Union}\{\}}^{\text{Int}},$$

which appears in the derivation tree. Thus, when L_INTRO discharges S , the variable is removed from $E.curr$ and stored in $E.past$, which allows the subtyping algorithm to access discharged variables 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.

FROM FORALL/EXIST TO EXIST/FORALL. In some cases, enforcing the correct ordering of type variable quantifications requires extra care. Consider the judgment:

$$\text{Vector}\{\text{Matrix}\{T\} \text{ where } T\} \not\prec: \text{Vector}\{\text{Matrix}\{S\}\} \text{ where } S.$$

The type on the left denotes a vector of arbitrary matrices; the type on the right denotes the set of vectors of matrices where inner matrices share the same element type. If the subtyping algorithm simply introduced variables S and T into the environment, the following judgments would succeed, because for all instances of T there is a matching type for S :

$$\begin{aligned} R_{S_{\text{Union}\{\}}^{\text{Any}}}, L_{T_{\text{Union}\{\}}^{\text{Any}}} \vdash T <: S \vdash R_{S_T^{\text{Any}}}, L_{T_{\text{Union}\{\}}^{\text{Any}}} \\ R_{S_T^{\text{Any}}}, L_{T_{\text{Union}\{\}}^{\text{Any}}} \vdash S <: T \vdash R_{S_T^{\text{Any}}}, L_{T_{\text{Union}\{\}}^{\text{Any}}} \end{aligned}$$

However, we must instead find an instance of S such that the judgment holds forall T : perhaps surprisingly, the outer invariant construct **Vector** forces the the form of the rule of the order of quantifications. Instead of a forall/exist query we must solve an *exist/forall* one. To correctly account for the form of the rule 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 into $E.curr$ whenever the derivation goes through an invariant constructor in the rule APP_INV .

A variable S is defined to be *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:

$$R_{S_{\text{Union}\{\}}^{\text{Any}}}, \text{Barrier}, L_{T_{\text{Union}\{\}}^{\text{Any}}} \vdash T <: S.$$

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 the lower bound:

$$^R S_{\text{Union}\{\}}^{\text{Any}}, \text{Barrier}, ^L T_{\text{Union}\{\}}^{\text{Any}} \vdash \text{Any} <: S \vdash ^R S_{\text{Any}}^{\text{Any}}.$$

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

$$^R S_{\text{Any}}^{\text{Any}}, \text{Barrier}, ^L T_{\text{Union}\{\}}^{\text{Any}} \vdash S <: T,$$

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.

Note that 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:

$$\text{Ref}\{\text{Ref}\{T\} \text{ where } \text{Int} <: T <: \text{Int}\} <: \text{Ref}\{\text{Ref}\{S\}\} \text{ where } S.$$

OPERATIONS ON ENVIRONMENT. Recall that an *environment*, denoted by E , is composed by two stacks, denoted $E.\text{curr}$ and $E.\text{past}$, of variable definitions and barriers. The following operations are defined on environments, where v ranges over variable definitions and barriers:

$\text{add}(v, E)$: push v at top of $E.\text{curr}$;

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

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

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

$\text{update}(^R T_l^u, E)$: update the lower and upper bounds of the variable definition T in $E.\text{curr}$; fail if the variable definition is not found;

$\text{consistent}(T, E)$: search T in E . If the search returns $^L T_l^u$, then return true if $E \vdash l <: u$ and false otherwise; while building this judgment, recursive consistency checks are disabled. If the search returns $^R T_l^u$, then check if $E \vdash l <: u$ is derivable. If not, return false. The shorthand $\text{consistent}(E)$ checks the consistency of all variables in the environment E .

DISTRIBUTIVITY. The rule `TUPLE_LIFT_UNION` rewrites tuple types on the left-hand side of the judgment into 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 into disjunctive normal form even inside invariant constructors. Rewriting is performed by the auxiliary function $\text{lift_union}(t)$, which pulls unions and existential types out of tuples, anticipating syntactically the forall quantifications in a derivation; the definition of this function can be found in [Zappa Nardelli et al. 2018]. 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 $\text{unlift_union}(t)$ returns a type t' such that $t = \text{lift_union}(t')$.

The need for the `TUPLE_UNLIFT_UNION` rule is due to the *complex interaction between invariant constructors, union types, and existentials*. For instance, the judgment

$$\text{Ref}\{\text{Union}\{\text{Tuple}\{\text{Int}, \text{Bool}\}\}\} <: \text{Ref}\{\text{Tuple}\{T\}\} \text{ where } T$$

is valid because T can be instantiated with $\text{Union}\{\text{Int}, \text{Bool}\}$. However, building a derivation without the `TUPLE_UNLIFT_UNION` rule fails. Initially, the left-to-right check for invariant application generates the constraint $T >: \text{Union}\{\text{Int}, \text{Bool}\}$. Thus, the right-to-left check

$$R_{\text{Union}\{\text{Int}, \text{Bool}\}}^{\text{Any}} T \vdash \text{Tuple}\{T\} <: \text{Union}\{\text{Tuple}\{\text{Int}, \text{Bool}\}\}$$

gets stuck trying to prove $T <: \text{Int}$ or $T <: \text{Bool}$. Rule `TUPLE_UNLIFT_UNION` enables rewriting the right-to-left check into

$$R_{\text{Union}\{\text{Int}, \text{Bool}\}}^{\text{Any}} T \vdash \text{Tuple}\{T\} <: \text{Tuple}\{\text{Union}\{\text{Int}, \text{Bool}\}\},$$

which is provable because the existential quantifications is syntactically delayed due to the union on the right-hand side.

SUBTYPING EXISTENTIALS AND VARIABLES. 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. When the variable gets out of scope, it is deleted from the `curr` list and added to the `past` list of the environment. Variables with exist semantics might have had their bounds updated in unsatisfiable way; before moving them to *E.past*, the consistency of their bounds is checked by the $\text{consistent}(T, E)$ auxiliary function.

Note that existential types with inconsistent bounds, such as

$$\text{Tuple}\{T\} \text{ where } \text{Any} <: T <: \text{Int},$$

are considered well-formed, but they are not subtypes of $\text{Union}\{\}$ despite the fact that there are no values denoted by such types. In Julia, attempting to instantiate such a type will produce an error:

```
julia> (Tuple{T} where Any<:T<:Int){Int}
ERROR: TypeError: in Tuple, in T, expected Any<:T<:Int, got Type{Int}
```

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 with R_RIGHT , which updates the lower bound with $\text{Union}\{l, t\}$, one would expect the rule R_LEFT to update T upper bound with $t \cap u$. However, until our work on Julia started, it was believed that, because of invariance, the explicit ordering of the checks performed by rule APP_INV 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. This turned out to be false. Consider the following example:

$$\begin{array}{c} \text{Vector}\{\text{Vector}\{\text{Any}\}\} \\ <: \\ \text{Vector}\{\text{Union}\{\text{Vector}\{\text{Any}\}, \text{Vector}\{T\}\}\} \text{ where } T <: \text{Int}. \end{array}$$

This successful judgment contradicts the idea that $\text{Vector}\{T\}$ can be a subtype of $\text{Vector}\{\text{Any}\}$ only if T is equivalent to Any , which is not possible here. Back in 2018, both Julia and the specification presented here could build a derivation for the above judgment: due to the existential on the right-hand side, the check that ought to ensure $t <: u$, that is $\text{Any} <: \text{Int}$, is skipped when performing the left-to-right subtype check of the invariant constructor Vector . In response to this finding, Julia designers introduced a `simple_meet` function⁴ to compute intersection in simple cases. As of May 2023, the problematic judgment above no longer holds.

To account for the exist/forall quantification the form of the rule, the R_RIGHT rule does not apply if the type on the left is a left variable and the variables are in the exists/forall quantification (the check $\neg\text{outside}(T, S, E)$ is responsible for this). Matching R-L variables is specially dealt with 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 constrain it to only one type. For this, the check $u_2 <: l_2$

⁴ Julia codebase already included a complex algorithm that computed an approximation of intersection of two types, which was used internally to compute dataflow information. However, the algorithm was too slow to be integrated in the subtyping algorithm.

$$\begin{aligned}
\llbracket \text{Top} \rrbracket &= \text{Union}\{\} \\
\llbracket \alpha \rrbracket &= \alpha \\
\llbracket \forall \alpha \leq \tau. \tau' \rrbracket &= \text{Tuple}\{\text{Ref}\{\alpha\}, \llbracket \tau' \rrbracket\} \text{ where } \alpha >: \llbracket \tau \rrbracket
\end{aligned}$$

Figure 7: Encoding of F_{\leq}^N types in Julia

is sufficient, as the other direction is later verified by the environment consistency check.

3.3 UNDECIDABILITY OF SUBTYPING

It is not unusual for a statically typed programming language to have undecidable subtyping, as witnessed by Java and Scala [Grigore 2017; Hu and Lhoták 2019]. In practice, undecidability means that the compiler might not terminate on some programs. Although undesirable, such property can be acceptable if it manifests rarely and allows for an expressive type system.

In Julia, however, subtyping is used at run time—for dispatch resolution. Thus, undecidability is concerning even on rare occasions. That is why the decidability of subtyping was one of the explicit goals of the original Julia language design [Bezanson 2015]. To this end, Julia disallowed several features that were known to cause undecidability, such as recursive constraints on type variables and circularities in the inheritance hierarchy [Tate et al. 2011].

Despite the intentional simplifications in the type language, Julia subtyping is in fact *undecidable*. As shown in [Chung 2023], Julia can encode system F_{\leq}^N , which is known to be undecidable [Pierce 1992]. Figure 7 shows the encoding⁵, with $\tau_1 \leq \tau_2$ in F_{\leq}^N defined as $\llbracket \tau_2 \rrbracket <: \llbracket \tau_1 \rrbracket$ in Julia. In practice, the undecidability manifests itself with a `StackOverflowError`. The reason is that, internally, Julia relies on a dedicated stack to resolve subtyping and terminates the program when the stack reaches a certain limit.

Although the nontermination of a particular subtyping algorithm does not necessarily mean that a terminating algorithm for the same language of types does not exist, I conjecture that

in the case of Julia’s type language, there is no terminating subtyping algorithm that would match the intended subtyping behavior described in Sections 3.1 and 3.2.

The key problem is the interaction of impredicative existential types, invariant constructors, and unions. Note that the encoding of F_{\leq}^N does not involve unions, meaning that Julia subtyping is undecidable even

⁵ Arrow type is dropped as irrelevant to the undecidability result.

if union types are removed. However, as discussed later, the presence of unions limits available strategies of addressing the undecidability.

In what follows, I briefly revisit subtyping of invariant constructors and existential types, discuss why their interaction is challenging, and conclude with possible ways of addressing the challenge. I intentionally depart from the subtyping judgment $E \vdash t <: t' \vdash E'$ presented in the previous section (3.2) and instead use a simpler $t <: t'$ notation, as I am exploring the space of possible subtyping algorithms.

First, recall that parametric types are invariant, meaning that types such as $\text{Ref}\{t\}$ and $\text{Ref}\{t'\}$ are subtypes only if their type arguments t and t' are equivalent. The equivalence of types is expressed via subtyping: t and t' are equivalent if both $t <: t'$ and $t' <: t$ hold. To check for equivalence, one can either rely on subtyping directly, or use a separate relation $t \approx t'$ such that $t \approx t' \implies t <: t'$ and $t' <: t$.

Next, recall that an existential type $t \text{ where } l <: T <: u$ is meant to represent a union of types $t[t'/T]$ for all possible valid instantiations $l <: t' <: u$ of the type variable T . Thus, whenever subtyping is checked for some $t_1 <: (t \text{ where } l <: T <: u)$, the subtyping algorithm needs to check that there exists a valid instantiation t' of T such that $t_1 <: t[t'/T]$. From now on, I will refer to such variable T (that is, a variable introduced by an existential type on the right-hand side of the judgment) as a **unification variable**; in Section 3.2, it was called an exist/right variable.

Consider the following subtype query

$$\begin{array}{c} \text{Tuple}\{\text{Vector}\{t'^S \text{ where } l' <: S <: u'\}, t'\} \\ <: \\ \text{Tuple}\{\text{Vector}\{t_1^T\}, t_2^T\} \text{ where } l <: T <: u, \end{array}$$

where S occurs in t'^S and T occurs in both t_1^T and t_2^T . Let us walk through possible steps of a subtyping algorithm \mathcal{A} .

1. Because the right-hand side type is an existential, T is a unification variable; \mathcal{A} can remember the variable and proceed to subtyping of

$$\text{Tuple}\{\text{Vector}\{t'^S \text{ where } l' <: S <: u'\}, t'\} <: \text{Tuple}\{\text{Vector}\{t_1^T\}, t_2^T\}.$$

2. Both sides of the subtyping query are now tuples of matching length. Tuples are covariant, so \mathcal{A} needs to check that their corresponding components are subtypes, i.e.,

$$\text{Vector}\{t'^S \text{ where } l' <: S <: u'\} <: \text{Vector}\{t_1^T\} \text{ and } t' <: t_2^T.$$

Since the unification variable T appears in both t_1^T and t_2^T , both subtyping queries can impose constraints on T .

a) The first tuple components are both invariant constructors,

$$\text{Vector}\{t'^S \text{ where } l' <: S <: u'\} <: \text{Vector}\{t_1^T\},$$

so \mathcal{A} needs to check that their type arguments are equivalent. Let \mathcal{A} use subtyping for the equivalence check and focus on the second, right-to-left check:

$$t_1^T <: t'^S \text{ where } l' <: S <: u'.$$

- i. The right-hand side type is an existential, so S is a unification variable; \mathcal{A} can remember the variable and proceed to subtyping of

$$t_1^T <: t'^S.$$

- ii. Let us assume that \mathcal{A} successfully processes this subtyping query. Note that *both sides of the query contain unification variables*, for T occurs in t_1^T and S occurs in t'^S . Thus, the unification variables S and T might be *constraining each other*.
- iii. \mathcal{A} needs to resolve the unification variable S , that is, check that constraints that were necessary to satisfy

$$t_1^T <: t'^S$$

are consistent with each other as well as the declared bounds l' and u' . However, S might be constrained by the unification variable T , and not all constraints on T are known, for $t' <: t_2^T$ has not been processed yet. Thus, *the subtyping algorithm \mathcal{A} is stuck*.

How can the problem in 2(a)iii be addressed? In the example, S cannot be resolved because not all relevant constraints are available. Perhaps, instead of trying to resolve the variable at the point where it goes out of lexical scope, \mathcal{A} could instead store the constraints and resolve them later. However, another—more challenging—problem is that the two unification variables S and T may be **arbitrarily constraining each other**. In the presence of impredicative existential types, constraints may have non-trivial solutions. For example, the following constraint set is satisfiable when both S and T are instantiated with $\text{Ref}\{Q\}$ where Q :

$$\{\text{Ref}\{T\} <: S, \text{Ref}\{\text{Ref}\{S\}\} <: T, S <: T\}$$

In general, for an arbitrary set of constraints with mutually constraining unification variables, finding a solution algorithmically is not straightforward if not undecidable. Therefore, instead, I suggest **eliminating mutual constraints on unification variables**. For this, I consider two possible angles of attack.

1. *Prevent unification variables from moving to the left-hand side of subtyping.* Recall that unification variables are introduced by existential types on the right. The only way for a unification variable to “travel” to the left is a right-to-left subtyping check for an invariant constructor. Thus, instead of using subtyping to check for equivalence of types, \mathcal{A} could rely on a separate relation $t \approx t'$ and keep unification variables on the right. Unfortunately, due to union types, checking for equivalence without relying on subtyping might not be possible. For instance, consider the following example:

$\text{Vector}\{\text{Ref}\{\text{Int}\}\} <: \text{Vector}\{\text{Union}\{\text{Ref}\{\text{Int}\}, \text{Ref}\{T\}\}\} \text{ where } T.$

Instantiating T with Int is the only solution that would make types $\text{Ref}\{\text{Int}\}$ and $\text{Union}\{\text{Ref}\{\text{Int}\}, \text{Ref}\{T\}\}$ equivalent. However, to find the solution, \mathcal{A} would effectively need to check $\text{Ref}\{T\} <: \text{Ref}\{\text{Int}\}$, bringing the unification variable to the left.

2. *Prevent new unification variables from appearing on the right-hand side of subtyping.* Assuming that \mathcal{A} has to rely on subtyping for the equivalence check, unification variables introduced by top-level existential types (i.e. existential types bound outside invariant constructors) will appear on the left-hand side. Once that happens, existential types bound *inside* invariant constructors will be the only source of new unification variables on the right. Thus, the elimination of variable-introducing existential types inside invariant constructors would prevent unwanted dependencies between unification variables.

In the next chapter, I present a decidable subtype relation that follows the strategy 2 by *restricting* existential types inside invariant constructors. A simpler approach would be to entirely eliminate impredicative existentials, that is, allow only top-level existential types such as

$\text{Vector}\{\text{Matrix}\{T\}\} \text{ where } T$

but not

$\text{Vector}\{\text{Matrix}\{T\} \text{ where } T\}.$

However, as discussed previously, types like the latter Vector type above are used by Julia programmers to represent heterogeneous data, so the simple approach appears unsatisfactory.

DECIDABLE SUBTYPING OF EXISTENTIAL TYPES

This chapter presents a decidable subtype relation for a core language of Julia types that includes covariant tuples, invariant type constructors, unions, existential types, and distributivity. As discussed in Section 3.3, it is the interaction of impredicative existential types, invariant constructors, and unions that makes Julia subtyping so challenging to decide, which is why I focus on this sublanguage first, in Section 4.1. In Section 4.3, I extend the core language with nominal subtyping and the diagonal rule.

The decidability of subtyping, discussed in detail in Section 4.2, is achieved by restricting existential types allowed to appear inside invariant constructors. In particular, the restriction limits such inner existentials to the ones expressible with Java wildcards [Torgersen et al. 2004]. As a result, *variable-introducing* existential types inside invariant constructors can be *eliminated* in accordance with the strategy 2 on page 26 (Section 3.3). The proposed restriction reduces the space of types representing heterogeneous data, but retains top-level existential types, which are used to represent parametric method definitions. As I show in Chapter 5, very few type annotations in a corpus of registered Julia packages use full-fledged existential types that are not representable under the restriction.

In Java, the wildcards mechanism provides *use-site variance* [Krab Thorup and Torgersen 1999], a restricted form of bounded existential types [Igarashi and Viroli 2002]. For example, the wildcard type

```
List<? extends Number>
```

represents an existential type

$$\exists X \text{:} \text{Number} . \text{List} \langle X \rangle .$$

Thus, the type `List<List<?>>` represents a heterogeneous list of lists where inner lists may have different element types. With existential types, the above type can be written as

$$\text{List} \langle \exists X . \text{List} \langle X \rangle \rangle ,$$

which corresponds to the following Julia type:

```
Vector{Vector{T} where T}.
```

Note that with wildcards, every occurrence of “?” introduces a fresh existential variable that *cannot be referenced by name*, and accordingly, has the following properties:

1. occurs in the type exactly once—as an argument of a type constructor;
2. is bound immediately outside the containing type constructor;
3. cannot have recursive constraints.

For example, the type `List<Pair<?, ?>>` represents a heterogeneous list of pairs `List<∃X. ∃Y. Pair<X, Y>>`. However, there is no way of restricting both elements of the pair to the same type, which is possible with full-fledged existential types: `List<∃X. Pair<X, X>>`.

Due to this “namelessness” property, subtyping of wildcard-induced existential types can be checked without explicitly introducing existential variables. For example, the following subtyping

```
List<? super l extends u> <: List<? super l' extends u'>
```

holds as long as the bounds of the implicit variable on the right-hand side contain the bounds of the left-hand side one, that is,

$$l' <: l \quad \text{and} \quad u <: u'.$$

Thus, in the case of Julia subtyping,

wildcard-induced existentials can be allowed inside invariant constructors without jeopardizing the decidability of subtyping,

as they will not introduce new unification variables.

Conveniently, the Julia language already supports a shorthand notation corresponding to the proposed restriction¹. For instance, types `Vector`, `Vector{<:Number}`, and `Vector{>:Int}` represent `Vector{T} where T`, `Vector{T} where T<:Number`, and `Vector{T} where T>:Int`, respectively.

This chapter is organized as follows:

- Section 4.1 provides a complete definition of decidable subtyping for the core language of Julia types, with the above described restriction on existential types inside invariant constructors.

¹ As of June 2023, it is impossible to specify both lower and upper bounds with this notation.

$\psi ::=$		<i>Type signature</i>	$\tau, l, u ::=$		<i>Type</i>
\top		top	\top		top
\perp		bottom	\perp		bottom
V		type variable	V		type variable
$\psi_1 \times \psi_2$		covariant tuple	$\tau_1 \times \tau_2$		covariant tuple
$N\{v, \dots\}$		invariant constr.	$N\{v, \dots\}$		invariant constr.
$\psi_1 \cup \psi_2$		union	$\tau_1 \cup \tau_2$		union
$\exists l <: V <: u. \psi$		existential			
$V ::=$		<i>Type variable</i>			
X, Y, \dots		universal var.			
α, β, \dots		unification var.			
$v ::=$	$l \ll u$	<i>Restricted existential var.</i>			

Figure 8: Grammar of type signatures and types

- Section 4.2 examines several properties of subtyping, in particular, decidability of subtyping (Theorem 1) and soundness of constraint resolution (Theorem 7).
- Section 4.3 extends the core language with nominal subtyping and the diagonal rule.

4.1 DEFINITION OF SUBTYPING

The restricted type language is given in Figure 8. For brevity, I switch to a shorter notation, with \top , \perp , \times , \cup , and \exists used instead of `Any`, `Union{}`, `Tuple`, `Union`, and `where`. N (a shorthand for $N\{\}$) and $N\{\dots\}$ represent non-parametric and invariant parametric types, where datatype declarations are implicit and do not impose restrictions on type parameters.

The type language distinguishes between more expressive **type signatures** ψ and less expressive **types** τ :

- type signatures ψ correspond to method signatures and allow for explicit existential types bound outside invariant constructors; variable bounds cannot be recursive but are allowed to refer to other variables in scope;
- types τ describe data; they are similar to type signatures but support only a restricted form of existential types $N\{v, \dots\}$.

Semantically, an existential $\exists l <: V <: u. \psi$ represents a union of $\psi[V \mapsto \tau]$ for all valid *type* instantiations $l <: \tau <: u$ of the type variable V . In the spirit of Java wildcards, a *restricted existential type* $N\{l_1 \ll u_1, \dots\}$ represents $\exists l_1 <: X <: u_1. \exists \dots N\{X_1, \dots\}$, and $N\{\tau_1, \dots\}$ is a shorthand for the tightly-bounded existential $N\{\tau_1 \ll \tau_1, \dots\}$. Note that the same seman-

tic type can have multiple syntactic representations. For example, the following pairs of types are equivalent:

$$\begin{array}{lll}
 \text{Int} & \approx & \text{Int} \cup \text{Int} \\
 (\text{Int} \cup \text{Flt}) \times \text{Str} & \approx & (\text{Int} \times \text{Str}) \cup (\text{Flt} \times \text{Str}) \\
 \exists \perp <: V <: \text{Str}. V \times \text{Int} & \approx & \text{Str} \times \text{Int} \\
 \exists \perp <: V <: T. \text{Ref}\{V\} & \approx & \text{Ref}\{\perp \ll T\}
 \end{array}$$

To visually aid the perception of inference rules that define subtyping, I use two styles of type variables: X, Y, \dots (referred to as universal variables) and α, β, \dots (referred to as unification variables), with V used when the distinction is irrelevant. Unification variables, discussed extensively in Section 3.3, are variables introduced by existential types on the right-hand side of a subtyping judgment. Universal variables are variables introduced on the left. In Section 3.2, universal and unification variables were called left/forall and right/exist, respectively.

In the next section, I give an overview of the subtyping algorithm for the type language in Figure 8, with the exact definition and decidability discussed separately. **The subtyping algorithm** is given by subtyping rules in Figures 10, 11, and 13, along with the constraint resolution algorithm in Figure 14. The rules are not syntax-directed, i.e., there may be multiple rules applicable to a pair of types or type signatures; however, the rules are *analytic* [Martin-Löf 1994]: there is a finite number of applicable rules, and the premises of each rule are comprised of the subcomponents of its conclusion. Subtyping holds if there is at least one successful derivation using the subtyping rules. Due to the presence of distributivity, the algorithm is exponential; Julia’s efficient implementation of distributive subtyping for unions and covariant tuples, exponential in time but linear in space, is described in [Chung et al. 2019].

4.1.1 Overview

Subtyping starts with subtyping of type signatures (Figure 13):

$$\Gamma \mid \Delta \vdash \psi <: \psi'.$$

Here, all explicit existential variables from ψ (universal variables) are introduced into environment Γ (defined in Figure 9), and variables from ψ' (unification variables) are introduced into Δ . To reach all existential types, it may be necessary to apply distributivity and go through union types. The following derivation provides an example:

$$\begin{array}{c}
\text{<discussed below>} \\
\hline
X <: \text{Int} \mid \alpha \vdash \text{Str} \times \text{Ref}\{X\} <: T \times \text{Ref}\{\alpha\} \quad \text{SS-TYPES} \\
\hline
X <: \text{Int} \mid \cdot \vdash \text{Str} \times \text{Ref}\{X\} <: \exists \alpha. (T \times \text{Ref}\{\alpha\}) \quad \text{SS-EXISTRIGHT} \\
\hline
\cdot \mid \cdot \vdash \text{Str} \times (\exists X <: \text{Int}. \text{Ref}\{X\}) <: \exists \alpha. (T \times \text{Ref}\{\alpha\}) \quad \text{SS-EXISTLEFT} \\
\hline
\cdot \mid \cdot \vdash \text{Str} \times (\exists X <: \text{Int}. \text{Ref}\{X\}) <: \text{Int} \cup \exists \alpha. (T \times \text{Ref}\{\alpha\}) \quad \text{SS-UNIONRIGHT}
\end{array}$$

First, SS-UNIONRIGHT picks the second type on the right. Second, because of the distributivity, the existential binding on the left can be pulled through the tuple by SS-EXISTLEFT. Finally, SS-EXISTRIGHT opens the existential on the right.

Once the algorithm reaches types on both sides, i.e.

$$\Gamma \mid \Delta \vdash \tau <: \tau',$$

subtyping should succeed if there is a valid substitution ρ of unification variables from Δ such that $\rho(\tau) <: \rho(\tau')$. This is done in two steps (SS-TYPES):

1. constrained subtyping $\Gamma \mid \text{dom}(\Delta) \vdash \tau <\bullet \tau' \rightsquigarrow K$ generates a constraint set K ;
2. **Solve**($\Gamma; \Delta; K$) resolves the constraints.

When constrained subtyping (Figure 11) takes over,

$$\Gamma \mid H \vdash \tau <\bullet \tau' \rightsquigarrow K \text{ where } H = \text{dom}(\Delta),$$

the algorithm checks subtyping, possibly generating constraints $l \leq \alpha$ and $\alpha \leq u$ on unification variables α from H that may appear in τ' . This step ignores the declared unification variable bounds, as emphasized by the environment H as opposed to Δ . Initially, all unification variables are located on the right, which is indicated with \bullet on the right of $<\bullet$. In the case of invariant constructors, the right-to-left subtyping check is performed by $\Gamma \mid H \vdash \tau' \bullet < \tau \rightsquigarrow K$, with all unification variables being located on the left. Thus, constrained subtyping maintains the invariant that unification variables always appear on at most one side of the subtyping judgment, which prevents the problem of mutually constraining unification variables discussed in Section 3.3. Because of this, types l, u in generated constraints are guaranteed to be *free from unification variables*. Thus, the running example above

ζ	$::= \square$		$\zeta \times \psi$		$\psi \times \zeta$	<i>Distributivity context for type signatures</i>
δ	$::= \square$		$\delta \times \tau$		$\tau \times \delta$	<i>Distributivity context for types</i>
Γ, Δ	$::= \cdot$		$\Gamma, l <: V <: u$			<i>Type variable environment</i>
H	$::= \cdot$		H, V			<i>Type variable list</i>
K	$::=$		$\{l \leq \alpha, \alpha \leq u, \dots\}$			<i>Constraint set</i>

Figure 9: Auxiliary definitions

continues as follows, generating the constraint set $\{X \leq \alpha, \alpha \leq X\}$.

	SC-UVAR _{RIGHT}	SC-UVAR _{LEFT}
SC-Top	$\frac{}{X<:\text{Int} \mid \alpha \vdash X \bullet \alpha \rightsquigarrow \{X \leq \alpha\}}$	$\frac{}{X<:\text{Int} \mid \alpha \vdash \alpha \bullet X \rightsquigarrow \{\alpha \leq X\}}$
$\frac{}{X<:\text{Int} \mid \alpha \vdash \text{Int} \bullet \top \rightsquigarrow \emptyset}$	$\frac{}{X<:\text{Int} \mid \alpha \vdash \text{Ref}\{X\} \bullet \text{Ref}\{\alpha\} \rightsquigarrow \{X \leq \alpha, \alpha \leq X\}}$	
$\frac{}{X<:\text{Int} \mid \alpha \vdash \text{Str} \times \text{Ref}\{X\} \bullet \top \times \text{Ref}\{\alpha\} \rightsquigarrow \{X \leq \alpha, \alpha \leq X\}}$		

If constrained subtyping succeeds and generates a constraint set K , the constraints are resolved by **Solve**($\Gamma; \Delta; K$) (Figure 14). Namely, all constraints on the same unification variable are checked for consistency with each other and with the declared variable bounds from Δ . If all the constraints are consistent, the variable is instantiated with the smallest type—a union of all (declared and generated) lower bounds. The consistency of constraints is checked with:

- constrained subtyping $\Gamma \mid H \vdash l \bullet \tau \rightsquigarrow K$ or $\Gamma \mid H \vdash \tau \bullet u \rightsquigarrow K$, to check that generated unification-free constraints τ are consistent with declared bounds l and u ; constrained subtyping is necessary because unification variable bounds may reference other unification variables;
- unification-free subtyping of types $\Gamma \vdash l <: u$ (Figure 10), to check that generated constraints are consistent with each other.

In the running example, the generated bound X of the unification variable α is trivially consistent with the declared bounds \perp and \top ; furthermore, $X <: \text{Int} \vdash X <: X$ by reflexivity, so the unification variable α is instantiated with X .

In unification-free subtyping $\Gamma \vdash \tau <: \tau'$, which is used for consistency checks, variables are treated as universal. For example, with the exception of the reflexive case, a variable V is a subtype of τ only if the upper bound u of V is a subtype of τ . This is similar to subtyping of left/forall variables in Figure 6, Section 3.2.

Next, I present the exact definitions of:

- unification-free subtyping of types $\Gamma \vdash \tau <: \tau'$ (Section 4.1.2);

- constrained subtyping of types $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$ (Section 4.1.3);
- constraint resolution $\text{Solve}(\Gamma; \Delta; K)$ (Section 4.1.4);
- signature subtyping $\Gamma \mid \Delta \vdash \psi <: \psi'$ (Section 4.1.4).

For simplicity of presentation, I assume the following variable name convention based on Barendregt’s convention:

Definition 1 (Variable name convention). *Everywhere in definitions and proofs, the following conditions hold:*

- all bound variables in τ, ψ are different from each other and from free variables;
- all variables in Γ, Δ, H are different from each other;
- whenever both Γ and Δ or Γ and H appear in the same judgment, $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ and $\text{dom}(\Gamma) \cap \text{dom}(H) = \emptyset$;
- whenever a variable environment and a type/type signature appear in the same judgment, bound variables of the type/type signature are different from variables in the environment;
- whenever multiple types/type signatures appear in the same judgment, their bound variables are different.

These conditions can be maintained by alpha-renaming.

While named type variables and the variable name convention are used for readability, they can be replaced by De Bruijn indices [de Bruijn 1972]. In the case of subtyping judgments with two environments, i.e., signature subtyping $\Gamma \mid \Delta \vdash \psi <: \psi'$ and constrained subtyping $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$, variable bindings in ψ/τ and ψ'/τ' need to range over different sets of indices to distinguish between universal variables from Γ and unification variables from Δ/H .

4.1.2 Unification-Free Subtyping of Types

Unification-free subtyping of types is given in Figure 10.

Following the semantic subtyping approach, covariant tuples distribute over union types (ST-UNIONLEFT), and tuples containing the bottom type are subtypes of all types (ST-BOT). Both of these rules are expressed with the distributivity context δ defined in Figure 9, which allows unions and \perp to be “pulled through” covariant tuples. Although in Julia, only the proper bottom type \perp is a subtype of all types (but not $\delta[\perp]$), I consider the more general ST-BOT in accordance with semantic subtyping. To obtain Julia’s semantics, ST-BOT should

$$\begin{array}{c}
\boxed{\Gamma \vdash \tau <: \tau} \\
\\
\text{ST-Top} \quad \frac{}{\Gamma \vdash \tau <: \top} \qquad \text{ST-Bot} \quad \frac{}{\Gamma \vdash \delta[\perp] <: \tau'} \\
\\
\text{ST-VARREFL} \quad \frac{l <: V <: u \in \Gamma}{\Gamma \vdash V <: V} \quad \text{ST-VARLEFT} \quad \frac{l <: V <: u \in \Gamma \quad \Gamma \vdash \delta[u] <: \tau'}{\Gamma \vdash \delta[V] <: \tau'} \quad \text{ST-VARRIGHT} \quad \frac{l <: V <: u \in \Gamma \quad \Gamma \vdash \tau <: l}{\Gamma \vdash \tau <: V} \\
\\
\text{ST-TUPLE} \quad \frac{\Gamma \vdash \tau_1 <: \tau'_1 \quad \Gamma \vdash \tau_2 <: \tau'_2}{\Gamma \vdash \tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2} \quad \text{ST-INV} \quad \frac{\forall i \in 1..n. \quad \Gamma \vdash v_i <: v'_i}{\Gamma \vdash N\{v_1, \dots, v_n\} <: N\{v'_1, \dots, v'_n\}} \\
\\
\text{ST-UNIONLEFT} \quad \frac{\Gamma \vdash \delta[\tau_1] <: \tau' \quad \Gamma \vdash \delta[\tau_2] <: \tau'}{\Gamma \vdash \delta[\tau_1 \cup \tau_2] <: \tau'} \quad \text{ST-UNIONRIGHT} \quad \frac{\exists i. \Gamma \vdash \tau <: \tau'_i}{\Gamma \vdash \tau <: \tau'_1 \cup \tau'_2} \\
\\
\boxed{\Gamma \vdash v <: v} \\
\\
\frac{\Gamma \vdash l' <: l \quad \Gamma \vdash u <: u'}{\Gamma \vdash l \ll u <: l' \ll u'} \\
\\
\boxed{\Gamma \vdash \delta <: \delta} \\
\\
\frac{}{\Gamma \vdash \square <: \square} \quad \frac{\Gamma \vdash \delta_1 <: \delta'_1 \quad \Gamma \vdash \tau_2 <: \tau'_2}{\Gamma \vdash \delta_1 \times \tau_2 <: \delta'_1 \times \tau'_2} \quad \frac{\Gamma \vdash \tau_1 <: \tau'_1 \quad \Gamma \vdash \delta_2 <: \delta'_2}{\Gamma \vdash \tau_1 \times \delta_2 <: \tau'_1 \times \delta'_2}
\end{array}$$

Figure 10: Subtyping of types (free from unification variables)

be written as $\Gamma \vdash \perp <: \tau'$; the change does not affect any properties of subtyping discussed in this chapter.

Subtyping of variables corresponds to the intuition that subtyping should hold for all their valid instantiations. Thus, when a variable appears covariantly on the left, it is replaced with the upper bound (ST-VARLEFT). When a variable appears on the right, it is replaced with the lower bound (ST-VARRIGHT). The ST-VARLEFT rule has to use the distributivity context, for the upper bound can be a union or bottom type. Without δ , the following judgment would not be derivable:

$$\perp <: V <: (\text{Int} \cup \text{Flt}) \vdash V \times T <: \text{Int} \times T \cup \text{Flt} \times T.$$

Finally, subtyping of invariant constructors/restricted existential types ensures that for each type parameter, the bounds on the left are contained within the bounds on the right. This corresponds to the intuition that for all valid instantiations of a variable on the left, there should be a valid instantiation of a variable on the right. For example, subtyping

$$\vdash \text{Ref}\{\perp \ll \text{Int}\} <: \text{Ref}\{\perp \ll T\}$$

holds, whereas

$$\vdash \text{Ref}\{\perp \ll \text{Int}\} \not<: \text{Ref}\{\text{Int} \ll T\}$$

does not.

Subtyping of distributivity contexts $\Gamma \vdash \delta <: \delta'$ is useful for proofs but is not used in the definition of $\Gamma \vdash \tau <: \tau'$.

4.1.3 Constrained Subtyping of Types

Constrained subtyping of types is given in Figure 11. The figure defines two mutually recursive relations,

$$\Gamma \mid H \vdash \tau \bullet \leq \tau' \rightsquigarrow K \quad \text{and} \quad \Gamma \mid H \vdash \tau \leq \bullet \tau' \rightsquigarrow K,$$

where the former is used when unification variables appear on the left, and the latter is used when unification variables appear on the right. For brevity, similar rules of the two relations are abbreviated with a single rule

$$\Gamma \mid H \vdash \tau \leq \tau' \rightsquigarrow K.$$

Most of the rules of constrained subtyping match the rules of unification-free subtyping, with the only difference being propagation

$\Gamma \mid H \vdash \tau < \tau \rightsquigarrow K$		
$\frac{}{\Gamma \mid H \vdash \tau < \top \rightsquigarrow \emptyset}$	$\frac{}{\Gamma \mid H \vdash \delta[\perp] < \tau' \rightsquigarrow \emptyset}$	$\frac{\alpha \in H}{\Gamma \mid H \vdash \delta[\alpha] \bullet < \tau' \rightsquigarrow \{\alpha \leq \perp\}}$
$\frac{l < X < u \in \Gamma}{\Gamma \mid H \vdash X < X \rightsquigarrow \emptyset}$	$\frac{\alpha \in H}{\Gamma \mid H \vdash \alpha \bullet < \tau' \rightsquigarrow \{\alpha \leq \tau'\}}$	$\frac{\alpha \in H}{\Gamma \mid H \vdash \tau < \bullet \alpha \rightsquigarrow \{\tau \leq \alpha\}}$
$\frac{l < X < u \in \Gamma \quad \Gamma \mid H \vdash \delta[u] < \tau' \rightsquigarrow K}{\Gamma \mid H \vdash \delta[X] < \tau' \rightsquigarrow K}$	$\frac{l < X < u \in \Gamma \quad \Gamma \mid H \vdash \tau < l \rightsquigarrow K}{\Gamma \mid H \vdash \tau < X \rightsquigarrow K}$	
$\frac{\Gamma \mid H \vdash \tau_1 < \tau'_1 \rightsquigarrow K_1 \quad \Gamma \mid H \vdash \tau_2 < \tau'_2 \rightsquigarrow K_2}{\Gamma \mid H \vdash \tau_1 \times \tau_2 < \tau'_1 \times \tau'_2 \rightsquigarrow K_1 \cup K_2}$		
$\frac{\forall i \in 1..n. \quad \Gamma \mid H \vdash v_i < v'_i \rightsquigarrow K_i}{\Gamma \mid H \vdash N\{v_1, \dots, v_n\} < N\{v'_1, \dots, v'_n\} \rightsquigarrow \bigcup_{i=1}^n K_i}$		
$\frac{\Gamma \mid H \vdash \delta[\tau_1] < \tau' \rightsquigarrow K_1 \quad \Gamma \mid H \vdash \delta[\tau_2] < \tau' \rightsquigarrow K_2}{\Gamma \mid H \vdash \delta[\tau_1 \cup \tau_2] < \tau' \rightsquigarrow K_1 \cup K_2}$	$\frac{\exists i. \Gamma \mid H \vdash \tau < \tau'_i \rightsquigarrow K}{\Gamma \mid H \vdash \tau < \tau'_1 \cup \tau'_2 \rightsquigarrow K}$	
$\frac{\alpha \in H \quad \alpha_1, \alpha_2 \text{ fresh} \quad \Gamma \mid H, \alpha_1 \vdash \delta[\alpha_1] \bullet < \tau'_1 \rightsquigarrow K_1 \quad \Gamma \mid H, \alpha_2 \vdash \delta[\alpha_2] \bullet < \tau'_2 \rightsquigarrow K_2 \quad K_1 = K'_1 \bigcup_{i=1}^n \{\alpha_1 \leq u_1^i\} \quad \alpha_1 \notin K'_1 \quad K_2 = K'_2 \bigcup_{j=1}^m \{\alpha_2 \leq u_2^j\} \quad \alpha_2 \notin K'_2 \quad K' = \{\alpha \leq \bigcap_{i=1}^n u_1^i \cup \bigcap_{j=1}^m u_2^j\}}{\Gamma \mid H \vdash \delta[\alpha] \bullet < \tau'_1 \cup \tau'_2 \rightsquigarrow K'_1 \cup K'_2 \cup K'}$		
$\Gamma \mid H \vdash v < v \rightsquigarrow K$		
$\frac{\Gamma \mid H \vdash l' < \bullet l \rightsquigarrow K_l \quad \Gamma \mid H \vdash u \bullet < u' \rightsquigarrow K_u}{\Gamma \mid H \vdash l < u \bullet < l' < u' \rightsquigarrow K_l \cup K_u}$		
$\frac{\Gamma \mid H \vdash l' \bullet < l \rightsquigarrow K_l \quad \Gamma \mid H \vdash u \bullet < \bullet u' \rightsquigarrow K_u}{\Gamma \mid H \vdash l < u \bullet < \bullet l' < u' \rightsquigarrow K_l \cup K_u}$		

Figure 11: Constrained subtyping of types

Every rule with the symbol \triangleleft is a shorthand for two rules, one where all occurrences of \triangleleft are replaced with $\bullet \triangleleft$, and another where all occurrences of \triangleleft are replaced with $\triangleleft \bullet$. Thus, the figure defines two mutually recursive relations, $\Gamma \mid H \vdash \tau \bullet \triangleleft \tau' \rightsquigarrow K$ and $\Gamma \mid H \vdash \tau \triangleleft \bullet \tau' \rightsquigarrow K$.

of constraints from recursive calls. For instance, compare subtyping of tuples:

$$\frac{\Gamma \vdash \tau_1 <: \tau'_1 \quad \Gamma \vdash \tau_2 <: \tau'_2}{\Gamma \vdash \tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2}$$

$$\frac{\Gamma \mid H \vdash \tau_1 < \tau'_1 \rightsquigarrow K_1 \quad \Gamma \mid H \vdash \tau_2 < \tau'_2 \rightsquigarrow K_2}{\Gamma \mid H \vdash \tau_1 \times \tau_2 < \tau'_1 \times \tau'_2 \rightsquigarrow K_1 \cup K_2}$$

In fact, when types do not contain unification variables, constrained and unification-free subtyping coincide (Lemma 12).

The only case where $\Gamma \mid H \vdash \tau \bullet < \tau' \rightsquigarrow K$ needs to call $\Gamma \mid H \vdash \tau < \bullet \tau' \rightsquigarrow K$ and vice versa is subtyping of invariant constructors/restricted existential types (SC-INV): here, the right-to-left check $\Gamma \mid H \vdash l' < l \rightsquigarrow K$ moves unification variables to the opposite side.

All the rules unique to constrained subtyping are highlighted in gray: SC-UBOT, SC-UVarLEFT, SC-UVarRIGHT, SC-UVarUNIONRIGHT. These are the rules that *generate* new constraints on unification variables rather than simply propagate them. The most interesting rule is SC-UVarUNIONRIGHT: it addresses the case where the unification variable could be instantiated with a union. Consider the following example:

$$\frac{\begin{array}{c} ? \\ \cdot \mid \alpha \vdash \alpha \times T \bullet < \text{Int} \times T \cup \text{Flt} \times T \rightsquigarrow ? \end{array} \quad \frac{\begin{array}{c} \dots \\ \cdot \mid \alpha \vdash \text{Int} \times T \bullet < \bullet \alpha \times T \rightsquigarrow \{\text{Int} \leq \alpha\} \end{array} \quad \dots}{\cdot \mid \alpha \vdash \text{Int} \times T \cup \text{Flt} \times T \bullet < \bullet \alpha \times T \rightsquigarrow \{\text{Int} \leq \alpha, \text{Flt} \leq \alpha\}} \quad \dots$$

$$\cdot \mid \alpha \vdash \text{Ref}\{\text{Int} \times T \cup \text{Flt} \times T\} \bullet < \bullet \text{Ref}\{\alpha \times T\} \rightsquigarrow ?$$

Without SC-UVarUNIONRIGHT, the best way to proceed with

$$\cdot \mid \alpha \vdash \alpha \times T \bullet < \text{Int} \times T \cup \text{Flt} \times T \rightsquigarrow ?$$

(besides SC-UBOT, which requires $\{\alpha \leq \perp\}$) would be to use SC-UNIONRIGHT and pick either $\text{Int} \times T$ or $\text{Flt} \times T$ on the right. This would result in constraining α with either $\{\alpha \leq \text{Int}\}$ or $\{\alpha \leq \text{Flt}\}$. Taken together with $\{\text{Int} \leq \alpha, \text{Flt} \leq \alpha\}$ from the right-hand side derivation, either of these constraints would render the resulting set of constraints unsatisfiable. However, subtyping

$$\cdot \mid \alpha \vdash \text{Ref}\{\text{Int} \times T \cup \text{Flt} \times T\} \bullet < \bullet \text{Ref}\{\alpha \times T\} \rightsquigarrow ?$$

clearly holds if α is instantiated with $\text{Int} \cup \text{Flt}$. Thus, when α occurs on the left covariantly, in a position where unification-free subtyping could appeal to ST-UNIONLEFT, the rule SC-UVarUNIONRIGHT allows

for a more permissive, union upper bound. Intuitively, when the right-hand side is a union, the rule proceeds as if the unification variable α stands for the union $\alpha_1 \cup \alpha_2$ such that, distributively, $\delta[\alpha_1]$ and $\delta[\alpha_2]$ are subtypes of τ_1' and τ_2' , respectively. Consider the application of SC-UVAR-UNIONRIGHT to the right-to-left check in the example:

$$\frac{\begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_1 \vdash \alpha_1 \times T \bullet \leq \text{Int} \times T \rightsquigarrow \{\alpha_1 \leq \text{Int}\} \end{array} \quad \begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_2 \vdash \alpha_2 \times T \bullet \leq \text{Flt} \times T \rightsquigarrow \{\alpha_2 \leq \text{Flt}\} \end{array}}{\cdot \mid \alpha \vdash \alpha \times T \bullet \leq \text{Int} \times T \cup \text{Flt} \times T \rightsquigarrow \{\alpha \leq \text{Int} \cup \text{Flt}\}}$$

Now the resulting constraint set $\{\alpha \leq \text{Int} \cup \text{Flt}, \text{Int} \leq \alpha, \text{Flt} \leq \alpha\}$ is satisfiable. Note that the rule SC-UVAR-UNIONRIGHT needs to be careful when the constraint set K_i produced by the premise

$$\Gamma \mid H, \alpha_i \vdash \delta[\alpha_i] \bullet \leq \tau_i' \rightsquigarrow K_i$$

contains multiple upper-bound constraints on α_i , because all those constraints need to hold at the same time. For instance, consider the following example, where $\tau_a <: \tau_a'$ and τ_a is incomparable with τ_b :

$$\cdot \mid \alpha \vdash \alpha \times \alpha \bullet \leq (\tau_a \times \tau_a \cup \tau_a' \times \tau_b) \cup \tau_b \times T \rightsquigarrow ?$$

There are multiple ways to instantiate α (e.g., with \perp), but the most permissive constraint set can be obtained by first applying SC-UVAR-UNIONRIGHT to the left occurrence of α :

$$\frac{\begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_1 \vdash \alpha_1 \times \alpha \bullet \leq \tau_a \times \tau_a \cup \tau_a' \times \tau_b \rightsquigarrow ? \end{array} \quad \begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_2 \vdash \alpha_2 \times \alpha \bullet \leq \tau_b \times T \rightsquigarrow \{\alpha_2 \leq \tau_b\} \end{array}}{\cdot \mid \alpha \vdash \alpha \times \alpha \bullet \leq (\tau_a \times \tau_a \cup \tau_a' \times \tau_b) \cup \tau_b \times T \rightsquigarrow ?}$$

Next, let us focus on the left premise and apply the same rule to the remaining α :

$$\frac{\begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_1, \alpha_3 \vdash \alpha_1 \times \alpha_3 \bullet \leq \tau_a \times \tau_a \rightsquigarrow \{\alpha_1 \leq \tau_a, \alpha_3 \leq \tau_a\} \end{array} \quad \begin{array}{c} \dots \\ \cdot \mid \alpha, \alpha_1, \alpha_4 \vdash \alpha_1 \times \alpha_4 \bullet \leq \tau_a' \times \tau_b \rightsquigarrow \{\alpha_1 \leq \tau_a', \alpha_4 \leq \tau_b\} \end{array}}{\cdot \mid \alpha, \alpha_1 \vdash \alpha_1 \times \alpha \bullet \leq \tau_a \times \tau_a \cup \tau_a' \times \tau_b \rightsquigarrow \{\alpha_1 \leq \tau_a, \alpha_1 \leq \tau_a', \alpha \leq \tau_a \cup \tau_b\}}$$

There are two constraints on α_1 , both of which have to be satisfied. Thus, SC-UVAR-UNIONRIGHT merges them into one type using the intersection function $\tau_a \sqcap \tau_a'$, defined in Figure 12. Intersection produces a type that is a subtype of both τ_a and τ_a' : in this case, the smaller type τ_a . Thus, the original subtyping judgment succeeds:

$$\cdot \mid \alpha \vdash \alpha \times \alpha \bullet \leq (\tau_a \times \tau_a \cup \tau_a' \times \tau_b) \cup \tau_b \times T \rightsquigarrow \{\alpha \leq \tau_a \cup \tau_b, \alpha \leq \tau_a \cup \tau_b\}$$

$\tau \sqcap_{\Gamma} \tau'$			
τ	\sqcap_{Γ}	τ'	$= \tau$ if $\Gamma \vdash \tau <: \tau'$
τ	\sqcap_{Γ}	τ'	$= \tau'$ if $\Gamma \vdash \tau' <: \tau$
$\tau_1 \cup \tau_2$	\sqcap_{Γ}	τ'	$= (\tau_1 \sqcap_{\Gamma} \tau') \cup (\tau_2 \sqcap_{\Gamma} \tau')$
τ	\sqcap_{Γ}	$\tau'_1 \cup \tau'_2$	$= (\tau \sqcap_{\Gamma} \tau'_1) \cup (\tau \sqcap_{\Gamma} \tau'_2)$
V	\sqcap_{Γ}	τ'	$= l \sqcap_{\Gamma} \tau'$ where $l <: V <: u \in \Gamma$
τ	\sqcap_{Γ}	V	$= \tau \sqcap_{\Gamma} l$ where $l <: V <: u \in \Gamma$
$\tau_1 \times \tau_2$	\sqcap_{Γ}	$\tau'_1 \times \tau'_2$	$= (\tau_1 \sqcap_{\Gamma} \tau'_1) \times (\tau_2 \sqcap_{\Gamma} \tau'_2)$
$N\{\dots, l_i \ll u_i, \dots\}$	\sqcap_{Γ}	$N\{\dots, l'_i \ll u'_i, \dots\}$	$= N\{\dots, (l_i \cup l'_i) \ll (u_i \sqcap_{\Gamma} u'_i), \dots\}$ where $\forall i. \Gamma \vdash (l_i \cup l'_i) <: (u_i \sqcap_{\Gamma} u'_i)$
τ	\sqcap_{Γ}	τ'	$= \perp$ otherwise

Figure 12: Intersection (\sqcap_{Γ}) of types

Some cases, such as \perp , \top , and the same type variable, are absent because they are covered by the cases $\Gamma \vdash \tau <: \tau'$ and $\Gamma \vdash \tau' <: \tau$.

(The two occurrences of the same constraint are due to the two separate applications of SC-UVAR-UNIONRIGHT, α_1/α_2 and α_3/α_4 .)

4.1.4 Signature Subtyping

The final piece of the subtyping algorithm is subtyping of top-level signatures, which is given in Figure 13:

$$\Gamma \mid \Delta \vdash \psi <: \psi'.$$

This step introduces all explicitly bound existential variables to environments Γ and Δ . Similarly to subtyping of types, the rules SS-BOT, SS-VARLEFT, SS-UNIONLEFT, and SS-INVLEFT use the distributivity context—in the case of type signatures, ζ . The rule SS-INVLEFT is needed to account for the distributivity of restricted existential types, similarly to explicit existentials. For example:

$$\begin{array}{c}
 \frac{\perp <: \alpha <: T \mid \alpha \vdash \text{Ref}\{X\} \times \text{Int} \leq \bullet \text{Ref}\{\alpha\} \times T \rightsquigarrow \{X \leq \alpha, \alpha \leq X\} \quad \dots}{\perp <: X <: T \mid \perp <: \alpha <: T \vdash \text{Ref}\{X\} \times \text{Int} <: \text{Ref}\{\alpha\} \times T} \text{SS-TYPES} \\
 \frac{\cdot \mid \perp <: \alpha <: T \vdash \text{Ref}\{\perp \ll T\} \times \text{Int} <: \text{Ref}\{\alpha\} \times T}{\cdot \mid \cdot \vdash \text{Ref}\{\perp \ll T\} \times \text{Int} <: \exists \alpha. \text{Ref}\{\alpha\} \times T} \text{SS-INVLEFT} \\
 \text{SS-EXISTRIGHT}
 \end{array}$$

Without SS-INVLEFT, constrained subtyping

$$\cdot \mid \alpha \vdash \text{Ref}\{\perp \ll T\} \times \text{Int} \leq \bullet \text{Ref}\{\alpha\} \times T \rightsquigarrow \{T \leq \alpha, \alpha \leq \perp\}$$

would generate unsatisfiable constraints.

Notice the absence of rules for subtyping tuples and invariant constructors: once types are reached on both sides, the rule SS-TYPES delegates further checks to constrained subtyping $\Gamma \mid H \vdash \tau \leq \bullet \tau' \rightsquigarrow K$, followed by constraint resolution **Solve**($\Gamma; \Delta; K$). The algorithm **Solve**,

$$\boxed{\Gamma \mid \Delta \vdash \psi <: \psi}$$

$$\begin{array}{c}
\text{SS-Top} \\
\hline
\Gamma \mid \Delta \vdash \psi <: \top
\end{array}
\qquad
\begin{array}{c}
\text{SS-Bot} \\
\hline
\Gamma \mid \Delta \vdash \zeta[\perp] <: \psi'
\end{array}$$

$$\begin{array}{c}
\text{SS-VARLEFT} \\
\frac{l <: X <: u \in \Gamma \quad \Gamma \mid \Delta \vdash \zeta[u] <: \psi'}{\Gamma \mid \Delta \vdash \zeta[X] <: \psi'}
\end{array}
\qquad
\begin{array}{c}
\text{SS-UNIONLEFT} \\
\frac{\Gamma \mid \Delta \vdash \zeta[\psi_1] <: \psi' \quad \Gamma \mid \Delta \vdash \zeta[\psi_2] <: \psi'}{\Gamma \mid \Delta \vdash \zeta[\psi_1 \cup \psi_2] <: \psi'}
\end{array}$$

$$\begin{array}{c}
\text{SS-INVLEFT} \\
\frac{X \text{ fresh} \quad \Gamma, l <: X <: u \mid \Delta \vdash \zeta[N\{\dots, X, \dots\}] <: \psi'}{\Gamma \mid \Delta \vdash \zeta[N\{\dots, l \ll u, \dots\}] <: \psi'}
\end{array}
\qquad
\begin{array}{c}
\text{SS-EXISTLEFT} \\
\frac{\Gamma, l <: X <: u \mid \Delta \vdash \zeta[\psi] <: \psi'}{\Gamma \mid \Delta \vdash \zeta[\exists l <: X <: u. \psi] <: \psi'}
\end{array}$$

$$\begin{array}{c}
\text{SS-UNIONRIGHT} \\
\frac{\exists i. \Gamma \mid \Delta \vdash \psi <: \zeta[\psi'_i]}{\Gamma \mid \Delta \vdash \psi <: \zeta[\psi'_1 \cup \psi'_2]}
\end{array}
\qquad
\begin{array}{c}
\text{SS-EXISTRIGHT} \\
\frac{\Gamma \mid \Delta, l <: \alpha <: u \vdash \psi <: \zeta[\psi']}{\Gamma \mid \Delta \vdash \psi <: \zeta[\exists l <: \alpha <: u. \psi']}
\end{array}$$

$$\begin{array}{c}
\text{SS-TYPES} \\
\frac{\Gamma \mid \text{dom}(\Delta) \vdash \tau <_{\bullet} \tau' \rightsquigarrow K \quad \text{Solve}(\Gamma; \Delta; K) = \rho}{\Gamma \mid \Delta \vdash \tau <: \tau'}
\end{array}$$

Figure 13: Subtyping of type signatures

defined in Figure 14, checks for consistency of all the constraints on each unification variable, starting with the last introduced one in Δ . Because variable bounds can refer to earlier-introduced variables, generated constraints are checked for consistency with declared constraints using constrained subtyping. The resulting constraints are then checked recursively for the smaller environment Δ .

Note that the constraint resolution algorithm does not need to compute ρ to check the consistency of constraints: the substitution is computed so that multiple dispatch can instantiate type variables in parametric method definitions. However, despite $\text{Solve}(\Gamma; \Delta; K)$ computing the smallest valid substitution for the given K , there can be multiple valid derivations of constrained subtyping that produce different sets of constraints. For example,

$$\cdot \mid \alpha \vdash \text{Ref}\{\text{Int}\} \cup \text{Ref}\{\text{Str}\} <_{\bullet} \top \cup \text{Ref}\{\alpha\} \rightsquigarrow \dots$$

is derivable with three different sets of constraints, solved by $\alpha \mapsto \perp$, $\alpha \mapsto \text{Int}$, and $\alpha \mapsto \text{Str}$. Therefore, the dispatch mechanism would need to decide which derivation to pick. And while, in this example, $\alpha \mapsto \perp$ is the smallest valid instantiation of α , the smallest one does not always exist globally. For example,

$$\cdot \mid \alpha, \beta \vdash \text{Ref}\{\text{Int}\} \cup \text{Ref}\{\text{Str}\} <_{\bullet} \text{Ref}\{\alpha\} \cup \text{Ref}\{\beta\} \rightsquigarrow \dots$$

```

Solve ( $\Gamma; \cdot; K$ )
| return []
Solve ( $\Gamma; \Delta, l <: \alpha <: u; K$ )
|  $K_\alpha \leftarrow \{l' \leq \alpha \mid l' \leq \alpha \in K\} \cup \{\alpha \leq u' \mid \alpha \leq u' \in K\};$ 
|  $K' \leftarrow K \setminus K_\alpha;$ 
|  $H \leftarrow \text{dom}(\Delta);$ 
| foreach  $l_i \leq \alpha, \alpha \leq u_j \in K_\alpha$  do  $\Gamma \vdash l_i <: u_j;$ 
| foreach  $l_i \leq \alpha \in K_\alpha$  do  $\Gamma \mid H \vdash l_i < \bullet u \rightsquigarrow K_{l_i};$ 
| foreach  $\alpha \leq u_j \in K_\alpha$  do  $\Gamma \mid H \vdash l \bullet < u_j \rightsquigarrow K_{u_j};$ 
|  $\rho \leftarrow \text{Solve}(\Gamma; \Delta; K' \cup_i K_{l_i} \cup_j K_{u_j});$ 
| return  $\rho[\alpha \mapsto \rho(l) \cup_i l_i]$ 

```

Figure 14: Constraints resolution algorithm **Solve**($\Gamma; \Delta; K$)

can be satisfied with either $[\alpha \mapsto \text{Int}, \beta \mapsto \text{Str}]$ or $[\alpha \mapsto \text{Str}, \beta \mapsto \text{Int}]$. Since the number of possible subtyping derivations (and corresponding constraint sets) is finite, it would be possible for the dispatch mechanism to compare substitutions produced by **Solve**($\Gamma; \Delta; K$) to pick the smallest one when it exists, or apply another selection mechanism when substitutions are incomparable.

Examining signature subtyping, the rule SS-UNIONLEFT may seem surprising because the two premises $\Gamma \mid \Delta \vdash \zeta[\psi_1] <: \psi'$ and $\Gamma \mid \Delta \vdash \zeta[\psi_2] <: \psi'$ are not required to have consistent instantiations of Δ . For example, in the following derivation,

$$\frac{\begin{array}{c} \cdot \mid \alpha \vdash \text{Ref}\{\text{Int}\} <: \text{Ref}\{\alpha\} \\ \cdot \mid \alpha \vdash \text{Ref}\{\text{Int}\} <: \text{Ref}\{\alpha\} \cup \text{Vector}\{\alpha\} \end{array}}{\cdot \mid \alpha \vdash \text{Ref}\{\text{Int}\} \cup \text{Vector}\{\text{Str}\} <: \text{Ref}\{\alpha\} \cup \text{Vector}\{\alpha\}}
 \quad
 \frac{\begin{array}{c} \cdot \mid \alpha \vdash \text{Vector}\{\text{Str}\} <: \text{Vector}\{\alpha\} \\ \cdot \mid \alpha \vdash \text{Vector}\{\text{Str}\} <: \text{Ref}\{\alpha\} \cup \text{Vector}\{\alpha\} \end{array}}{\cdot \mid \alpha \vdash \text{Ref}\{\text{Int}\} \cup \text{Vector}\{\text{Str}\} <: \text{Ref}\{\alpha\} \cup \text{Vector}\{\alpha\}}$$

α gets instantiated with `Int` in the first premise and `Str` in the second. This is correct because the types $\exists \alpha. (\psi_1 \cup \psi_2)$ and $(\exists \alpha. \psi_1) \cup (\exists \alpha. \psi_2)$ are semantically equivalent, with the equivalence being derivable in Julia as well:

```

julia> Union{Ref{Int}, Vector{String}} <:
    Union{Ref{T}, Vector{T}} where T
true

# t1 == t2 holds when t1 <: t2 and t2 <: t1
julia> Union{Ref{T} where T, Vector{T} where T} ==
    Union{Ref{T}, Vector{T}} where T
true

```

$$\begin{array}{c}
\boxed{\Gamma \vdash \tau} \\
\\
\frac{}{\Gamma \vdash \top} \quad \frac{}{\Gamma \vdash \perp} \quad \frac{l <: V <: u \in \Gamma}{\Gamma \vdash V} \\
\\
\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \quad \frac{N \text{ has arity } n \quad \forall i \in 1..n. \Gamma \vdash v_i}{\Gamma \vdash N\{v_1, \dots, v_n\}} \\
\\
\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \cup \tau_2} \\
\\
\boxed{\Gamma \vdash v} \\
\\
\frac{\Gamma \vdash l \quad \Gamma \vdash u \quad \Gamma \vdash l <: u}{\Gamma \vdash l \ll u} \\
\\
\boxed{\Gamma \vdash \delta} \\
\\
\frac{}{\Gamma \vdash \square} \quad \frac{\Gamma \vdash \delta \quad \Gamma \vdash \tau}{\Gamma \vdash \delta \times \tau} \quad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \delta}{\Gamma \vdash \tau \times \delta} \\
\\
\boxed{\Gamma \vdash \psi} \\
\\
\frac{}{\Gamma \vdash \top} \quad \frac{}{\Gamma \vdash \perp} \quad \frac{l <: V <: u \in \Gamma}{\Gamma \vdash V} \\
\\
\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \times \psi_2} \quad \frac{N \text{ has arity } n \quad \forall i \in 1..n. \Gamma \vdash v_i}{\Gamma \vdash N\{v_1, \dots, v_n\}} \\
\\
\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \cup \psi_2} \quad \frac{\Gamma \vdash l \quad \Gamma \vdash u \quad \Gamma \vdash l <: u \quad \Gamma, l <: V <: u \vdash \psi}{\Gamma \vdash \exists l <: V <: u. \psi} \\
\\
\boxed{\Gamma \mid H \vdash \tau} \\
\\
\frac{l <: X <: u \in \Gamma}{\Gamma \mid H \vdash X} \quad \frac{\alpha \in H}{\Gamma \mid H \vdash \alpha} \quad \frac{}{\Gamma \mid H \vdash \top} \quad \frac{}{\Gamma \mid H \vdash \perp} \\
\\
\frac{\Gamma \mid H \vdash \tau_1 \quad \Gamma \mid H \vdash \tau_2}{\Gamma \mid H \vdash \tau_1 \times \tau_2} \quad \frac{N \text{ has arity } n \quad \forall i \in 1..n. \Gamma \mid H \vdash v_i}{\Gamma \mid H \vdash N\{v_1, \dots, v_n\}} \\
\\
\frac{\Gamma \mid H \vdash \tau_1 \quad \Gamma \mid H \vdash \tau_2}{\Gamma \mid H \vdash \tau_1 \cup \tau_2} \\
\\
\boxed{\vdash \Gamma} \\
\\
\frac{}{\vdash \cdot} \quad \frac{\vdash \Gamma \quad \Gamma \vdash l \quad \Gamma \vdash u \quad \Gamma \vdash l <: u}{\vdash \Gamma, l <: V <: u}
\end{array}$$

Figure 15: Validity of types and type signatures

4.1.5 Validity of Types and Type Signatures

To provide transitivity and prevent the subtyping algorithm from getting stuck, it should be called with valid types and type signatures, as defined in Figure 15. In particular:

- $\Gamma \vdash \tau <: \tau'$ requires that $\Gamma \vdash \tau, \tau'$;
- $\Gamma \mid H \vdash \tau \bullet \tau' \rightsquigarrow K$ requires that $\Gamma \vdash \tau$ and $\Gamma \mid H \vdash \tau'$;
- $\Gamma \mid H \vdash \tau \bullet \tau' \rightsquigarrow K$ requires that $\Gamma \mid H \vdash \tau$ and $\Gamma \vdash \tau'$;
- $\Gamma \mid \Delta \vdash \psi <: \psi'$ requires that $\Gamma \vdash \psi$ and $\Gamma \uparrow\uparrow \Delta \vdash \psi'$, where $\Gamma \uparrow\uparrow \Delta$ concatenates two environments.

The validity check ensures that free variables are bound in corresponding environments and that variable bounds are non-recursive and consistent.

The consistency of declared variable bounds is necessary for transitivity. This requirement is a departure from Julia, where types like

```
Ref{T} where Any<:T<:Int
```

are considered valid. Although semantically, such types denote empty sets and cannot be instantiated, Julia does not consider them to be subtypes of `Union{}`:

```
julia> (Ref{T} where Any<:T<:Int){Int}
ERROR: TypeError: in Ref, in T, expected Any<:T<:Int

julia> (Ref{T} where Any<:T<:Int) <: Union{}
false
```

4.2 PROPERTIES OF SUBTYPING

This section defines and proves multiple properties about subtyping, most importantly:

- decidability of subtyping (Theorem 1);
- transitivity of unification-free subtyping (Theorem 3);
- soundness of constrained subtyping (Theorem 6) and constraint resolution (Theorem 7).

A detailed discussion of the decidability is given in Section 4.2.1. At a high-level, the subtyping algorithm terminates because:

$\mathcal{M}(\Gamma; \tau)$	
$\mathcal{M}(\Gamma; \top)$	$= 1$
$\mathcal{M}(\Gamma; \perp)$	$= 1$
$\mathcal{M}(\cdot; V)$	$= 1$
$\mathcal{M}(\Gamma, l < V <: u; V)$	$= 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u)$
$\mathcal{M}(\Gamma, l < V' <: u; V)$	$= \mathcal{M}(\Gamma; V)$
$\mathcal{M}(\Gamma; \tau_1 \times \tau_2)$	$= 1 + \mathcal{M}(\Gamma; \tau_1) + \mathcal{M}(\Gamma; \tau_2)$
$\mathcal{M}(\Gamma; N\{v_1, \dots, v_n\})$	$= 1 + \mathcal{M}(\Gamma; v_1) + \dots + \mathcal{M}(\Gamma; v_n)$
$\mathcal{M}(\Gamma; \tau_1 \cup \tau_2)$	$= 1 + \mathcal{M}(\Gamma; \tau_1) + \mathcal{M}(\Gamma; \tau_2)$
$\mathcal{M}(\Gamma; v)$	
$\mathcal{M}(\Gamma; \tau \ll \tau)$	$= \mathcal{M}(\Gamma; \tau)$
$\mathcal{M}(\Gamma; l \ll u)$	$= 2 \times (1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u))$
$\mathcal{M}(\Gamma; \delta)$	
$\mathcal{M}(\Gamma; \square)$	$= 0$
$\mathcal{M}(\Gamma; \delta \times \tau)$	$= 1 + \mathcal{M}(\Gamma; \delta) + \mathcal{M}(\Gamma; \tau)$
$\mathcal{M}(\Gamma; \tau \times \delta)$	$= 1 + \mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \delta)$
$\mathcal{M}(\Gamma; \psi)$	
$\mathcal{M}(\Gamma; \top)$	$= 1$
$\mathcal{M}(\Gamma; \perp)$	$= 1$
$\mathcal{M}(\cdot; V)$	$= 1$
$\mathcal{M}(\Gamma, l < V <: u; V)$	$= 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u)$
$\mathcal{M}(\Gamma, l < V' <: u; V)$	$= \mathcal{M}(\Gamma; V)$
$\mathcal{M}(\Gamma; \psi_1 \times \psi_2)$	$= 1 + \mathcal{M}(\Gamma; \psi_1) + \mathcal{M}(\Gamma; \psi_2)$
$\mathcal{M}(\Gamma; N\{v_1, \dots, v_n\})$	$= 1 + \mathcal{M}(\Gamma; v_1) + \dots + \mathcal{M}(\Gamma; v_n)$
$\mathcal{M}(\Gamma; \exists l < V <: u. \psi)$	$= 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u) + \mathcal{M}(\Gamma, l < V <: u; \psi)$
$\mathcal{M}(\Gamma; \zeta)$	
$\mathcal{M}(\Gamma; \square)$	$= 0$
$\mathcal{M}(\Gamma; \zeta \times \psi)$	$= 1 + \mathcal{M}(\Gamma; \zeta) + \mathcal{M}(\Gamma; \psi)$
$\mathcal{M}(\Gamma; \psi \times \zeta)$	$= 1 + \mathcal{M}(\Gamma; \psi) + \mathcal{M}(\Gamma; \zeta)$

Figure 16: Measure of types and type signatures

1. for each recursive call of unification-free subtyping, constrained subtyping, and signature subtyping, the arguments are cumulatively smaller with respect to a measure function defined in Figure 16;
2. constraint resolution terminates because consistency checks terminate and the Δ argument decreases with each recursive call.

Particularly interesting elements of the proofs are highlighted in **bold**.

4.2.1 Decidability

To show the decidability of the subtyping algorithm, I will use the measure \mathcal{M} of types and type signatures defined in Figure 16. The measure function is defined recursively and is similar to syntactic size, except for the treatment of type variables: for variables from Γ , the

$ \tau $	
$ \top $	$= 1$
$ \perp $	$= 1$
$ V $	$= 1$
$ \tau_1 \times \tau_2 $	$= 1 + \tau_1 + \tau_2 $
$ N\{v_1, \dots, v_n\} $	$= 1 + v_1 + \dots + v_n $
$ \tau_1 \cup \tau_2 $	$= 1 + \tau_1 + \tau_2 $
$ v $	
$ l \ll u $	$= l + u $
$ \delta $	
$ \square $	$= 0$
$ \delta \times \tau $	$= 1 + \delta + \tau $
$ \tau \times \delta $	$= 1 + \tau + \delta $
$ \psi $	
$ \top $	$= 1$
$ \perp $	$= 1$
$ V $	$= 1$
$ \psi_1 \times \psi_2 $	$= 1 + \psi_1 + \psi_2 $
$ N\{v_1, \dots, v_n\} $	$= 1 + v_1 + \dots + v_n $
$ \psi_1 \cup \psi_2 $	$= 1 + \psi_1 + \psi_2 $
$ \exists l < V < u. \psi $	$= 1 + l + u + \psi $
$ \Gamma $	
$ \cdot $	$= 0$
$ \Gamma, l < V < u $	$= \Gamma + l + u $
$ \zeta $	
$ \square $	$= 0$
$ \zeta \times \psi $	$= 1 + \zeta + \psi $
$ \psi \times \zeta $	$= 1 + \psi + \zeta $

Figure 17: Syntactic size

measure of a variable includes the measures of its bounds. The definition of measure for restricted existential variables v reflects the fact that $N\{v_1, \dots, v_n\}$ represents both invariant constructors and restricted existential types. When v_i represents a single type τ_i , i.e. $v_i = \tau_i \ll \tau_i$, its measure is simply the measure of the type τ_i in $N\{\dots, \tau_i, \dots\}$. Otherwise, $v_i = l_i \ll u_i$ represents an existential type with a single occurrence of the bound variable, $\exists l_i < V_i < u_i. N\{\dots, V_i, \dots\}$, and measures accordingly, comprising both the binding and its single occurrence.

Note that the measure function itself always terminates and evaluates to a positive integer. This is the case because for every recursive call $\mathcal{M}(\Gamma'; \tau')$ of $\mathcal{M}(\Gamma; \tau)$, the combined syntactic size of the arguments $|\tau'| + |\Gamma'|$ is strictly smaller than $|\tau| + |\Gamma|$. The same is true for recursive calls $\mathcal{M}(\Gamma'; \psi')$ of $\mathcal{M}(\Gamma; \psi)$. The syntactic size is defined in Figure 17.

In the remainder of this chapter, the following facts about the distributivity contexts will be used implicitly (proofs are straightforward by induction):

- $\delta[\tau] = \tau'$ for some τ' ;
- $\delta[\delta'] = \delta''$ for some δ'' ;
- $\delta[\delta'[\tau]] = (\delta[\delta'])[\tau]$;
- $\Gamma \vdash \delta <: \delta'$ and $\Gamma \vdash \tau <: \tau' \implies \Gamma \vdash \delta[\tau] <: \delta'[\tau']$;
- $|\delta[\tau]| = |\delta| + |\tau|$;
- $|\zeta[\psi]| = |\zeta| + |\psi|$;
- $\mathcal{M}(\Gamma; \delta[\tau]) = \mathcal{M}(\Gamma; \delta) + \mathcal{M}(\Gamma; \tau)$;
- $\mathcal{M}(\Gamma; \zeta[\psi]) = \mathcal{M}(\Gamma; \zeta) + \mathcal{M}(\Gamma; \psi)$;
- $\rho(\delta[\tau]) = \rho(\delta)[\rho(\tau)]$.

This section presents the following key properties:

- termination of $\Gamma \vdash \tau <: \tau'$ (Lemma 1);
- termination of $\tau \sqcap_{\Gamma} \tau'$ (Lemma 2);
- termination of $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$ (Lemma 3);
- termination of $\mathbf{Solve}(\Gamma; \Delta; K)$ (Lemma 4);
- termination of $\Gamma \mid \Delta \vdash \psi <: \psi'$ (Theorem 1).

Lemma 1 (Termination of $\Gamma \vdash \tau <: \tau'$). *The subtyping algorithm built from the rules of subtyping of types $\Gamma \vdash \tau <: \tau'$ terminates.*

Proof. It follows from the fact that, for each subtyping rule, the measure of each premise $\Gamma \vdash \tau_p <: \tau'_p$ is strictly smaller than the measure of the conclusion $\Gamma \vdash \tau <: \tau'$, i.e.

$$\mathcal{M}(\Gamma; \tau_p) + \mathcal{M}(\Gamma; \tau'_p) < \mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \tau').$$

For example, in the case ST-VARLEFT,

$$\mathcal{M}(\Gamma; \delta[u]) + \mathcal{M}(\Gamma; \tau') < \mathcal{M}(\Gamma; \delta[V]) + \mathcal{M}(\Gamma; \tau')$$

because

$$\mathcal{M}(\Gamma; u) < \mathcal{M}(\Gamma; V) = 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u). \quad \square$$

Lemma 2 (Termination of intersection \sqcap_{Γ}). $\forall \Gamma, \tau, \tau', \tau \sqcap_{\Gamma} \tau'$ terminates.

Proof. $\tau \sqcap_{\Gamma} \tau'$ terminates because subtyping of types terminates, and, for each recursive call $\tau_r \sqcap_{\Gamma} \tau'_r$ of $\tau \sqcap_{\Gamma} \tau'$, the measure $\mathcal{M}(\Gamma; \tau_r) + \mathcal{M}(\Gamma; \tau'_r)$ is strictly smaller than $\mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \tau')$. \square

Lemma 3 (Termination of $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$). *The subtyping algorithm built from the rules of constrained subtyping of types $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$ terminates.*

Proof. Similarly to the previous theorem, the measure decreases, i.e.

$$\mathcal{M}(\Gamma; \tau_p) + \mathcal{M}(\Gamma; \tau'_p) < \mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \tau')$$

for each premise $\Gamma \mid H \vdash \tau_p < \tau'_p \rightsquigarrow K$ of the conclusion $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$.

The only interesting case is SC-UVAR-UNIONRIGHT. By the variable name convention (Definition 1), $\alpha \notin \Gamma$. Therefore, $\mathcal{M}(\Gamma; \alpha) = \mathcal{M}(\Gamma; \alpha_1) = \mathcal{M}(\Gamma; \alpha_2)$ and the measure of the left-hand side type is the same in both premises and the conclusion, while the measure of the right-hand side type in both premises is strictly smaller than in the conclusion. Furthermore, $\prod_{i=1}^n u_1^i$ and $\prod_{j=1}^m u_2^j$ terminate by Lemma 2. \square

Lemma 4 (Termination of $\text{Solve}(\Gamma; \Delta; K)$). *The constraint resolution algorithm $\text{Solve}(\Gamma; \Delta; K)$ terminates.*

Proof. It follows from the fact that:

1. the subtyping algorithms $\Gamma \vdash \tau <: \tau'$ and $\Gamma \mid H \vdash \tau < \tau' \rightsquigarrow K$ used to check the consistency of constraints terminate;
2. the argument Δ of the only recursive call to **Solve** is strictly smaller than that of the original call. \square

Lemma 5 (Context weakening in \mathcal{M}). *The measure of a type signature does not change if the environment is extended (in any position) with a variable that occurs neither in the signature nor in the environment, i.e., $\forall \psi, \Gamma, \Gamma'. \forall l <: V <: u$ with $\neg \text{occ}(V; \psi)$ and $\neg \text{occ}(V; \Gamma)$ and $\neg \text{occ}(V; \Gamma')$.*

$$\mathcal{M}(\Gamma \uparrow \uparrow \Gamma'; \psi) = \mathcal{M}(\Gamma, l <: V <: u \uparrow \uparrow \Gamma'; \psi),$$

where $\Gamma \uparrow \uparrow \Gamma''$ denotes the concatenation of lists, and occ is defined in Figure 18.

Proof. By strong induction on $n = |\Gamma| + |\Gamma'| + |\psi|$. For details, see Lemma 13 on page 87. \square

Theorem 1 (Termination of $\Gamma \mid \Delta \vdash \psi <: \psi'$). *The subtyping algorithm built from the rules of subtyping of type signatures $\Gamma \mid \Delta \vdash \psi <: \psi'$ terminates.*

$occ(V; \tau)$	
$occ(V; \top)$	$= \text{false}$
$occ(V; \perp)$	$= \text{false}$
$occ(V; V)$	$= \text{true}$
$occ(V; V')$	$= \text{false}$
$occ(V; \tau_1 \times \tau_2)$	$= occ(V; \tau_1) \vee occ(V; \tau_2)$
$occ(V; N\{v_1, \dots, v_n\})$	$= occ(V; v_1) \vee \dots \vee occ(V; v_n)$
$occ(V; \tau_1 \cup \tau_2)$	$= occ(V; \tau_1) \vee occ(V; \tau_2)$
$occ(V; v)$	
$occ(V; l \ll u)$	$= occ(V; l) \vee occ(V; u)$
$occ(V; \psi)$	
$occ(V; \top)$	$= \text{false}$
\dots	
$occ(V; \exists l < V <: u. \psi)$	$= \text{true}$
$occ(V; \exists l < V' <: u. \psi)$	$= occ(V; l) \vee occ(V; u) \vee occ(V; \psi)$
$occ(V; \Gamma)$	
$occ(V; \cdot)$	$= \text{false}$
$occ(V; \Gamma, l <: V <: u)$	$= \text{true}$
$occ(V; \Gamma, l <: V' <: u)$	$= occ(V; \Gamma) \vee occ(V; l) \vee occ(V; u)$

Figure 18: Occurrence of a variable

Proof. It follows from the fact that, for each subtyping rule, the measure of each premise $\Gamma \mid \Delta \vdash \psi_p <: \psi_p'$ is strictly smaller than the measure of the conclusion $\Gamma \mid \Delta \vdash \psi <: \psi'$, i.e.

$$\mathcal{M}(\Gamma'; \psi_p) + \mathcal{M}(\Gamma' \multimap \Delta'; \psi_p') < \mathcal{M}(\Gamma; \psi) + \mathcal{M}(\Gamma \multimap \Delta; \psi').$$

Most of the cases are similar to the cases of Lemma 1 on the termination of $\Gamma \vdash \tau <: \tau'$. The remaining cases are:

- **SS-INVLEFT.** Since X is a fresh variable, by Lemma 5 (weakening), $\mathcal{M}(\Gamma, l <: X <: u; \zeta[N\{\dots\}]) = \mathcal{M}(\Gamma; \zeta[N\{\dots\}])$, and also $\mathcal{M}(\Gamma, l <: X <: u \multimap \Delta; \psi') = \mathcal{M}(\Gamma \multimap \Delta; \psi')$.

By the definition of \mathcal{M} ,

$$\begin{aligned} & \mathcal{M}(\Gamma, l <: X <: u; X) \\ &= 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u) \\ &< 2 \times (1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u)) \\ &= \mathcal{M}(\Gamma; l \ll u), \end{aligned}$$

- **SS-EXISTLEFT.** By the variables name convention (Definition 1), X is different from variables in Γ, Δ , as well as the bound variables of ζ, ψ , and ψ' . Therefore, by Lemma 5 (weakening),

$\mathcal{M}(\Gamma, l <: X <: u; \zeta) = \mathcal{M}(\Gamma; \zeta)$, and similarly for ψ' . By the definition of \mathcal{M} ,

$$\mathcal{M}(\Gamma; \exists l <: X <: u. \psi) = 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u) + \mathcal{M}(\Gamma, l <: X <: u; \psi),$$

which is strictly larger than $\mathcal{M}(\Gamma, l <: X <: u; \psi)$ in the premise.

- SS-EXISTRIGHT. Similarly to SS-EXISTLEFT.
- SS-TYPES. The first premise, $\Gamma \mid \text{dom}(\Delta) \vdash \tau <_{\bullet} \tau' \leadsto K$, terminates by Lemma 3. Since the constraint resolution **Solve**($\Gamma; \Delta; K$) terminates by Lemma 4, the entire step also terminates. \square

4.2.2 Unification-Free Subtyping

This section presents the following key properties:

- reflexivity $\Gamma \vdash \tau <: \tau$ (Theorem 2);
- transitivity $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3 \implies \Gamma \vdash \tau_1 <: \tau_3$ (Theorem 3);
- soundness with respect to substitution (Theorem 4).

Theorem 2 (Reflexivity of subtyping of types). $\forall \tau, \Gamma$, with $\Gamma \vdash \tau$.

$$\Gamma \vdash \tau <: \tau.$$

Proof. By induction on the structure of τ .

- Case \top by ST-TOP.
- Case \perp by ST-BOT.
- Case V by ST-VARREFL.
- Case $\tau_1 \times \tau_2$ by IH and ST-TUPLE.
- Case $N\{v_1, \dots, v_n\}$ by IH on l_i, u_i , and ST-INV.
- Case $\tau_1 \cup \tau_2$ by IH, ST-UNIONRIGHT, and ST-UNIONLEFT. \square

Lemma 6 (Subtyping of \perp implies arbitrary subtyping).

$$\forall \tau, \delta_{\perp}, \Gamma. \quad \Gamma \vdash \tau <: \delta_{\perp}[\perp] \implies (\forall \tau', \delta'. \quad \Gamma \vdash \delta'[\tau] <: \tau').$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta_{\perp}[\perp]$. For details, see Lemma 14 on page 88. \square

Lemma 7 (Subtyping of inner union on the right). $\forall \tau, \delta', \tau'_1, \tau'_2, \Gamma$, with $\vdash \Gamma$ and $\Gamma \vdash \tau, \delta', \tau'_1, \tau'_2$.

$$\begin{aligned} \Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2] \\ \implies \\ (\forall \delta_1, \delta_2, \text{ with } \Gamma \vdash \delta_1, \delta_2 \text{ and } \Gamma \vdash \delta_1 <: \delta_2. \quad \Gamma \vdash \delta_1[\tau] <: \delta_2[\delta'[\tau'_1]] \cup \delta_2[\delta'[\tau'_2]]). \end{aligned}$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2]$. For details, see Lemma 15 on page 89. \square

Lemma 8 (Adding inner union on the right). $\forall \tau, \delta', \tau', \Gamma$, with $\vdash \Gamma$ and $\Gamma \vdash \tau, \delta', \tau'$.

$$\Gamma \vdash \tau <: \delta'[\tau'] \implies (\forall \tau''. \Gamma \vdash \tau <: \delta'[\tau' \cup \tau'']).$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta'[\tau']$. For details, see Lemma 16 on page 89. \square

Lemma 9 (Subtyping of union on the right). $\forall \tau, \delta', \tau'_1, \tau'_2, \Gamma$, with $\vdash \Gamma$ and $\Gamma \vdash \tau, \delta', \tau'_1, \tau'_2$.

$$\Gamma \vdash \tau <: \delta'[\tau'_1] \cup \delta'[\tau'_2] \implies \Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2].$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta'[\tau'_1] \cup \delta'[\tau'_2]$. Four cases are possible: ST-BOT, ST-VARLEFT, ST-UNIONLEFT, and ST-UNIONRIGHT. The first three are analogous to the cases of Lemma 7 (subtyping inner union on the right): the form of the rule, IH, constructor. The remaining case is ST-UNIONRIGHT, subcase $i = 1$. By the form of the rule, $\Gamma \vdash \tau <: \delta'[\tau'_1]$. Thus, the case concludes by Lemma 8 (adding inner union on the right): $\Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2]$. \square

Lemma 10 (Context weakening in subtyping of types). $\forall \Gamma, \tau, \tau'. \forall \Gamma'$ with $\text{dom}(\Gamma') \cap \text{dom}(\Gamma) = \emptyset$ and $(\forall V \in \text{dom}(\Gamma'). \neg \text{occ}(V; \tau) \text{ and } \neg \text{occ}(V; \tau'))$,

$$\Gamma \vdash \tau <: \tau' \implies \Gamma \uparrow \Gamma' \vdash \tau <: \tau'.$$

Proof. Straightforward induction on the derivation of $\Gamma \vdash \tau <: \tau'$. \square

Lemma 11 (Subtyping of upper bound). $\forall \Gamma, \tau, \delta, V$, with $l <: V <: u \in \Gamma$, $\vdash \Gamma$, $\Gamma \vdash \tau$, and $\Gamma \vdash \delta[V]$, if

1. $\forall \tau_0$ with $\Gamma \vdash \tau_0$ and $\mathcal{M}(\Gamma; \tau_0) \leq \mathcal{M}(\Gamma; \tau)$,
 $\Gamma \vdash \tau_0 <: l \implies \Gamma \vdash \tau_0 <: u$
2. $\Gamma \vdash \tau <: \delta[V]$

then $\Gamma \vdash \tau <: \delta[u]$.

Proof. Suppose $\Gamma \vdash \tau_{inner}, \tau_{inner}'$ and $\Gamma \vdash \tau_{inner} <: \tau_{inner}'$. We show by induction on this derivation that, for all δ_{inner} , if

1. $\mathcal{M}(\Gamma; \tau_{inner}) \leq \mathcal{M}(\Gamma; \tau)$ and
2. $\tau_{inner}' = \delta_{inner}[V]$,

then $\Gamma \vdash \tau_{inner} <: \delta_{inner}[u]$. The most interesting case of the proof is highlighted in **bold**.

- Case ST-BOT. By ST-BOT.
- Case ST-VARREFL. $\Gamma \vdash V <: V$. By Theorem 2 (reflexivity), $\Gamma \vdash u <: u$. Thus, by ST-VARLEFT, $\Gamma \vdash V <: u$.
- Case ST-VARLEFT. $\Gamma \vdash \delta'[V'] <: \delta_{inner}[V]$. By the form of the rule, $l' <: V' <: u' \in \Gamma$ and $\Gamma \vdash \delta'[u'] <: \delta_{inner}[V]$. Since $\mathcal{M}(\Gamma; \delta'[u']) < \mathcal{M}(\Gamma; \delta'[V'])$, by transitivity of \leq on natural numbers, we have $\mathcal{M}(\Gamma; \delta'[u']) \leq \mathcal{M}(\Gamma; \tau)$. Therefore, the IH is applicable, providing $\Gamma \vdash \delta'[u'] <: \delta_{inner}[u]$. Thus, by ST-VARLEFT, $\Gamma \vdash \delta'[V'] <: \delta_{inner}[u]$.
- **Case ST-VarRight.** $\Gamma \vdash \tau_{inner} <: V$. By the form of the rule, $l <: V <: u \in \Gamma$ and $\Gamma \vdash \tau_{inner} <: l$. Since by assumption, $\mathcal{M}(\Gamma; \tau_{inner}) \leq \mathcal{M}(\Gamma; \tau)$, the first assumption of the lemma is applicable, providing $\Gamma \vdash \tau_{inner} <: u$.
- Case ST-TUPLE, subcase $\delta_{inner} = \delta' \times \tau_{22}$, i.e. $\tau_{inner} = \tau_{11} \times \tau_{12}$ and $\Gamma \vdash \tau_{11} \times \tau_{12} <: \delta'[V] \times \tau_{22}$ ($\delta_{inner} = \square$ is not possible, and $\delta_{inner} = \tau_{21} \times \delta'$ is proved analogously). By the form of the rule, $\Gamma \vdash \tau_{11} <: \delta'[V]$ and $\Gamma \vdash \tau_{12} <: \tau_{22}$. Since $\mathcal{M}(\Gamma; \tau_{11}) < \mathcal{M}(\Gamma; \tau_{inner})$, by transitivity of \leq on natural numbers, we have $\mathcal{M}(\Gamma; \tau_{11}) \leq \mathcal{M}(\Gamma; \tau)$. Thus, by IH, $\Gamma \vdash \tau_{11} <: \delta'[u]$. Then, by ST-TUPLE, $\Gamma \vdash \tau_{11} \times \tau_{12} <: \delta'[u] \times \tau_{22}$.
- Case ST-UNIONLEFT is proved analogously to ST-TUPLE: by the form of the rule, IH, and ST-UNIONLEFT.

The remaining cases (ST-TOP, ST-TUPLE, ST-INV, ST-UNIONRIGHT) are not possible. The conclusion of the lemma follows by instantiating τ_{inner} and δ_{inner} with τ and δ . \square

Theorem 3 (Transitivity of subtyping of types). $\forall \tau_1, \tau_2, \tau_3, \Gamma$,
with $\vdash \Gamma$ and $\Gamma \vdash \tau_1, \tau_2, \tau_3$.

$$\Gamma \vdash \tau_1 <: \tau_2 \text{ and } \Gamma \vdash \tau_2 <: \tau_3 \implies \Gamma \vdash \tau_1 <: \tau_3.$$

Proof. By strong induction on $n = \mathcal{M}(\Gamma; \tau_1) + 2 \times \mathcal{M}(\Gamma; \tau_2) + \mathcal{M}(\Gamma; \tau_3)$. The summand $2 \times \mathcal{M}(\Gamma; \tau_2)$ accounts for the fact that τ_2 occurs twice,

in the derivation of $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$; this fact is needed in the case ST-VARLEFT of $\Gamma \vdash \tau_2 <: \tau_3$.

Cases $n = 1..3$ are not possible as the minimal measure of a type is 1. In the inductive step for n , the induction hypothesis (IH) states that $\forall n' < n. \forall \tau'_1, \tau'_2, \tau'_3$ with $n' = \mathcal{M}(\Gamma; \tau'_1) + 2 \times \mathcal{M}(\Gamma; \tau'_2) + \mathcal{M}(\Gamma; \tau'_3)$, it holds that

$$\Gamma \vdash \tau'_1 <: \tau'_2 \text{ and } \Gamma \vdash \tau'_2 <: \tau'_3 \implies \Gamma \vdash \tau'_1 <: \tau'_3.$$

Proceed by case analysis on $\Gamma \vdash \tau_2 <: \tau_3$ (the most interesting cases are highlighted in **bold**).

- Case ST-TOP $\Gamma \vdash \tau_2 <: \top$ where $\tau_3 = \top$ concludes by ST-TOP: $\Gamma \vdash \tau_1 <: \top$.
- Case SS-BOT $\Gamma \vdash \delta[\perp] <: \tau_3$ where $\tau_2 = \delta[\perp]$. The case concludes by Lemma 6 (subtyping of \perp) applied to the assumption $\Gamma \vdash \tau_1 <: \delta[\perp]$ with $\delta' = \square, \tau' = \tau_3$: $\Gamma \vdash \tau_1 <: \tau_3$.
- Case ST-VARREFL $\Gamma \vdash V <: V$. Since $\tau_2 = \tau_3 = V$, the case concludes by the assumption $\Gamma \vdash \tau_1 <: V$.
- **Case ST-VarLeft** $\Gamma \vdash \delta[V] <: \tau_3$ where $\tau_2 = \delta[V]$. By the form of the rule, $l <: V <: u \in \Gamma$ and $\Gamma \vdash \delta[u] <: \tau_3$. By assumption, $\Gamma \vdash \tau_1 <: \delta[V]$.

First, we will use Lemma 11 (subtyping of upper bound) to derive $\Gamma \vdash \tau_1 <: \delta[u]$. To use the lemma, we need to show that $\forall \tau'$ with $\mathcal{M}(\Gamma; \tau') \leq \mathcal{M}(\Gamma; \tau_1)$. $\Gamma \vdash \tau' <: l \implies \Gamma \vdash \tau' <: u$. Since τ_2 contains V , $\mathcal{M}(\Gamma; V) \leq \mathcal{M}(\Gamma; \tau_2)$, and $\mathcal{M}(\Gamma; V) = 1 + \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u)$, we have $\mathcal{M}(\Gamma; \tau_1) + 2 \times \mathcal{M}(\Gamma; l) + \mathcal{M}(\Gamma; u) < \mathcal{M}(\Gamma; \tau_1) + 2 \times \mathcal{M}(\Gamma; \tau_2) + \mathcal{M}(\Gamma; \tau_3)$. Thus, the IH is applicable to $\Gamma \vdash \tau' <: l$ and $\Gamma \vdash l <: u$ for any τ' with $\mathcal{M}(\Gamma; \tau') \leq \mathcal{M}(\Gamma; \tau_1)$.

Finally, since $\tau_2 = \delta[V]$ and $\mathcal{M}(\Gamma; u) < \mathcal{M}(\Gamma; V)$, the IH is applicable to $\Gamma \vdash \tau_1 <: \delta[u]$ and $\Gamma \vdash \delta[u] <: \tau_3$, which concludes the case with $\Gamma \vdash \tau_1 <: \tau_3$.

- Case ST-VARRIGHT $\Gamma \vdash \tau_2 <: V$. By the form of the rule, $l <: V <: u \in \Gamma$ and $\Gamma \vdash \tau_2 <: l$. Since $\mathcal{M}(\Gamma; l) < \mathcal{M}(\Gamma; V)$, by IH, $\Gamma \vdash \tau_1 <: l$. Thus, by ST-VARRIGHT, $\Gamma \vdash \tau_1 <: V$.
- Case ST-TUPLE $\Gamma \vdash \tau_{21} \times \tau_{22} <: \tau_{31} \times \tau_{32}$ where $\tau_i = \tau_{i1} \times \tau_{i2}$. Case analysis on $\Gamma \vdash \tau_1 <: \tau_{21} \times \tau_{22}$.
 - Case ST-BOT by ST-BOT.
 - Case ST-VARLEFT by the form of the rule, IH on $\Gamma \vdash \delta[u] <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, and ST-VARLEFT on $\Gamma \vdash \delta[u] <: \tau_3$.

- Case ST-TUPLE by the form of the rule, IH on $\Gamma \vdash \tau_{1j} <: \tau_{2j}$ and $\Gamma \vdash \tau_{2j} <: \tau_{3j}$, and ST-TUPLE.
- Case ST-UNIONLEFT $\Gamma \vdash \delta[\tau_{11} \cup \tau_{12}] <: \tau_2$ by the form of the rule, IH on $\Gamma \vdash \delta[\tau_{1j}] <: \tau_2$ and $\Gamma \vdash \tau_2 <: \tau_3$, and ST-UNIONLEFT.

The remaining cases (ST-TOP, ST-VARREFL, ST-VARRIGHT, ST-INV, ST-UNIONRIGHT) are not possible.

- Case ST-INV is proved similarly to ST-TUPLE, with possible cases of $\Gamma \vdash \tau_1 <: \tau_2$ being ST-BOT, ST-VARLEFT, ST-INV, and ST-UNIONLEFT.
- **Case ST-UnionLeft** $\Gamma \vdash \delta[\tau_{21} \cup \tau_{22}] <: \tau_3$. By Lemma 7 applied to $\Gamma \vdash \tau_1 <: \delta[\tau_{21} \cup \tau_{22}]$ with $\delta_1 = \square, \delta_2 = \square, \Gamma \vdash \tau_1 <: \delta[\tau_{21}] \cup \delta[\tau_{22}]$. Case analysis on the latter.
 - Case ST-BOT by ST-BOT.
 - Case ST-VARLEFT where $\tau_1 = \delta'[V]$. By the form of the rule, $l <: V <: u \in \Gamma$ and $\Gamma \vdash \delta'[u] <: \delta[\tau_{21}] \cup \delta[\tau_{22}]$. By Lemma 9 applied to the latter, $\Gamma \vdash \delta'[u] <: \delta[\tau_{21} \cup \tau_{22}]$.
Since $\mathcal{M}(\Gamma; u) < \mathcal{M}(\Gamma; V)$, by IH, $\Gamma \vdash \delta'[u] <: \tau_3$. Thus, the case concludes by ST-VARLEFT: $\Gamma \vdash \delta'[V] <: \tau_3$.
 - Case ST-UNIONLEFT similarly to ST-VARLEFT.
 - Case ST-UNIONRIGHT, subcase $i = 1$. By the form of the rule, $\Gamma \vdash \tau_1 <: \delta[\tau_{21}]$. By the form of the rule of the outer case assumption $\Gamma \vdash \delta[\tau_{21} \cup \tau_{22}] <: \tau_3$, $\Gamma \vdash \delta[\tau_{21}] <: \tau_3$. Since $\mathcal{M}(\Gamma; \delta[\tau_{21}]) < \mathcal{M}(\Gamma; \delta[\tau_{21} \cup \tau_{22}])$, by IH, $\Gamma \vdash \tau_1 <: \tau_3$.

The remaining cases (ST-TOP, ST-VARREFL, ST-VARRIGHT, ST-TUPLE, ST-INV) are not possible.

- Case ST-UNIONRIGHT $\Gamma \vdash \tau_2 <: \tau_{31} \cup \tau_{32}$ where $\tau_3 = \tau_{31} \cup \tau_{32}$, subcase $i = 1$. By the form of the rule, $\Gamma \vdash \tau_2 <: \tau_{31}$. Since $\mathcal{M}(\Gamma; \tau_{31}) < \mathcal{M}(\Gamma; \tau_{31} \cup \tau_{32})$, by IH, $\Gamma \vdash \tau_1 <: \tau_{31}$. Thus, the case concludes by ST-UNIONRIGHT. \square

Theorem 4 (Soundness of subtyping of types with respect to subsitution).

$\forall \Gamma, \tau, \tau' \text{ with } \vdash \Gamma \text{ and } \Gamma \vdash \tau, \tau'. \forall \Gamma', \rho \text{ with } \vdash \Gamma' \text{ and } \Gamma \models_{\Gamma'} \rho.$

$$\Gamma \vdash \tau <: \tau' \implies \Gamma' \vdash \rho(\tau) <: \rho(\tau').$$

where $\Gamma \models_{\Gamma'} \rho$ is defined in Figure 19.

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \tau'$.

Cases ST-TOP and ST-BOT are straightforward by ST-TOP and ST-BOT, respectively.

$$\begin{array}{c}
\boxed{\Gamma \models_{\Gamma'} \rho} \\
\hline
\frac{\forall l <: V <: u \in \Gamma. \quad \Gamma' \vdash \rho(V) \quad \Gamma' \vdash \rho(l) <: \rho(V) \quad \Gamma' \vdash \rho(V) <: \rho(u)}{\Gamma \models_{\Gamma'} \rho} \\
\\
\boxed{K \models_{\Gamma} \rho} \\
\hline
\frac{\forall \tau \leq \alpha \in K. \Gamma \vdash \tau <: \rho(\alpha) \quad \forall \alpha \leq \tau' \in K. \Gamma \vdash \rho(\alpha) <: \tau'}{K \models_{\Gamma} \rho}
\end{array}$$

Figure 19: Validity of substitution with respect to environment and constraints

Case ST-VARREFL by Theorem 2 (reflexivity): $\Gamma' \vdash \rho(V) <: \rho(V)$.

Cases ST-TUPLE, ST-INV, ST-UNIONLEFT, and ST-UNIONRIGHT are straightforward using the form of the rule, the induction hypothesis, and constructor. For example, consider the case ST-TUPLE $\Gamma \vdash \tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2$. By the form of the rule, $\Gamma \vdash \tau_i <: \tau'_i$. By IH, $\Gamma' \vdash \rho(\tau_i) <: \rho(\tau'_i)$. The case concludes by ST-TUPLE with

$$\Gamma' \vdash \rho(\tau_1) \times \rho(\tau_2) <: \rho(\tau'_1) \times \rho(\tau'_2)$$

and the fact that $\rho(\tau_1 \times \tau_2) = \rho(\tau_1) \times \rho(\tau_2)$.

The remaining cases ST-VARLEFT and ST-VARRIGHT are similar. For example, consider ST-VARLEFT $\Gamma \vdash \delta[V] <: \tau'$. By the form of the rule, $l <: V <: u \in \Gamma$ and $\Gamma \vdash \delta[u] <: \tau'$. By IH, $\Gamma' \vdash \rho(\delta[u]) <: \rho(\tau')$. By the form of the rule of the assumption $\Gamma \models_{\Gamma'} \rho$, we know $\Gamma' \vdash \rho(V) <: \rho(u)$. Since $\rho(\delta[u]) = \rho(\delta)[\rho(u)]$ and $\Gamma' \vdash \rho(\delta) <: \rho(\delta)$ by reflexivity, we have $\Gamma' \vdash \rho(\delta)[\rho(V)] <: \rho(\delta)[\rho(u)]$. The case concludes by Theorem 3 (transitivity) with the middle type $\rho(\delta[u])$:

$$\Gamma' \vdash \rho(\delta[V]) <: \rho(\tau'). \quad \square$$

4.2.3 Intersection

Theorem 5 (Soundness of intersection). $\tau \sqcap_{\Gamma} \tau'$ defines an intersection of τ, τ' , namely: $\forall \Gamma, \tau, \tau'$ s.t. $\vdash \Gamma$ and $\Gamma \vdash \tau, \tau'$. $\tau \sqcap_{\Gamma} \tau'$ is a valid lower bound of τ and τ' , i.e.,

$$\Gamma \vdash \tau \sqcap_{\Gamma} \tau' \quad \text{and} \quad \Gamma \vdash (\tau \sqcap_{\Gamma} \tau') <: \tau \quad \text{and} \quad \Gamma \vdash (\tau \sqcap_{\Gamma} \tau') <: \tau'.$$

Proof. Straightforward by strong induction on $\mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \tau')$.

For example, the case where $\Gamma \vdash \tau <: \tau'$ follows from the assumptions and reflexivity $\Gamma \vdash \tau <: \tau$.

In the case $(\tau_1 \cup \tau_2) \sqcap_{\Gamma} \tau'$, we know by IH that $\tau_1 \sqcap_{\Gamma} \tau'$ and $\tau_2 \sqcap_{\Gamma} \tau'$ are valid lower bounds of τ_1, τ' and τ_2, τ' , respectively. Thus, the case concludes by ST-UNIONLEFT.

In the case $V \sqcap_{\Gamma} \tau'$, we have $\Gamma \vdash l <: V$ by Theorem 2 (reflexivity) and ST-VARRIGHT. Then, by IH, $\Gamma \vdash l \sqcap_{\Gamma} \tau' <: l$. The case concludes by Theorem 3 (transitivity): $\Gamma \vdash l \sqcap_{\Gamma} \tau' <: V$.

The remaining cases for tuples and invariant constructors use the induction hypotheses and constructors ST-TUPLE and ST-INV, respectively, as well as ST-UNIONLEFT in the invariant case.

The catch-all case $\tau \sqcap_{\Gamma} \tau' = \perp$ follows from ST-BOT. \square

4.2.4 Constrained Subtyping

This section presents the following key properties:

- $\Gamma \vdash \tau <: \tau'$ and $\Gamma \mid H \vdash \tau < \tau' \leadsto K$ coincide on unification-free types (Lemma 12);
- soundness of $\Gamma \mid H \vdash \tau < \tau' \leadsto K$ (Theorem 6);
- soundness of **Solve**($\Gamma; \Delta; K$) (Theorem 7).

Lemma 12. *Constrained subtyping coincides with subtyping on unification-free types. $\forall \Gamma, \tau, \tau'$ with $\vdash \Gamma$ and $\Gamma \vdash \tau$ and $\Gamma \vdash \tau'$ the following holds:*

1. $\forall H. \Gamma \mid H \vdash \tau < \tau' \leadsto K \implies K = \emptyset \text{ and } \Gamma \vdash \tau <: \tau';$
2. $\Gamma \vdash \tau <: \tau' \implies \forall H. \Gamma \mid H \vdash \tau < \tau' \leadsto \emptyset.$

Proof. Straightforward by induction on the derivation of:

1. $\Gamma \mid H \vdash \tau < \tau' \leadsto K$ (more precisely, by mutual induction on $\Gamma \mid H \vdash \tau \bullet \leq \tau' \leadsto K$ and $\Gamma \mid H \vdash \tau < \bullet \tau' \leadsto K$);
2. $\Gamma \vdash \tau <: \tau'.$

In the first case, the rules SC-UBOT, SC-UVARLEFT, SC-UVARRIGHT, and SC-UVAR-UNIONRIGHT could not have been used to build the derivation because they refer to a unification variable α from H , and both τ and τ' are valid in Γ alone. All other rules of constrained subtyping have matching subtyping rules.

In the second case, all subtyping rules have matching rules of constrained subtyping. The assumption $\vdash \Gamma$ and Lemma 10 (context weakening) allow for concluding $\Gamma \vdash \delta[u]$ and $\Gamma \vdash l$ to apply the induction hypothesis in the rules SC-VARLEFT/ST-VARLEFT and SC-VARRIGHT/ST-VARRIGHT. \square

Theorem 6 (Soundness of constrained subtyping). $\forall \Gamma, H, \tau, \tau'$ with $\vdash \Gamma$, if

$$\Gamma \mid H \vdash \tau \bullet \tau' \rightsquigarrow K \text{ and } \Gamma \vdash \tau \text{ and } \Gamma \mid H \vdash \tau'$$

or

$$\Gamma \mid H \vdash \tau \bullet \tau' \rightsquigarrow K \text{ and } \Gamma \mid H \vdash \tau \text{ and } \Gamma \vdash \tau',$$

then $\forall \rho$ with $\text{dom}(\rho) \supseteq H$ and $\text{dom}(\rho) \cap \text{dom}(\Gamma) = \emptyset$ and $K \models_{\Gamma} \rho$.

$$\Gamma \vdash \rho(\tau) <: \rho(\tau').$$

Proof. By strong induction on $\mathcal{M}(\Gamma; \tau) + \mathcal{M}(\Gamma; \tau')$.

- Case SC-TOP $\Gamma \mid H \vdash \tau <: \top \rightsquigarrow \emptyset$. By definition, $\rho(\top) = \top$. Thus, the case concludes by ST-TOP: $\Gamma \vdash \rho(\tau) <: \top$.
- Case SC-BOT by ST-BOT, similarly to SC-TOP.
- Case SC-UBOT $\Gamma \mid H \vdash \delta[\alpha] \bullet \tau' \rightsquigarrow \{\alpha \leq \perp\}$. Since $\rho(\delta[\alpha]) = \rho(\delta)[\rho(\alpha)] = \delta'[\rho(\alpha)]$ for some δ' and by assumption, $\Gamma \vdash \rho(\alpha) <: \perp$, we have $\Gamma \vdash \delta'[\rho(\alpha)] <: \delta'[\perp]$. By ST-BOT, $\Gamma \vdash \delta'[\perp] <: \rho(\tau')$. Therefore, the case concludes by transitivity: $\Gamma \vdash \delta'[\rho(\alpha)] <: \rho(\tau')$.
- Case SC-VARREFL by ST-VARREFL, for $\rho(X) = X$.
- Case SC-UVARLEFT $\Gamma \mid H \vdash \alpha \bullet \tau' \rightsquigarrow \{\alpha \leq \tau'\}$ by assumption $\Gamma \vdash \rho(\alpha) <: \tau'$, since $\rho(\tau') = \tau'$ due to $\Gamma \vdash \tau'$.
- Case SC-UVARRIGHT similarly to SC-UVARLEFT.
- Case SC-VARLEFT (SC-VARRIGHT) by the form of the rule, IH, and ST-VARLEFT (ST-VARRIGHT), for $\rho(X) = X$ and $\rho(u) = u$ ($\rho(l) = l$).
- Cases SC-TUPLE, SC-INV, SC-UNIONLEFT, and SC-UNIONRIGHT are all similar: by the form of the rule, IH, and the corresponding subtyping constructor.
- **Case SC-UVar-UnionRight** $\Gamma \mid H \vdash \delta[\alpha] \bullet \tau'_1 \cup \tau'_2 \rightsquigarrow K'_1 \cup K'_2 \cup K'$. Since $\Gamma \vdash \tau'_1 \cup \tau'_2$, we have $\rho(\tau'_1 \cup \tau'_2) = \tau'_1 \cup \tau'_2$. By assumption, $K'_1 \cup K'_2 \cup K' \models_{\Gamma} \rho$. By the form of the rule, $\Gamma \mid H, \alpha_1 \vdash \delta[\alpha_1] \bullet \tau'_1 \rightsquigarrow K_1$ and $\Gamma \mid H, \alpha_2 \vdash \delta[\alpha_2] \bullet \tau'_2 \rightsquigarrow K_2$, where $K_1 = K'_1 \bigcup_{i=1}^n \{\alpha_1 \leq u_1^i\}$ and $K_2 = K'_2 \bigcup_{j=1}^m \{\alpha_2 \leq u_2^j\}$.
Let $\bigcap_{i=1}^n u_1^i$ be denoted with u_1 and $\bigcap_{j=1}^m u_2^j$ with u_2 . By Theorem 5 (soundness of intersection), we have $\Gamma \vdash u_1 <: u_1^i$ and $\Gamma \vdash u_2 <: u_2^j$ for all i, j . Therefore, we can take $\rho_1 = \rho[\alpha_1 \mapsto u_1]$ and $\rho_2 = \rho[\alpha_2 \mapsto u_2]$, and it holds that $K_1 \models_{\Gamma} \rho_1$ and $K_2 \models_{\Gamma} \rho_2$. Therefore, the induction hypotheses are applicable, and we get

$\Gamma \vdash \rho_1(\delta)[u_1] <: \tau_1' (H_1)$ and $\Gamma \vdash \rho_2(\delta)[u_2] <: \tau_2' (H_2)$. By assumption on ρ , we have $\Gamma \vdash \rho(\alpha) <: u_1 \cup u_2$. Therefore,

$$\Gamma \vdash \rho(\delta)[\rho(\alpha)] <: \rho(\delta)[u_1 \cup u_2].$$

Since α_1, α_2 were fresh, we know $\rho(\delta) = \rho_1(\delta) = \rho_2(\delta)$. Therefore, by ST-UNIONLEFT applied to H_1, H_2 , we have

$$\Gamma \vdash \rho(\delta)[u_1 \cup u_2] <: \tau_1' \cup \tau_2'.$$

Finally, the subcase $n \geq 1, m \geq 1$ concludes by transitivity:

$$\Gamma \vdash \rho(\delta)[\rho(\alpha)] <: \tau_1' \cup \tau_2'. \quad \square$$

Theorem 7 (Soundness of constraints resolution). $\forall \Gamma, \Delta$ s.t. $\vdash \Gamma, \Delta$. $\forall K$ s.t. $\forall l \leq \alpha \in K. \Gamma \vdash l$ and $\Delta \vdash \alpha$ and $\forall \alpha \leq u \in K. \Delta \vdash \alpha$ and $\Gamma \vdash u$.

$$\text{Solve}(\Gamma; \Delta; K) = \rho \implies \Delta \models_{\Gamma} \rho \text{ and } K \models_{\Gamma} \rho.$$

where $\Delta \models_{\Gamma} \rho$ and $K \models_{\Gamma} \rho$ are defined in Figure 19.

Proof. By induction on Δ . The base case is trivial ($K = \emptyset$ because $\Delta = \cdot$).

In the inductive step $\Delta, l <: \alpha <: u$, the induction hypothesis applies to the call $\text{Solve}(\Gamma; \Delta; K' \cup_i K_{l_i} \cup_j K_{u_j})$, which returns ρ . Therefore, we know that $\Delta \models_{\Gamma} \rho$ and $K' \cup_i K_{l_i} \cup_j K_{u_j} \models_{\Gamma} \rho$.

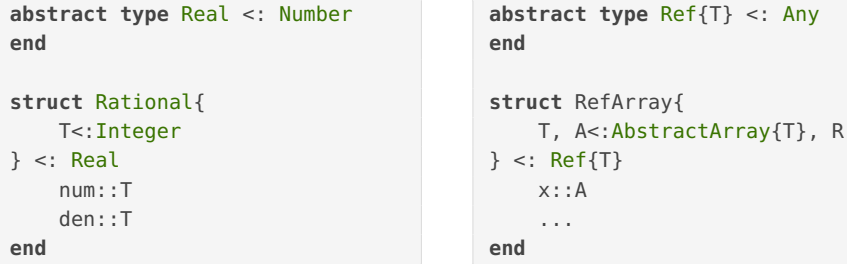
Let $\rho(l) \cup_i l_i$ be denoted with τ_{α} and $\rho[\alpha \mapsto \tau_{\alpha}]$ with ρ_{α} . By Theorem 6 (soundness of constrained subtyping) applied to $\Gamma \mid H \vdash l_i < \bullet u \rightsquigarrow K_{l_i}$ with $K_{l_i} \models_{\Gamma} \rho$ and $\Gamma \mid H \vdash l \bullet \leq u_j \rightsquigarrow K_{u_j}$ with $K_{u_j} \models_{\Gamma} \rho$, we know

$$\Gamma \vdash l_i <: \rho(u) (H_{l_i}) \text{ and } \Gamma \vdash \rho(l) <: u_j (H_{u_j}).$$

By ST-UNIONRIGHT and reflexivity, we know $\Gamma \vdash \rho(l) <: \tau_{\alpha}$. By Theorem 4 (soundness of subtyping with respect to substitution), $\Gamma \vdash \rho(l) <: \rho(u) (H)$, for $\Delta \vdash l <: u$ by the Δ validity assumption. Thus, we have $\Gamma \vdash \tau_{\alpha} <: \rho(u)$ by ST-UNIONLEFT, H , and H_{l_i} . Since $\Delta \models_{\Gamma} \rho$ and α does not occur in Δ , it holds that $\Delta \models_{\Gamma} \rho_{\alpha}$, and we get

$$\Delta, l <: \alpha <: u \models_{\Gamma} \rho_{\alpha}.$$

Finally, we know that $\forall i, j, \Gamma \vdash l_i <: \tau_{\alpha}$ holds by ST-UNIONRIGHT and reflexivity, and $\Gamma \vdash \tau_{\alpha} <: u_j$ holds by ST-UNIONLEFT, H_{u_j} , and $\Gamma \vdash l_i <: u_j$, which means $K_{\alpha} \models_{\Gamma} \rho_{\alpha}$. Since α does not occur in K' , we also know $K' \models_{\Gamma} \rho_{\alpha}$. Thus, $K \models_{\Gamma} \rho_{\alpha}$, which concludes the proof. \square



```

abstract type Real <: Number
end

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

abstract type Ref{T} <: Any
end

struct RefArray{
    T, A<:AbstractArray{T}, R
} <: Ref{T}
    x::A
    ...
end

```

Figure 20: Examples of inheritance in type declarations

4.3 EXTENDED SUBTYPING

In this section, I discuss two Julia features that were omitted from the core language in Figure 8: nominal subtyping (Section 4.3.1) and the diagonal rule (Section 4.3.2).

4.3.1 Nominal subtyping

The Julia language supports a limited form of single-parent inheritance: abstract nominal types can be inherited by both abstract and concrete types, but concrete types are final and cannot be inherited. Type parameters of parametric nominal types are invariant; they can be constrained by non-recursive lower and upper bounds, and can be referenced from the supertype declaration. For instance, `Point{X<:Real}` can be declared a subtype of `AbstractPoint{X}`. Figure 20 provides a few more examples. A type that is being inherited needs to be defined before the inheriting type, and mutually recursive type declarations are not supported.

Because of its fairly restricted nature, inheritance in Julia does not interfere with the decidability of subtyping, unlike, for example, in Java [Tate et al. 2011]. In particular, the lack of variance annotations, recursive inheritance, and recursive constraints prevents properties such as expansive inheritance [Kennedy and Pierce 2007], which are known to cause undecidability. Thus, in Julia, subtyping of nominal types is straightforward: the subtyping algorithm simply walks the finite inheritance hierarchy, substituting type arguments of parametric types. In the specification of Julia subtyping [Zappa Nardelli et al. 2018], this step is encoded with one extra rule `APP_SUPER`, which is given in Figure 21. In the context of the decidability proof from Section 4.2.1, the measure of nominal types needs to be modified to include the measure of the declared supertype, akin to how Greenman et al. [2014] proved decidability of subtyping for Java with Material-Shape Separation.

Without changes, the framework of restricted existential types presented in Section 4.1 is suitable for *partial* handling of inheritance in Julia. For example, consider the following derivation, where $\text{Vector}\{X\}$ is a declared subtype of $\text{AbstractVector}\{X\}$:

$$\frac{\text{Vector}\{X\} \text{ inherits } \text{AbstractVector}\{X\} \quad \frac{\Gamma \vdash \perp <: \perp \quad \Gamma \vdash \text{Int} <: T}{\Gamma \vdash \text{AbstractVector}\{\perp \ll \text{Int}\} <: \text{AbstractVector}\{\perp \ll T\}}}{\Gamma \vdash \text{Vector}\{\perp \ll \text{Int}\} <: \text{AbstractVector}\{\perp \ll T\}}$$

Similarly to APP_INV, subtyping is checked by substituting X with $\perp \ll \text{Int}$ in the supertype declaration $\text{AbstractVector}\{X\}$. However, type parameters do not have to be immediate arguments of the supertype declaration. For example, the following type declaration is allowed:

```
struct ZooVec{X} <: AbstractVector{Zoo{X}}
...
end
```

In this case, simply substituting X with a restricted existential variable would be **incorrect**:

$$\frac{\text{ZooVec}\{X\} \text{ inherits } \text{AbstractVector}\{\text{Zoo}\{X\}\} \quad \Gamma \vdash \text{AbstractVector}\{\text{Zoo}\{\perp \ll \text{Int}\}\} ???}{\Gamma \vdash \text{ZooVec}\{\perp \ll \text{Int}\} \not<: \text{AbstractVector}\{\text{Zoo}\{\perp \ll T\}\}}$$

Recall that $\text{ZooVec}\{\perp \ll \text{Int}\}$ denotes an existential type, namely:

$$\exists \perp <: Y <: \text{Int}. \text{ZooVec}\{Y\}.$$

Therefore, the proper supertype of this type is

$$\exists \perp <: Y <: \text{Int}. \text{AbstractVector}\{\text{Zoo}\{Y\}\},$$

which is different from

$$\text{AbstractVector}\{\exists \perp <: Y <: \text{Int}. \text{Zoo}\{Y\}\}$$

denoted by $\text{AbstractVector}\{\text{Zoo}\{\perp \ll \text{Int}\}\}$.

The simplest solution to this problem would be to disallow instantiating type variables such as X in ZooVec (that is, variables that occur in the supertype but not as immediate arguments) with restricted existentials. Thus, $\text{ZooVec}\{\text{Int}\}$ and $\exists \perp <: X <: \text{Int}. \text{ZooVec}\{X\}$ would be allowed, whereas $\text{ZooVec}\{\perp \ll \text{Int}\}$ would not.

A more general solution would be to open all restricted existential types before walking up the inheritance hierarchy, similar to [Tate 2013]. Thus, the example above would proceed as follows:

$$\boxed{E \vdash t <: t \vdash E}$$

$$\begin{array}{c}
\text{APP_INV} \\
\frac{E_0 = \text{add}(E, \text{Barrier}) \quad \forall 0 < i \leq n. E_{i-1} \vdash t_i <: t'_i \vdash E'_i \wedge E'_i \vdash t'_i <: t_i \vdash E_i}{E \vdash \text{name}\{t_1, \dots, t_n\} <: \text{name}\{t'_1, \dots, t'_n\} \vdash \text{del}(\text{Barrier}, E_n)} \\
\\
\text{APP_SUPER} \\
\frac{\text{name}\{T_1, \dots, T_n\} <: t'' \in \text{tds} \quad E \vdash t''[t_1/T_1, \dots, t_n/T_n] <: t' \vdash E'}{E \vdash \text{name}\{t_1, \dots, t_n\} <: t' \vdash E'}
\end{array}$$

Figure 21: Julia subtyping: nominal types

An excerpt from [Zappa Nardelli et al. 2018], extends Figure 6.

$$\begin{array}{c}
\text{ZooVec}\{X\} \text{ inherits } \text{AbsractVector}\{\text{Zoo}\{X\}\} \\
\frac{\Gamma, \perp <: Y <: \text{Int} \vdash \text{AbsractVector}\{\text{Zoo}\{Y\}\} \not\prec: \text{AbsractVector}\{\text{Zoo}\{\perp \ll T\}\} \quad Y \text{ fresh} \quad \Gamma, \perp <: Y <: \text{Int} \vdash \text{ZooVec}\{Y\} \not\prec: \text{AbsractVector}\{\text{Zoo}\{\perp \ll T\}\}}{\Gamma \vdash \text{ZooVec}\{\perp \ll \text{Int}\} \not\prec: \text{AbsractVector}\{\text{Zoo}\{\perp \ll T\}\}}
\end{array}$$

Additional subtyping rules that support this approach to handling inheritance are given in Figure 22. The rules ST-INV_{SUPER} and SC-INV_{SUPER} are similar to APP_{SUPER}: they walk the inheritance hierarchy, substituting type arguments; importantly, these rules can be applied only when all the arguments of the left-hand side invariant constructor are tight, i.e., τ_i rather than v_i . The rule ST-INV_{LEFT} of unification-free subtyping is similar to SS-INV_{LEFT}: it simply opens a restricted existential type variable. The remaining rules of constrained subtyping are more interesting:

- SC-INV_{LEFT-UL} handles the case when unification variables are on the left. Although the bounds l and u may contain unification variables, the fresh variable X is treated as universal and added to Γ . Since X does not occur in the right-hand side type, unification variables from its bounds cannot “leak” to the right.
- SC-INV_{LEFT-UR} handles the case when unification variables are on the right. The fresh variable X can occur in generated constraints, so it needs to be discharged before the constraints are propagated. The discharge algorithm is given in Figure 23. Intuitively, since X is introduced after unification variables, constraints need to be satisfied for all possible valid instantiations of X . To reflect this, covariant occurrences of X in lower-bound (upper-bound) constraints are replaced with the upper bound (lower bound) of X . If X occurs invariantly, it has to be tightly bound ($\Gamma \vdash u <: l$), or else discharge and subtyping fail.

$$\boxed{\Gamma \vdash \tau <: \tau}$$

$$\text{ST-INVLEFT} \quad \frac{X \text{ fresh} \quad \Gamma, l <: X <: u \vdash N\{\dots, X, \dots\} <: N^l\{v_1^l, \dots, v_m^l\}}{\Gamma \vdash N\{\dots, l \ll u, \dots\} <: N^l\{v_1^l, \dots, v_m^l\}}$$

$$\text{ST-INVSUPER} \quad \frac{N\{Y_1, \dots, Y_n\} \text{ inherits } \tau'' \quad \Gamma \vdash \tau''[Y_1 \mapsto \tau_1, \dots, Y_n \mapsto \tau_n] <: N^l\{v_1^l, \dots, v_m^l\}}{\Gamma \vdash N\{\tau_1, \dots, \tau_n\} <: N^l\{v_1^l, \dots, v_m^l\}}$$

$$\boxed{\Gamma \mid H \vdash \tau \leq \tau \rightsquigarrow K}$$

$$\text{SC-INVLEFT-UL} \quad \frac{X \text{ fresh} \quad \Gamma, l <: X <: u \mid H \vdash N\{\dots, X, \dots\} \bullet \leq N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K}{\Gamma \mid H \vdash N\{\dots, l \ll u, \dots\} \bullet \leq N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K}$$

$$\text{SC-INVLEFT-UR} \quad \frac{X \text{ fresh} \quad \Gamma, l <: X <: u \mid H \vdash N\{\dots, X, \dots\} \leq \bullet N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K \quad K' = \text{Discharge}(\Gamma; l <: X <: u; K)}{\Gamma \mid H \vdash N\{\dots, l \ll u, \dots\} \leq \bullet N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K'}$$

$$\text{SC-INVSUPER} \quad \frac{N\{Y_1, \dots, Y_n\} \text{ inherits } \tau'' \quad \Gamma \mid H \vdash N''\{\tau_1''[Y_1 \mapsto \tau_1, \dots], \dots\} \leq N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K}{\Gamma \mid H \vdash N\{\tau_1, \dots, \tau_n\} \leq N^l\{v_1^l, \dots, v_m^l\} \rightsquigarrow K}$$

Figure 22: Additional subtyping rules to support inheritance

This figure extends unification-free subtyping (Figure 10) and constrained subtyping (Figure 11). Signature subtyping (Figure 13) remains unchanged. **Discharge** is defined in Figure 23.

$$\begin{array}{l}
\text{Discharge}(\Gamma; l <: X <: u; K) \\
\quad \text{if } \Gamma \vdash u <: l \text{ then} \\
\quad \quad \text{return } K[X \mapsto l] \\
\quad \text{else} \\
\quad \quad K_u \leftarrow \{cov_{\mapsto}(X; u; l') \leq \alpha \mid l' \leq \alpha \in K\}; \\
\quad \quad K_l \leftarrow \{\alpha \leq cov_{\mapsto}(X; l; u') \mid \alpha \leq u' \in K\}; \\
\quad \quad \text{return } K_u \cup K_l \\
\quad \text{end}
\end{array}$$

	$cov_{\mapsto}(V; \tau_V; \tau)$
$cov_{\mapsto}(V; \tau_V; \top)$	$= \top$
$cov_{\mapsto}(V; \tau_V; \perp)$	$= \perp$
$cov_{\mapsto}(V; \tau_V; V)$	$= \tau_V$
$cov_{\mapsto}(V; \tau_V; V')$	$= V'$
$cov_{\mapsto}(V; \tau_V; \tau_1 \times \tau_2)$	$= cov_{\mapsto}(V; \tau_V; \tau_1) \times cov_{\mapsto}(V; \tau_V; \tau_2)$
$cov_{\mapsto}(V; \tau_V; N\{\dots\})$	$= \text{if } occ(V; N\{\dots\}) \text{ then fail else } N\{\dots\}$
$cov_{\mapsto}(V; \tau_V; \tau_1 \cup \tau_2)$	$= cov_{\mapsto}(V; \tau_V; \tau_1) \cup cov_{\mapsto}(V; \tau_V; \tau_2)$

Figure 23: Variable discharge algorithm **Discharge**($\Gamma; l <: X <: u; K$)

The following example illustrates SC-INVLEFT-UL:

$$\begin{array}{c}
 \frac{\Gamma, \alpha <: X <: \text{Int} \mid \alpha \vdash \text{Int} \bullet \alpha \rightsquigarrow \{\text{Int} \leq \alpha\}}{\Gamma, \alpha <: X <: \text{Int} \mid \alpha \vdash \text{Int} \bullet X \rightsquigarrow \{\text{Int} \leq \alpha\}} \quad \Gamma, \alpha <: X <: \text{Int} \mid \alpha \vdash X \bullet \ll T \rightsquigarrow \emptyset \\
 \hline
 \Gamma, \alpha <: X <: \text{Int} \mid \alpha \vdash \text{Ref}\{X\} \bullet \ll \text{Ref}\{\text{Int} \ll T\} \rightsquigarrow \{\text{Int} \leq \alpha\} \\
 \hline
 \Gamma \mid \alpha \vdash \text{Ref}\{\alpha \ll \text{Int}\} \bullet \ll \text{Ref}\{\text{Int} \ll T\} \rightsquigarrow \{\text{Int} \leq \alpha\}
 \end{array}$$

The following example illustrates SC-INVLEFT-UR where the variable discharge succeeds:

$$\begin{array}{c}
 \Gamma, \text{Int} <: X <: T \mid \alpha \vdash \perp \bullet \ll X \rightsquigarrow \emptyset \quad \Gamma, \text{Int} <: X <: T \mid \alpha \vdash X \bullet \alpha \rightsquigarrow \{X \leq \alpha\} \\
 \hline
 \Gamma, \text{Int} <: X <: T \mid \alpha \vdash \text{Ref}\{X\} \bullet \ll \text{Ref}\{\perp \ll \alpha\} \rightsquigarrow \{X \leq \alpha\} \\
 \text{Discharge}(\Gamma; \text{Int} <: X <: T; \{X \leq \alpha\}) = \{T \leq \alpha\} \\
 \hline
 \Gamma \mid \alpha \vdash \text{Ref}\{\text{Int} \ll T\} \bullet \ll \text{Ref}\{\perp \ll \alpha\} \rightsquigarrow \{T \leq \alpha\}
 \end{array}$$

The following example illustrates SC-INVLEFT-UR where the variable discharge fails:

$$\begin{array}{c}
 \Gamma, \text{Int} <: X <: T \mid \alpha \vdash \text{AbstractVector}\{\text{Zoo}\{X\}\} \bullet \ll \text{AbstractVector}\{\alpha\} \rightsquigarrow \{\text{Zoo}\{X\} \leq \alpha, \alpha \leq \text{Zoo}\{X\}\} \\
 \hline
 \Gamma, \text{Int} <: X <: T \mid \alpha \vdash \text{ZooVec}\{X\} \bullet \ll \text{AbstractVector}\{\alpha\} \rightsquigarrow \{\text{Zoo}\{X\} \leq \alpha, \alpha \leq \text{Zoo}\{X\}\} \\
 \text{Discharge}(\Gamma; \text{Int} <: X <: T; \{\text{Zoo}\{X\} \leq \alpha, \alpha \leq \text{Zoo}\{X\}\}) = \text{fail} \\
 \hline
 \Gamma \mid \alpha \vdash \text{ZooVec}\{\text{Int} \ll T\} \bullet \ll \text{AbstractVector}\{\alpha\}
 \end{array}$$

The syntax of type declarations and extended validity rules are given in Figure 24. Note that the supertype cannot be a restricted existential type: $A\{\tau_1, \dots, \tau_m\}$ is an invariant constructor with tight type arguments. The validity of types is checked with respect to an implicit datatype table. In a type declaration, type parameters X_i may reference previous parameters: this is handled by checking the validity of type parameters in $\mathcal{T} \vdash td$ using $\vdash \Gamma$ from Figure 15. Similarly to declared bounds of existential variables, bounds of type parameters need to be consistent.

4.3.2 Diagonal Rule

As discussed in Section 3.1, Julia provides special support for a generic programming pattern where method arguments are expected to be of the same concrete type. For example, in the existential type `Tuple{T,T}` where T , type variable T can be instantiated only with concrete types. This is called the **diagonal rule**.

In this work, the diagonal rule is modeled explicitly, using a new kind of type variables—**concrete variables**. In type signatures, existential types specify the kind of the bound variable explicitly:

$$\psi ::= \dots \mid \exists l <: V : \kappa <: u. \psi$$

$$\begin{array}{ll}
N ::= & \text{Invariant constructor name} \\
& | \quad C \quad \text{concrete} \\
& | \quad A \quad \text{abstract} \\
\\
\mathcal{T} ::= & \emptyset \mid \mathcal{T}, td \quad \text{Datatype table} \\
td ::= & N\{l_1 <: X_1 <: u_1, \dots, l_n <: X_n <: u_n\} \text{ inherits } ud \quad \text{Type declaration} \\
ud ::= & T \mid A\{\tau_1, \dots, \tau_m\} \quad \text{Supertype}
\end{array}$$

$$\boxed{\vdash \mathcal{T}}$$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \mathcal{T} \quad \mathcal{T} \vdash td}{\vdash \mathcal{T}, td}$$

$$\boxed{\mathcal{T} \vdash td}$$

$$\frac{\vdash l_1 <: X_1 <: u_1, \dots \quad ud = A\{\dots\} \implies A \text{ is defined in } \mathcal{T} \quad l_1 <: X_1 <: u_1, \dots \vdash ud}{\mathcal{T} \vdash N\{l_1 <: X_1 <: u_1, \dots\} \text{ inherits } ud}$$

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash l \quad \Gamma \vdash u \quad X \text{ fresh} \quad \Gamma, l <: X <: u \vdash N\{\dots, X, \dots\}}{\Gamma \vdash N\{\dots, l <: u, \dots\}}$$

$$\frac{N\{l_1 <: X_1 <: u_1, \dots\} \text{ inherits } ud \quad \Gamma \vdash ud[X_1 \mapsto \tau_1, \dots]}{\Gamma \vdash N\{\tau_1, \dots\}}$$

Figure 24: Validity of type declarations and types in the presence of inheritance (extends Figure 8 and Figure 15)

$$\boxed{\Gamma \vdash \star \tau}$$

$$\frac{l <: V : \star <: u \in \Gamma}{\Gamma \vdash \star V} \quad \frac{}{\Gamma \vdash \star C\{\tau_1, \dots, \tau_n\}} \quad \frac{\Gamma \vdash \star \tau_1 \quad \Gamma \vdash \star \tau_2}{\Gamma \vdash \star \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash \star \tau_1 \quad \Gamma \vdash \star \tau_2 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \star \tau_1 \cup \tau_2} \quad \frac{i \neq j \quad \Gamma \vdash \star \tau_i \quad \Gamma \vdash \tau_j <: \perp}{\Gamma \vdash \star \tau_1 \cup \tau_2}$$

Figure 25: Type concreteness

The kind annotation,

$$\kappa ::= \circ \mid \star,$$

allows for two options:

- kind \circ denotes a regular variable that can be instantiated with any type;
- kind \star denotes a concrete variable that can be instantiated only with a concrete type.

All earlier-presented definitions and reasoning apply to \circ -kinded vars without modifications.

Because concrete variables have to be instantiated with concrete types, the only allowed declared lower bound is \perp . To achieve tight bounds, a concrete type can be chosen for the upper bound. Thus, the validity is extended as follows:

$$\frac{\Gamma \vdash u \quad \Gamma, \perp <: V : \star <: u \vdash \psi}{\Gamma \vdash \exists \perp <: V : \star <: u. \psi}$$

Figure 25 defines concreteness $\Gamma \vdash \star \tau$. Type τ is concrete if it is a concrete variable, a concrete invariant constructor with tight arguments, a tuple of concrete types, a union of equivalent concrete types, or a union of a concrete and a bottom type. The concreteness judgment is used by the subtyping algorithm.

The subtyping algorithm requires minimal modifications, as defined in Figure 26. All previously defined rules are applicable to both kinds of variables. The two new rules cover the tight-bound case for a concrete variable: as the only valid instantiation of the variable is u , it can be used instead of \perp to check subtyping with the concrete variable on the right. The same reasoning explains the new cases for intersection \sqcap_Γ . Constraints for concrete unification variables are generated in the same way as for regular variables. The key difference is in the constraint resolution algorithm: the smallest type satisfying

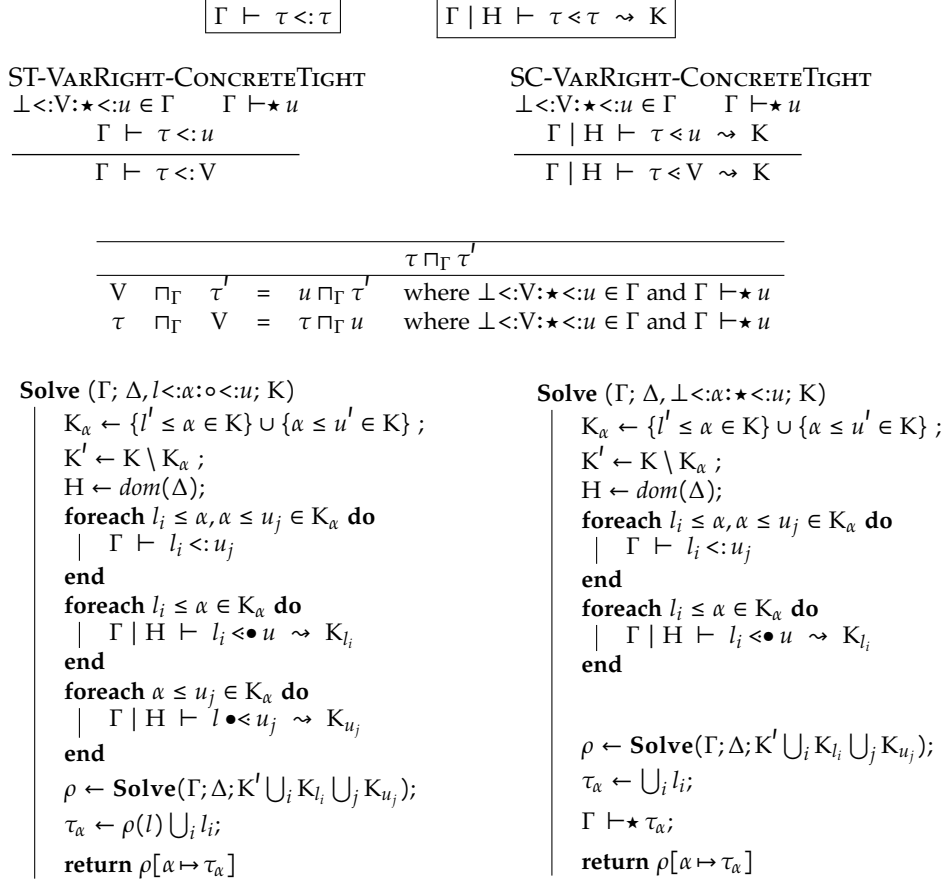


Figure 26: Subtyping with concrete variables

Extends Figure 11 and Figure 12, and replaces Figure 14. Constraints resolution algorithm for regular variables is exactly the same as the algorithm in Figure 14; it is provided here for convenience of comparison.

the constraints, $\tau_{\alpha} = \bigcup_i l_i$, needs to be concrete² in order for constraint resolution to succeed.

² It is possible that there are no lower-bound constraints l_i , in which case instantiating α with \perp would technically be incorrect. However, the absence of generated lower bounds means that any instantiation within the declared bounds would satisfy the constraints, so it can be picked arbitrarily; in this case, the only necessary condition is that $\neg(\Gamma \vdash \rho(u) <: \perp)$, as otherwise, no concrete instantiations would be valid.

EVALUATION OF MIGRATION EFFORT

To estimate the effort required for migrating existing Julia code to a version of Julia with the restriction on existential types, I perform a corpus analysis of all 9K packages from the official Julia package registry. In particular, the analysis extracts type annotations from the source code of the packages and checks how many of those satisfy the restriction described in Chapter 4.

Out of more than 2M successfully extracted, *statically* identifiable type annotations, only two do not have an equivalent type signature that is expressible under the restriction; one more annotation has an expressible semantic¹ equivalent. These results are encouraging and suggest that with the restriction, decidability of subtyping comes at a minimal practical cost to expressiveness.

CORPUS. The chosen corpus of packages is the entire official *General registry*²: this registry is the default source of packages used by Julia programs and contains the majority of all public Julia packages. The list of packages is obtained using the JuliaHub service³: JuliaHub conveniently provides a JSON file⁴, which I process with `JuliaPkgsList.jl`⁵ to extract information about packages and latest registered versions. There were 9355 entries listed on JuliaHub as of 2023-05-20. Out of those, using `JuliaPkgDownloader.jl`⁶, 9315 packages were downloaded successfully. Some JuliaHub entries were not valid registered packages (e.g., the entry for Julia itself), some did not contain a version, and some were no longer publicly available or could not be processed for other reasons.

Thus, the resulting corpus consists of 172K files with 19.5M lines of code across 9K packages (Table 2).

¹ The equivalence is not currently derivable in Julia, but nonetheless, the type can be replaced without affecting the semantics of the analyzed program.

² <https://github.com/JuliaRegistries/General>

³ JuliaHub (<https://juliahub.com/ui/Packages>) is listed on <https://julialang.org/packages/> as one of the services for browsing Julia packages.

⁴ <https://juliahub.com/app/packages/info>

⁵ <https://github.com/julbinb/JuliaPkgsList.jl>

⁶ <https://github.com/julbinb/JuliaPkgDownloader.jl/>

packages	files	lines of code
9 315	172 024	19 476 938

Table 2: Corpus of Julia packages from General registry (May 2023)
Lines of code exclude comments.

5.1 METHODOLOGY

The analysis⁷, written in Julia and executed with Julia 1.8.5, extracts type annotations from all `.jl` files in the corpus and reports annotations that cannot be trivially rewritten into equivalent ones admitted by the restricted type language. To extract type annotations, the analysis relies on the Julia parser and the `MacroTools.jl`⁸ package, which allows for convenient pattern matching over Julia’s abstract syntax tree. The analysis is static and, as such, does not process dynamically generated code.

I identify and extract the following kinds of type annotations:

- method signatures: for example, `Tuple{T, Vector{T}}` where `T` is extracted from method `f(x::T, v::Vector{T}) where T = ...;`
- return type annotations: for example, `Bool` from `f(x) :: Bool = ...;`
- all uses of explicit type assertions `:: t` outside method signatures: these include field type annotations, run-time type assertions, and local variable type annotations.

Type annotations are analyzed as follows:

- For each bound type variable, the analysis records information about the position of the `where`-binding in a type, as well as positions of all occurrences of the variable relative to the binding. Position is defined by a list of constructors. For example, in the type `Tuple{Ref{T}, T, S where S} where T`, variable `T` is bound in `[]` and occurs in `{[Tuple, Ref], [Tuple]}`; `S` is bound in `[Tuple]` and occurs in `{[]}`.
- All variable occurrences are checked against the restriction using the binding position to distinguish between top-level and inner-to-invariant-constructor variables, which need to satisfy different requirements.

Figure 34 provides a list of examples from the test suit of the analysis implementation.

Some types not syntactically representable by the restricted grammar of τ (Figure 8) have an *equivalent* representable counterpart (according

⁷ <https://github.com/prl-julia/julia-sub>

⁸ <https://github.com/FluxML/MacroTools.jl>

to Julia subtyping). Such unrepresentable types are not flagged by the analysis, and they fall into one of the following two categories:

- An existential variable occurs once covariantly, for example, `Tuple{T} where T<:u`: this type is equivalent to `Tuple{u}`, which is a valid τ .
- An existential variable occurs only once in an invariant position, but the occurrence is separated from the binding by covariant constructors, for example, `Tuple{Vector{T}} where T` or `Union{Vector{T}, Missing} where T`: these types are equivalent to `Tuple{Vector{T} where T}` and `Union{Vector{T} where T, Missing}`, respectively, which are valid restricted types τ .

In Julia, the first kind of benign unrepresentable types is already automatically rewritten into the equivalent existential-free form: for example, `Vector{Tuple{T} where T<:Number}` is evaluated to `Vector{Tuple{Number}}` in the REPL.

5.2 RESULTS

The majority of the source code was processed without failures, with 206 packages having at least one file that could not be parsed by Julia (an example of a parsing error is given in Figure 35). In successfully parsed files, the analysis identified a total of 2 136 609 type annotations. Out of these annotations:

- 1 573 were not be processed at all, usually because a type variable binding contained a macro or quoted expression; several examples of that are given in Figure 36;
- 22 396 were partially processed: this happens when all type variable bindings are processed successfully, but a part of the type cannot not be processed due to a macro or quoted expression; in this case, the analysis can miss some occurrences of type variables.

In total, 1629 packages had at least one parsing or processing error.

Of the 2 135 036 statically identifiable, fully-or-partially analyzable type annotations, 2 135 030 (i.e. 99.999%) were identified as satisfying the proposed restriction, and 6 annotations were flagged as potentially being impacted—they are listed in Figure 27.

Three of these six annotations were false positives related to `Vararg`. In Julia, variadic arguments are represented as `Vararg-in-Tuple` types: for example, `Tuple{Vararg{Int}}` stands for a tuple of arbitrarily many integers. According to Julia subtyping, `Vararg` is covariant in its type argument, whereas the analysis reported it as if it were invariant.

```

# false positive, package ForneyLab.jl
# src/algorithms/
#   posterior_factorization.jl
Tuple{
  Vararg{
    Union{
      T, Set{T}, Vector{T}
    } where T <: Variable
  }, Any}
# equivalent to:
Tuple{
  Vararg{
    Union{
      Variable,
      Set{<:Variable},
      Vector{<:Variable}
    }
  }, Any}

# false positive, package Tries.jl
# src/Tries.jl
Tuple{
  Vararg{
    Pair{NTuple{N, K}, T} where N
  }
} where T where K
# equivalent to:
Tuple{
  Vararg{
    Pair{NTuple{N, K}, T}
  } where N
} where T where K

# false positive, package Tries.jl
# src/Tries.jl
Tuple{
  Vararg{
    Pair{NTuple{N, K}, <:Any} where N
  }
} where K
# equivalent to:
Tuple{
  Vararg{
    Pair{NTuple{N, K}, <:Any}
  } where N
} where K

# false positive (semantically)
# package Muon.jl
# src/alignedmapping.jl
Dict{K,
  Union{
    AbstractArray{<:Number},
    AbstractArray{
      Union{Missing, T}
    } where T <: Number,
    DataFrame
  }}
# semantically equivalent to,
# and admitted by the code:
Dict{K,
  Union{
    AbstractArray{<:Number},
    AbstractArray{T} where
      Missing<:T<:Union{Missing, Number},
    DataFrame
  }}

# true positive, package Alicorn.jl
# test/Utils/UtilsTests.jl
Array{
  Tuple{T,
    Array{T, N} where N,
    Bool
  } where T
}
# admitted by the code:
Array{
  Tuple{Any,
    Array{<:Any, <:Any},
    Bool
  }}

# true positive
# package UnitfulEquivalences.jl
# src/UnitfulEquivalences.jl
Tuple{Type{
  Union{
    Quantity{T, D, U},
    Level{L, S,
      Quantity{T, D, U}
    } where {L, S}
  } where {T, U}
}} where D
# cannot be easily rewritten

```

Figure 27: Type annotations flagged by the analysis:
false positives (left) and true positive (right)

Furthermore, one of the remaining three type annotations can be rewritten into a semantically equivalent type that does satisfy our restriction. The flagged type in question—the `Dict` type at the top right of Figure 27—contains an unrepresentable type:

```
AbstractArray{Union{Missing, T}} where T<:Number
```

The above type is semantically equivalent to:

```
AbstractArray{T} where Missing<:T<:Union{Missing, Number}
```

However, In Julia 1.8.5, the former type is a subtype of the latter but not vice versa.

Thus, only two remaining types truly cannot be expressed under the proposed restriction. In the first case,

```
Array{Tuple{T, Array{T, N} where N, Bool} where T}
```

the problem is that variable `T` occurs twice. In the second case, which seemingly has the same problem as above,

```
Tuple{Type{Union{
    Quantity{T, D, U},
    Level{L, S, Quantity{T, D, U}} where {L, S}
} where {T, U}
}} where D
```

the type is equivalent to

```
Tuple{Type{Union{
    Quantity{T, D, U} where {T, U},
    Level{L, S, Quantity{T, D, U}} where {L, S, T, U}
}
}} where D
```

where all variables occur exactly once. However, in the second component of the union, variables `T`, `U` are not bound immediately outside the invariant `Quantity`.

Additionally, I extract and analyze type declarations. There were a total of 141 232 declarations, 239 of which could not be processed and 3 804 were partially processed (similarly to the case of type annotations). Only one type declaration did not satisfy the restriction in its supertype declaration, which was in the same package and with the same type as the false positive `Dict{...Union{T,Missing}...}`.

5.3 MIGRATION STRATEGIES

In general, for cases where an equivalent representable type does not exist, I suggest two migration strategies:

1. Use more permissive types. For example, replace unrepresentable `Ref{Dict{T, T} where T}` with `Ref{Dict{<:Any, <:Any}}`. This approach may require changing type annotations in method signatures as well as code where type-annotated values are created:

```
f(r :: Ref{Dict{T,T} where T}) = ...
r = Ref{Dict{T,T} where T}(...)
f(r)

# transforms to

f(r :: Ref{Dict{<:Any,<:Any}}) = ...
r = Ref{Dict{<:Any,<:Any}}(...)
f(r)
```

If it is necessary that the type arguments of the dictionary are the same, a dynamic check can be added. For instance, if `d` is a dictionary value, the following code ensures that the type parameters are equivalent:

```
typeof(d).parameters[1] == typeof(d).parameters[2]
```

If there is already a method definition for the more permissive type, it needs to be merged with the no-longer-supported definition: to “dispatch” to the right code, a dynamic check akin to the above can be used.

2. Define a wrapper type and use it instead of the unrepresentable one. For example:

```
struct EqDict{T}
  d :: Dict{T,T}
end
```

Then, instead of `Ref{Dict{T, T} where T}`, the representable type `Ref{EqDict{<:Any}}` can be used, but the original data needs to be wrapped.

MIGRATING THE THREE EXAMPLES. For the three examples identified by the analysis (listed on the right of Figure 27 in Chapter 5.2), I performed a manual inspection and attempted migrating the code, which included the following steps:

- run tests;
- insert `@info "inside"` near the impacted type annotation and run tests again to ensure that the affected part of the program is exercised by the tests (macro `@info` simply prints its argument);
- modify the code and run tests again, compare the result.

Two out of the three examples were successfully migrated with a single change—in the problematic type annotation. In both cases, the problematic type annotations were field or type variable annotations, and one of those cases also included the above mentioned problematic supertype declaration.

The first successful migration was done for `Muon.jl` Julia package: Figure 28 depicts relevant code and shows two necessary changes in comments. There, one change is in a supertype declaration, and another is in a field type annotation.

The second successful migration was done for `Alicorn.jl`: Figure 29 depicts relevant code and shows one necessary change in a comment. There, the change is in the type annotation of local variable `examples`.

The final example, depicted in Figure 30, involves `Type` and dispatch on type values (discussed in the first paragraph on page 8). Because the problematic type annotation appears in a method that is used to macro-generate other methods, I could not easily track all the uses of the method and migrate it. The example also suggests that the case of `Type{T}` might need special treatment to allow τ to be instantiated with type signatures ψ rather than τ . To ensure decidability, an incomplete equality relation (discussed on page 26) could likely be used in place of the left-to-right and right-to-left subtyping checks.

Overall, the results of the static analysis of type annotations are encouraging: in practice, the impact of the restriction appears to be limited.

```

struct AlignedMapping{T <: Tuple, K, R} <:
  AbstractAlignedMapping{
    T,
    K,
    Union{
      AbstractArray{<:Number},
      # MIGRATION: replace with
      # AbstractArray{T} where Missing <: T <: Union{Number,T}
      AbstractArray{Union{Missing, T}} where T <: Number,
      AbstractDataFrame,
    },
  }
  ref::R # any type as long as it supports size()
  d::Dict{K,
    Union{
      AbstractArray{<:Number},
      # MIGRATION: replace with
      # AbstractArray{T} where Missing <: T <: Union{Number,T}
      AbstractArray{Union{Missing, T}} where T <: Number,
      DataFrame,
    },
  }
}

function AlignedMapping{T, K}(r, d::AbstractDict{K})
  where {T <: Tuple, K}
  @info "inside AlignedMapping"
  for (k, v) in d
    checkdim(T, v, r, k)
  end
  return new{T, K, typeof(r)}(r, d)
end

```

Figure 28: Muon.jl code fragment with unrepresentable types

```

function _getExamplesFor_isElementOf()
  # MIGRATION: replace with
  # examples::Array{ Tuple{Any, Array, Bool} } = [
  examples::Array{ Tuple{T, Array{T,N} where N, Bool} where T } = [
    ("a", ["a" "b" "c" "d"], true),
    (1, [2 3; 4 5], false),
  ]
  @info "inside _getExamplesFor_isElementOf"
  return examples
end

```

Figure 29: Alicorn.jl code fragment with unrepresentable type

```

# COULD NOT MIGRATE
dimtype(::Type{
    Union{
        Quantity{T,D,U},
        Level{L,S,Quantity{T,D,U}} where {L,S}
    } where {T,U}
}) where D = begin
    @info "inside dimtype"
    typeof(D)
end

macro eqrelation(name, relation)
    ...
    quote
        UnitfulEquivalences.edconvert(
            ::dimtype($(esc(a))), x::$esc(b), ::$(esc(name))
        ) = x * $(esc(rhs))
        UnitfulEquivalences.edconvert(
            ::dimtype($(esc(b))), x::$esc(a), ::$(esc(name))
        ) = x / $(esc(rhs))
        nothing
    end
    ...
end

```

Figure 30: UnitfulEquivalences.jl code fragment with unrepresentable type (could not be migrated easily)

RELATED WORK

Subtyping is typically associated with object-oriented programming languages and static type systems. As a part of a type system, subtyping allows an expression of a more specific type (subtype) to be used in place where an expression of a more general type (supertype) is expected, enabling more programs to be typable. Subtyping can also be used at run time in both statically and dynamically typed languages to perform type tests, casts, and dynamic dispatch.

There is a wide variety of research related to subtyping; this chapter focuses primarily on work that concerns decidability of subtyping, semantic subtyping, and subtyping of Java generics with wildcards, as Julia’s subtype relation was inspired by the latter two [Bezanson 2015]. In particular, the treatment of tuple and union types aligns with the semantic subtyping approach, and bounded existential types are commonly used as a model of wildcards.

6.1 DECIDABILITY OF SUBTYPING

Even in the context of statically typed languages, decidability is a desirable property of static type systems and subtype relations, but establishing decidability can be challenging. For example, the undecidability of checking subtyping for Java generics was established only fairly recently by Grigore [2017], and the previously suspected undecidability of subtyping and type checking for the core calculus of Scala 3, DOT (Dependent Object Calculus) [Amin et al. 2016], was proven by Hu and Lhoták [2019].

Typically, undecidability is shown by a reduction from a known undecidable problem. For example, the halting problem for Turing machines is used to prove the undecidability of Java generics [Grigore 2017] as well as System $F_{<}$ [Pierce 1992]. $F_{<}$ [Cardelli et al. 1991] is a language with bounded universal types, where the undecidability of subtyping is caused by the combination of contravariance and rebounding of type variables in the context. Proofs by reduction from $F_{<}$ are commonly used to establish undecidability in languages with some form of bounded quantification. For instance, the undecidability

of DOT [Hu and Lhoták 2019] is shown by reducing a variant of $F_{<}$ to $D_{<}$, a restriction of DOT featuring type members and path-dependent types. A similar approach is used by Mackay et al. [2019] for $W_{y\vee core}$, another simplification of DOT featuring path-dependent and recursive types. In the case of Julia, the undecidability of subtyping also follows from its power to encode a variant of $F_{<}$, as discussed in Section 3.3.

In practice, the benefits of having a more expressive type system might outweigh the cost of undecidability, which amounts to the compiler crashing or not terminating on some input programs. If the undecidability arises only in rare, contrived cases, being able to successfully type check more benign programs might be preferable to reliably rejecting those programs with a decidable but less expressive type system.

However, when subtyping is used at run time, the cost of undecidability is higher because a crash or non-termination occurs during program execution rather than compilation. In mainstream programming languages that require run-time subtyping checks, run-time types are often more restrictive than static ones, which simplifies the corresponding subtyping problem. For example, in the .NET intermediate language, the decidability of subtyping was shown for ground types, which are used at run time [Kennedy and Pierce 2007]¹. In the case of Java generics, where static subtyping is undecidable, the type erasure mechanism allows for decidable run-time subtyping. In Julia, however, it is the *run-time* subtyping that is undecidable, yet subtyping is heavily used by the dynamic semantics.

Identifying decidable fragments of undecidable type systems and subtype relations remains an important challenge. In some cases, it might be possible to recover decidability by restricting the system in a way that does not affect most practical programs. For example, the material-shape separation for Java generics [Greenman et al. 2014] enables decidability of subtyping by limiting F-bounded polymorphism (i.e., recursive constraints on type variables) to the subset of types, called shapes, that are used exclusively as constraints; this separation appears to be in agreement with an industry-wide practice. Mackay et al. [2019] extend the material-shape separation to the context of path-dependent and recursive types. Earlier, Kennedy and Pierce [2007] identified three decidable fragments of undecidable subtyping in the context of nominal inheritance and variance without F-bounded polymorphism: the fragments can be obtained by restricting either contravariance, or expansive class tables, or multiple instantiation inheritance. Aiming for decidable subtyping, Julia designers deliberately restricted the language to *not* support F-bounded polymorphism,

¹ Due to the lack of documentation, it is not clear whether subtyping between ground types is still decidable in C# as of 2023.

contravariant constructors, and multiple inheritance. Based on those restrictions, [Bezanson \[2015\]](#) conjectured the decidability of subtyping but pointed out that the combination of invariant constructors and contravariance in lower bounds of existential types is akin to the source of undecidability in $F_{<}$; the conjecture relied on the fact that left- and right-hand side variables are treated asymmetrically in Julia. Still, Julia subtyping is powerful enough to encode the undecidable $F_{<}$.

A number of decidable subtype relations, e.g. Wv_{self} [[Mackay et al. 2019](#)] and Kernel $D_{<}$ [[Hu and Lhoták 2019](#)], adopt subtyping rules inspired by a decidable, restricted variant of $F_{<}$ called Kernel $F_{<}$ [[Cardelli and Wegner 1985](#)]. The downside of such relations is their rejecting more subtyping judgments than is desirable in practice. A strictly more powerful than Kernel $D_{<}$ system, Strong Kernel $D_{<}$ [[Hu and Lhoták 2019](#)], is also decidable. This relation splits the type variable context into two parts, for the left- and right-hand sides of the subtyping judgment, thus avoiding the problematic rebounding of type variables in a single context; the same idea applies to $F_{<}$ to obtain the decidable Strong Kernel $F_{<}$. A similar approach with context splitting is used by [Mackay et al. \[2019\]](#) for decidable subtyping in Wv_{fix} . In both Wv_{fix} and Strong Kernel systems, an undesired consequence of maintaining two separate contexts is that the resulting subtype relations lack transitivity. Another decidable variant of $F_{<}$, called $F_{<}^R$, was proposed by [Mackay et al. \[2020\]](#). There, rebounding of type variables is allowed only for type bounds that do not themselves contain bounded quantification, with unrestricted types resorting to the Kernel subtyping rule. With this syntactic restriction, $F_{<}^R$ admits some useful judgments rejected by Kernel $F_{<}$, yet does not lose transitivity.

6.2 SEMANTIC SUBTYPING

In semantic subtyping [[Frisch et al. 2008](#)], types are given a set-theoretic interpretation, and subtyping is defined as the set inclusion on interpretations. For example, [Hosoya et al. \[2000\]](#) define a static type system for an XML processing language and interpret types as sets of valid XML documents. Thus, semantic subtyping provides an intuitive mental model of types to the users. In addition, it automatically satisfies useful properties such as reflexivity and transitivity.

However, semantic subtyping still needs a decision procedure, and that is where the complexity typically comes from, especially if efficiency matters. For example, in the Julia language, semantic subtyping of union and tuple types relies on a space-efficient algorithm that lazily explores the disjunctive normal form [[Chung et al. 2019](#)]. In [[Frisch et al. 2002](#)], which supports not only union, but also intersection and

negation types, subtyping $t \leq s$ is checked via the emptiness test t and $\neg s \approx \perp$ and also relies on the disjunctive normal form. Related to Julia’s treatment of right-hand side existential variables as unification variables, [Castagna et al. \[2015\]](#) generate and solve subtype constraints as a part of type inference process in their set-theoretic framework with union, intersection, and negation types. [Schimpf et al. \[2023\]](#) successfully apply the same framework to typing Erlang. However, their approach to constraint solving does not translate to Julia due to the presence of invariant constructors and impredicativity.

While types such as unions and tuples have a straightforward semantic interpretation, other types that are common for statically typed languages can be challenging to interpret. To this end, [Frisch et al. \[2002\]](#) provide an interpretation for function types, and [Castagna and Xu \[2011\]](#) interpret predicative parametric polymorphism. In the latter, the interpretation needs to be parameterized by semantic assignments of type variables. In the context of object-oriented languages, [Dardha et al. \[2016\]](#) propose an integration of structural types and semantic subtyping, and [Ancona and Corradi \[2016\]](#) study semantic subtyping for imperative languages with mutable fields. In the case of the Julia language, the key challenge is the combination of impredicative bounded existential types and invariant type constructors.

6.3 JAVA WILDCARDS

The wildcard mechanism of Java generics [[Torgersen et al. 2004](#)] provides use-site variance of parametric types [[Krab Thorup and Torgersen 1999](#)]. For example, a variable of type `List<? extends Number>` can be assigned any list with the element type that is a subtype of `Number`; using the variable, elements of the list can be safely read at type `Number`. Use-site variance has been recognized as a restricted form of bounded existential types [[Igarashi and Viroli 2002](#)]. Thus, the wildcard-typed list above represents an existential type $\exists X<: \text{Number}. \text{List}<X>$. There have been multiple formalizations of Java wildcards, such as WildFJ [[Torgersen et al. 2005](#)] and TameFJ [[Cameron et al. 2008](#)], but they were focused on type soundness rather than a decidable subtyping algorithm. [Smith and Cartwright \[2008\]](#) found inconsistencies in Java’s type inference and subtyping algorithms and proposed a solution using a limited form of union types, with a conjecture on the decidability of subtyping.

Subtyping Java wildcards is challenging and, as shown by [Grigore \[2017\]](#), undecidable. [Wehr and Thiemann \[2009\]](#) identify multiple undecidable subtype relations for bounded existential types in formal models inspired by Java. [Tate et al. \[2011\]](#) highlight multiple sources of non-termination in Java subtyping, e.g. the presence of recursive

constraints on type variables and wildcards being allowed in the inheritance hierarchy; by making practical restrictions on the Java language, they provide a terminating subtyping algorithm. Julia lacks all the Java features that were linked to non-termination of subtyping but supports union and existential types, which are not present in Java's surface language.

CONCLUSIONS AND FUTURE WORK

The specification of Julia subtyping (Chapter 3.2) proved essential for revising Julia’s subtyping algorithm, finding bugs, and detecting its undecidability. The specification can still be used by Julia programmers to better understand subtyping and multiple dispatch, although it no longer captures all aspects of subtyping due to changes between Julia versions 0.6.2 and 1.9.2.

Because the undecidability of subtyping affects program execution, finding a decidable subtype relation without sacrificing *too much* expressive power is a desirable direction for evolving the Julia language. The proposed **restriction** and its corresponding **decidable subtyping** (Chapter 4) provide a good candidate for such evolution: **less than 0.01%** of statically identifiable type annotations do not satisfy the restriction, and most of them can be rewritten without affecting program behavior (Chapter 5). Furthermore, the proposed subtype relation can serve as a specification to evaluate the correctness of an implementation of subtyping, should Julia maintainers choose to use this work.

The results of the static corpus analysis are promising, but further evaluation of the restriction is needed. First, some Julia programs generate code with macros and `eval`, and such code is not processed by the static analysis. Second, Julia internally relies on a data-flow analysis to infer types and generate optimized, type-specialized code. Therefore, not only user-defined code can be impacted by the restriction, but multiple components of the compiler, too. Thus, there are several avenues for **future work**:

- Conduct a dynamic corpus analysis to check the restriction for all types encountered during program execution. The analysis should be able to detect restriction violations in both dynamically generated code and the results of type inference.
- Examine the interaction of the restricted type language with relevant parts of the Julia compiler.
- Provide a correct and efficient implementation of the subtyping algorithm from Chapter 4.

Decidable subtyping proposed in this work can be of interest for languages other than Julia, as it extends use-site-variance subtyping with more expressive, top-level existential types. However, the framework I presented should be applied with care: for example, its interaction with more expressive nominal subtyping may not be straightforward.

BIBLIOGRAPHY

- Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2003. An Indexed Model of Impredicative Polymorphism and Mutable References. (2003). <https://www.ccs.neu.edu/home/amal/papers/impred.pdf> (cited on p. 94)
- Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_14 (cited on p. 75)
- Davide Ancona and Andrea Corradi. 2016. Semantic Subtyping for Imperative Object-Oriented Languages. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2016)*. <https://doi.org/10.1145/2983990.2983992> (cited on p. 78)
- Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2023. Decidable Subtyping of Existential Types for Julia. *In submission* (2023). (cited on p. 4)
- Julia Belyakova. 2019. Decidable Tag-based Semantic Subtyping for Nominal Types, Tuples, and Unions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2019)*. Article 3. <https://doi.org/10.1145/3340672.3341115> (cited on pp. 4 and 91)
- Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). <https://doi.org/10.1145/3428275> (cited on pp. 2 and 4)
- Jeff Bezanson. 2015. *Abstraction in technical computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. <http://hdl.handle.net/1721.1/99811> (cited on pp. 14, 23, 75, 77, and 91)
- Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 120 (Oct 2018). <https://doi.org/10.1145/3276490> (cited on p. 5)

- 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> (cited on pp. 1 and 5)
- Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-Oriented Programming. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1986)*. <https://doi.org/10.1145/28697.28700> (cited on p. 5)
- Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *European Conference on Object-Oriented Programming (ECOOP 2008)*. https://doi.org/10.1007/978-3-540-70592-5_2 (cited on p. 78)
- Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*. https://doi.org/10.1007/3-540-54415-1_73 (cited on p. 75)
- Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec 1985). <https://doi.org/10.1145/6041.6042> (cited on p. 77)
- 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 2015)*. <https://doi.org/10.1145/2676726.2676991> (cited on p. 78)
- Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *International Conference on Functional Programming (ICFP 2011)*. <https://doi.org/10.1145/2034773.2034788> (cited on p. 78)
- Craig Chambers. 1992. Object-oriented multi-methods in Cecil. In *European Conference on Programming Languages (ECOOP 1992)*. <https://doi.org/10.1007/BFb0053029> (cited on p. 5)
- Benjamin Chung. 2023. *A Type System for Julia*. Ph.D. Dissertation. Northeastern University. (cited on pp. 2, 15, and 23)
- Benjamin Chung, Francesco Zappa Nardelli, and Jan Vitek. 2019. Julia’s Efficient Algorithm for Subtyping Unions and Covariant Tuples (Pearl). In *European Conference on Programming Languages (ECOOP 2019, Vol. 134)*. <https://doi.org/10.4230/LIPIcs.EC00P.2019.24> (cited on pp. 30 and 77)

- Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2016. Semantic Subtyping for Objects and Classes. *Comput. J.* 60, 5 (Dec 2016). <https://doi.org/10.1093/comjnl/bxw080> (cited on p. 78)
- N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. [https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0) (cited on p. 33)
- Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic subtyping. *Symposium on Logic in Computer Science* (2002). <https://doi.org/10.1109/LICS.2002.1029823> (cited on pp. 77 and 78)
- Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sep 2008). <https://doi.org/10.1145/1391289.1391293> (cited on pp. 11 and 77)
- Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-Bounded Polymorphism into Shape. In *Conference on Programming Language Design and Implementation (PLDI 2014)*. <https://doi.org/10.1145/2594291.2594308> (cited on pp. 58 and 76)
- Radu Grigore. 2017. Java Generics Are Turing Complete. In *Symposium on Principles of Programming Languages (POPL 2017)*. <https://doi.org/10.1145/3009837.3009871> (cited on pp. 23, 75, and 78)
- Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2000. Regular Expression Types for XML. In *International Conference on Functional Programming (ICFP 2000)*. <https://doi.org/10.1145/351240.351242> (cited on p. 77)
- Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of $D<$: And Its Decidable Fragments. *Proc. ACM Program. Lang.* 4, POPL, Article 9 (Dec 2019). <https://doi.org/10.1145/3371077> (cited on pp. 2, 23, 75, 76, and 77)
- Atsushi Igarashi and Mirko Viroli. 2002. On Variance-Based Subtyping for Parametric Types. In *European Conference on Object-Oriented Programming (ECOOP 2002)*. https://doi.org/10.1007/3-540-47993-7_19 (cited on pp. 27 and 78)
- Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD 2007)*.

- <https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/> (cited on pp. 58 and 76)
- Kresten Krab Thorup and Mads Torgersen. 1999. Unifying Genericity. In *European Conference on Object-Oriented Programming (ECOOP 1999)*. https://doi.org/10.1007/3-540-48743-3_9 (cited on pp. 27 and 78)
- Gary T. Leavens and Todd D. Millstein. 1998. Multiple Dispatch as Dispatch on Tuples. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1998)*. <https://doi.org/10.1145/286936.286977> (cited on p. 9)
- Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2019. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.* 4, POPL 2020, Article 66 (Dec 2019). <https://doi.org/10.1145/3371134> (cited on pp. 76 and 77)
- Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems (APLAS 2020)*. https://doi.org/10.1007/978-3-030-64437-6_7 (cited on p. 77)
- Per Martin-Löf. 1994. *Analytic and Synthetic Judgements in Type Theory*. Springer Netherlands, Dordrecht, 87–99. https://doi.org/10.1007/978-94-011-0834-8_5 (cited on p. 30)
- Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 150 (Oct 2021). <https://doi.org/10.1145/3485527> (cited on pp. 1, 2, 4, and 8)
- Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Symposium on Principles of Programming Languages (POPL 1992)*. <https://doi.org/10.1145/143165.143228> (cited on pp. 23 and 75)
- Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-theoretic Types for Erlang. <https://doi.org/10.48550/arXiv.2302.12783> arXiv:2302.12783 [cs.PL] (cited on p. 78)
- Daniel Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It?. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2008)*. <https://doi.org/10.1145/1449764.1449804> (cited on p. 78)
- Ross Tate. 2013. Mixed-Site Variance. In *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented*

- Languages* (Indianapolis, IN, USA). <http://www.cs.cornell.edu/~ross/publications/mixedsite/> (cited on p. 59)
- Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java’s Type System. In *Conference on Programming Language Design and Implementation (PLDI 2011)*. <https://doi.org/10.1145/1993498.1993570> (cited on pp. 23, 58, and 78)
- Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. 2005. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL 2005)*. https://homepages.inf.ed.ac.uk/wadler/fool/program/final/14/14_Paper.pdf (cited on p. 78)
- Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. 2004. Adding Wildcards to the Java Programming Language. In *Symposium on Applied Computing (SAC 2004)*. <https://doi.org/10.1145/967900.968162> (cited on pp. 3, 11, 27, and 78)
- Stefan Wehr and Peter Thiemann. 2009. On the Decidability of Subtyping with Bounded Existential Types. In *Programming Languages and Systems*. https://doi.org/10.1007/978-3-642-10672-9_10 (cited on p. 78)
- Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (Oct 2018). <https://doi.org/10.1145/3276483> (cited on pp. vii, 3, 14, 15, 16, 21, 58, and 60)

A

APPENDIX

A.1 PROPERTIES OF SUBTYPING

A.1.1 Decidable Subtyping

Lemma 13 (Context weakening in \mathcal{M} (Lemma 5)). *The measure of a type signature does not change if the environment is extended (in any position) with a variable that occurs neither in the signature nor in the environment, i.e., $\forall \psi, \Gamma, \Gamma'. \forall l < V <: u$ with $\neg \text{occ}(V; \psi)$ and $\neg \text{occ}(V; \Gamma)$ and $\neg \text{occ}(V; \Gamma')$.*

$$\mathcal{M}(\Gamma ++ \Gamma'; \psi) = \mathcal{M}(\Gamma, l <: V <: u ++ \Gamma'; \psi),$$

where $\Gamma ++ \Gamma''$ denotes the concatenation of lists, and occ is defined in Figure 18.

Proof. By strong induction on $n = |\Gamma| + |\Gamma'| + |\psi|$.

Case $n = 0$ is not possible: the minimal size of a type signature is 1.

Base cases for \top and \perp are straightforward; in the base case for V' , environment $\Gamma ++ \Gamma'$ is empty and $V \neq V'$, meaning that

$$\mathcal{M}(\cdot; V') = 1 = \mathcal{M}(l <: V <: u; V').$$

In the inductive step for n , the induction hypothesis (IH) states that $\forall n' < n. \forall \psi', \Gamma'', \Gamma'''$ with $n' = |\Gamma''| + |\Gamma'''| + |\psi'|$. $\forall l <: V <: u$ with $\neg \text{occ}(V; \psi')$ and $\neg \text{occ}(V; \Gamma'')$ and $\neg \text{occ}(V; \Gamma''')$.

$$\mathcal{M}(\Gamma'' ++ \Gamma'''; \psi') = \mathcal{M}(\Gamma'', l <: V <: u ++ \Gamma'''; \psi').$$

Case analysis on ψ . Base cases \top and \perp are straightforward. Cases \times , $N\{\dots\}$, and \cup are also straightforward using the induction hypothesis for components of ψ . The remaining cases are:

- Case V' . Case analysis on Γ' .
 - Case \cdot . Because $\neg \text{occ}(V; V')$, we know $V \neq V'$. Thus, $\mathcal{M}(\Gamma, l' <: V <: u'; V') = \mathcal{M}(\Gamma; V')$ by definition of \mathcal{M} .

– Case $\Gamma', l' <: V' <: u'$. By definition,

$$\mathcal{M}(\Gamma \multimap \Gamma', l' <: V' <: u'; V') = 1 + \mathcal{M}(\Gamma \multimap \Gamma'; l') + \mathcal{M}(\Gamma \multimap \Gamma'; u').$$

Since $|\Gamma| + |\Gamma'| + |l'| < |\Gamma| + |\Gamma', l' <: V' <: u'| + |V'| = |\Gamma| + |\Gamma'| + |l'| + |u'| + 1$, the IH applies with $\Gamma'' = \Gamma, \Gamma''' = \Gamma', \psi' = l'$, which gives $\mathcal{M}(\Gamma \multimap \Gamma'; l') = \mathcal{M}(\Gamma, l <: V <: u \multimap \Gamma'; l')$, and similarly for u' . Thus,

$$\mathcal{M}(\Gamma \multimap \Gamma', l' <: V' <: u'; V') = \mathcal{M}(\Gamma, l <: V <: u \multimap \Gamma', l' <: V' <: u'; V').$$

• Case $\exists l' <: V' <: u'. \psi$. By definition,

$$\begin{aligned} & \mathcal{M}(\Gamma \multimap \Gamma'; \exists l' <: V' <: u'. \psi) \\ &= \\ & 1 + \mathcal{M}(\Gamma \multimap \Gamma'; l') + \mathcal{M}(\dots u') + \mathcal{M}(\Gamma \multimap \Gamma', l' <: V' <: u'; \psi). \end{aligned}$$

Similarly to the last subcase of the V' case, the IH applies to l' and u' . Furthermore, since $|\Gamma| + |\Gamma'| + |l'| + |u'| + |\psi| < |\Gamma| + |\Gamma'| + 1 + |l'| + |u'| + |\psi|$, the IH applies to ψ with $\Gamma'' = \Gamma, \Gamma''' = (\Gamma', l' <: V' <: u')$, $\psi' = \psi$. All pieces combined,

$$\mathcal{M}(\Gamma \multimap \Gamma'; \exists l' <: V' <: u'. \psi) = \mathcal{M}(\Gamma, l <: V <: u \multimap \Gamma'; \exists l' <: V' <: u'. \psi).$$

□

A.1.2 Unification-Free Subtyping

Lemma 14 (Subtyping of \perp implies arbitrary subtyping (Lemma 6)).

$$\forall \tau, \delta_{\perp}, \Gamma. \quad \Gamma \vdash \tau <: \delta_{\perp}[\perp] \implies (\forall \tau', \delta'. \quad \Gamma \vdash \delta'[\tau] <: \tau').$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta_{\perp}[\perp]$.

• Case ST-BOT $\Gamma \vdash \delta[\perp] <: \delta_{\perp}[\perp]$ where $\tau = \delta[\perp]$.

The case concludes by ST-BOT: $\Gamma \vdash \delta'[\delta[\perp]] <: \tau'$.

• Case ST-VARLEFT $\Gamma \vdash \delta[V] <: \delta_{\perp}[\perp]$.

By the form of the rule, $\Gamma \vdash \delta[u] <: \delta_{\perp}[\perp]$. By IH, $\Gamma \vdash \delta'[\delta[u]] <: \tau'$. Thus, the case concludes by ST-VARLEFT: $\Gamma \vdash \delta'[\delta[V]] <: \tau'$.

• Case ST-TUPLE, subcase where $\delta_{\perp} = \delta_{\perp}' \times \tau_2'$ ($\delta_{\perp} = \square$ is not possible, and $\delta_{\perp} = \tau_1 \times \delta_{\perp}'$ is proved analogously), $\tau = \tau_1 \times \tau_2$: $\Gamma \vdash \tau_1 \times \tau_2 <: \delta_{\perp}'[\perp] \times \tau_2'$.

By the form of the rule, $\Gamma \vdash \tau_1 <: \delta'_\perp[\perp]$. By IH, $\Gamma \vdash \delta^h[\tau_1] <: \tau'$ for all δ^h , so we can take it to be $\delta'[\Box \times \tau_2]$. Thus, the case concludes by IH: $\Gamma \vdash \delta'[\tau_1 \times \tau_2] <: \tau'$.

- Case ST-UNIONLEFT $\Gamma \vdash \delta[\tau_1 \cup \tau_2] <: \delta_\perp[\perp]$ where $\tau = \tau_1 \cup \tau_2$. By the form of the rule, $\Gamma \vdash \delta[\tau_1] <: \delta_\perp[\perp]$ and $\Gamma \vdash \delta[\tau_2] <: \delta_\perp[\perp]$. By IH, $\Gamma \vdash \delta'[\delta[\tau_1]] <: \tau'$ and $\Gamma \vdash \delta'[\delta[\tau_2]] <: \tau'$. Thus, the case concludes by ST-UNIONLEFT: $\Gamma \vdash \delta'[\delta[\tau_1 \cup \tau_2]] <: \tau'$.

The remaining cases (ST-TOP, ST-VARREFL, ST-VARRIGHT, ST-INV, ST-UNIONRIGHT) are not possible. \square

Lemma 15 (Subtyping of inner union on the right (Lemma 7)). $\forall \tau, \delta', \tau'_1, \tau'_2, \Gamma$, with $\vdash \Gamma$ and $\Gamma \vdash \tau, \delta', \tau'_1, \tau'_2$.

$$\begin{aligned} \Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2] \\ \implies \\ (\forall \delta_1, \delta_2, \text{ with } \Gamma \vdash \delta_1, \delta_2 \text{ and } \Gamma \vdash \delta_1 <: \delta_2. \quad \Gamma \vdash \delta_1[\tau] <: \delta_2[\delta'[\tau'_1]] \cup \delta_2[\delta'[\tau'_2]]). \end{aligned}$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta'[\tau'_1 \cup \tau'_2]$.

- Case ST-BOT by ST-BOT.
- Case ST-VARLEFT by the form of the rule, IH, and ST-VARLEFT.
- Case ST-TUPLE, subcase where $\delta' = \delta'' \times \tau'$: $\Gamma \vdash \tau_1 \times \tau_2 <: \delta''[\tau'_1 \cup \tau'_2] \times \tau'$. By the form of the rule, $\Gamma \vdash \tau_1 <: \delta''[\tau'_1 \cup \tau'_2]$ and $\Gamma \vdash \tau_2 <: \tau'$.
By IH applied to $\Gamma \vdash \tau_1 <: \delta''[\tau'_1 \cup \tau'_2]$, $\Gamma \vdash \delta_1^h[\tau_1] <: \delta_2^h[\delta''[\tau'_1]] \cup \delta_2^h[\delta''[\tau'_2]]$ for all δ_1^h, δ_2^h s.t. $\Gamma \vdash \delta_1^h <: \delta_2^h$. Thus, we can take them to be $\delta_2[\Box \times \tau_2]$ and $\delta_2[\Box \times \tau']$, respectively, which concludes the case with $\Gamma \vdash \delta_1[\tau_1 \times \tau_2] <: \delta_2[\delta''[\tau'_1] \times \tau_2] \cup \delta_2[\delta''[\tau'_2] \times \tau']$.
- Case ST-UNIONLEFT by the form of the rule, IH, and ST-UNIONLEFT.
- Case ST-UNIONRIGHT, subcase $i = 1$ where $\delta' = \Box$: $\Gamma \vdash \tau <: \tau'_1 \cup \tau'_2$. By the form of the rule, $\Gamma \vdash \tau <: \tau'_1$. By assumption, $\Gamma \vdash \delta_1 <: \delta_2$, and thus, $\Gamma \vdash \delta_1[\tau] <: \delta_2[\tau'_1]$. The case concludes by ST-UNIONRIGHT with $i = 1$: $\Gamma \vdash \delta_1[\tau] <: \delta_2[\tau'_1] \cup \delta_2[\tau'_2]$.

The remaining cases (ST-TOP, ST-VARREFL, ST-VARRIGHT, ST-INV) are not possible. \square

Lemma 16 (Adding inner union on the right (Lemma 8)). $\forall \tau, \delta', \tau', \Gamma$, with $\vdash \Gamma$ and $\Gamma \vdash \tau, \delta', \tau'$.

$$\Gamma \vdash \tau <: \delta'[\tau'] \implies (\forall \tau''. \Gamma \vdash \tau <: \delta'[\tau' \cup \tau'']).$$

Proof. By induction on the derivation of $\Gamma \vdash \tau <: \delta'[\tau']$.

- Case ST-TOP where $\delta' = \square$. By assumption $\Gamma \vdash \tau <: \top$ and ST-UNIONRIGHT with $i = 1$, $\Gamma \vdash \tau <: \top \cup \tau''$.
- Case ST-BOT by ST-BOT.
- Case ST-VARREFL where $\delta' = \square$ by assumption and ST-UNIONRIGHT with $i = 1$.
- Case ST-VARLEFT by the form of the rule, IH, and ST-VARLEFT.
- Case ST-VARRIGHT where $\delta' = \square$ by assumption and ST-UNIONRIGHT with $i = 1$.
- Case ST-TUPLE. Subcase δ' by assumption and ST-UNIONRIGHT with $i = 1$. The other two subcases by the form of the rule, IH, and ST-TUPLE.
- Case ST-INV where $\delta' = \square$ by assumption and ST-UNIONRIGHT with $i = 1$.
- Case ST-UNIONLEFT by the form of the rule, IH, and ST-UNIONLEFT.
- Case ST-UNIONRIGHT where $\delta' = \square$ by assumption and ST-UNIONRIGHT with $i = 1$. \square

A.2 SEMANTIC MODEL

Julia's subtype relation was inspired by semantic subtyping. The description of the original Julia design [Bezanson 2015] suggested an intuitive interpretation of types as sets of values, but the interpretation is not well defined. Furthermore, the treatment of invariant parametric types in the interpretation does not match the subtype relation. In particular, for concrete nominal `Name`, the interpretation

$$\llbracket \text{Name}\{\tau\} \rrbracket = \{x \mid \text{typeof}(x) = \text{Name}\{\tau\}\}$$

does not account for the fact that there are multiple syntactic representations τ' corresponding to the same interpretation as τ . Thus, for example, types `Vector{Union{Int,Any}}` and `Vector{Any}` are not equivalent according to the interpretation, but they are equivalent in Julia.

A.2.1 Tuples and Unions

In [Belyakova 2019], I proposed and mechanized in Coq a semantic interpretation of (a subset of) Julia types τ as **sets of concrete value types** σ (or type tags) rather than values, with semantic subtyping defined as $\llbracket \tau \rrbracket \subseteq \llbracket \tau' \rrbracket$. For the type language of non-parametric nominal types, tuples, and unions, a decidable syntactic subtype relation based on disjunctive normal form coincides with the set inclusion of interpretations. Figure 31 presents the interpretation and corresponding syntactic subtyping in the style of Chapter 4 for the type language of base types N , covariant tuples, and unions. Σ denotes the set of all value types σ .

In [Belyakova 2019], I show that the decidable syntactic subtype relation is sound and complete with respect to the interpretation.

Theorem 8. $\forall \tau, \tau'.$

$$\begin{aligned} & \tau <: \tau' \\ & \iff \\ & (\forall \sigma. \sigma \in \llbracket \tau \rrbracket \implies \sigma \in \llbracket \tau' \rrbracket). \end{aligned}$$

A.2.2 Invariant Constructors

The interpretation in Figure 31 can be extended to support invariant type constructors such as `Vector{...}`. To account for the issue of multiple syntactic representations of types, in the new system, types are given an indexed interpretation $\llbracket \cdot \rrbracket_n$. This approach was also mechanized in Coq.

Grammar

$$\begin{aligned}
\tau &::= \top \mid \perp \mid N \mid \tau_1 \times \tau_2 \mid \tau_1 \cup \tau_2 && \text{Type} \\
\sigma &::= N \mid \sigma_1 \times \sigma_2 && \text{Concrete value type} \\
\delta &::= \square \mid \delta \times \tau \mid \tau \times \delta && \text{Distributivity context}
\end{aligned}$$

Interpretation $\llbracket \cdot \rrbracket$

$$\begin{aligned}
\llbracket \top \rrbracket &= \Sigma \\
\llbracket \perp \rrbracket &= \emptyset \\
\llbracket N \rrbracket &= \{N\} \\
\llbracket \tau_1 \times \tau_2 \rrbracket &= \{\sigma_1 \times \sigma_2 \mid \sigma_1 \in \llbracket \tau_1 \rrbracket, \sigma_2 \in \llbracket \tau_2 \rrbracket\} \\
\llbracket \tau_1 \cup \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket
\end{aligned}$$

Subtyping $\tau <: \tau'$

$$\begin{array}{c}
\overline{\tau <: \top} \qquad \overline{\delta[\perp] <: \tau'} \qquad \overline{N <: N} \\
\\
\frac{\tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2}{\tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2} \qquad \frac{\delta[\tau_1] <: \tau' \quad \delta[\tau_2] <: \tau'}{\delta[\tau_1 \cup \tau_2] <: \tau'} \qquad \frac{\exists i. \tau <: \tau'_i}{\tau <: \tau'_1 \cup \tau'_2}
\end{array}$$

Figure 31: Semantic interpretation and decidable subtyping for simple types

Figure 32 defines the extension. Intuitively, the 0-interpretation is not allowed to look inside invariant type constructor and contains all value types with the same constructor name. The higher the interpretation, the more spurious elements are filtered out; since the equality of the arguments is checked in the lower interpretation, the definition is well formed. Then, semantic subtyping is defined as:

$$\forall w. \llbracket \tau \rrbracket_w \subseteq \llbracket \tau' \rrbracket_w.$$

Despite the more complex indexed interpretation, the syntactic sub-type relation is still sound and complete.

Theorem 9. $\forall \tau, \tau'.$

$$\begin{aligned}
&\tau <: \tau' \\
&\iff \\
&(\forall w. \forall \sigma. \sigma \in \llbracket \tau \rrbracket_w \implies \sigma \in \llbracket \tau' \rrbracket_w).
\end{aligned}$$

A.2.3 Existential Types

To match the intuition described in Chapter 3, an existential type and type variable can be interpreted as:

$$\begin{aligned}
\llbracket \exists l <: X <: u. \tau \rrbracket_w^\eta &= \bigcup_{\llbracket l \rrbracket_w^\eta \subseteq s \subseteq \llbracket u \rrbracket_w^\eta} \llbracket \tau \rrbracket_w^{\eta[X \mapsto s]} \\
\llbracket X \rrbracket_w^\eta &= \eta(X),
\end{aligned}$$

where η maps variables to interpretation sets.

Grammar

$$\tau ::= \dots \mid N\{\tau_1, \dots, \tau_n\} \quad \text{Type}$$

$$\sigma ::= \dots \mid N\{\tau_1, \dots, \tau_n\} \quad \text{Concrete value type}$$

Interpretation $\llbracket \cdot \rrbracket_w$

$$\begin{aligned} \llbracket \top \rrbracket_w &= \Sigma \\ \llbracket \perp \rrbracket_w &= \emptyset \\ \llbracket \tau_1 \times \tau_2 \rrbracket_w &= \{\sigma_1 \times \sigma_2 \mid \sigma_1 \in \llbracket \tau_1 \rrbracket_w, \sigma_2 \in \llbracket \tau_2 \rrbracket_w\} \\ \llbracket \tau_1 \cup \tau_2 \rrbracket_w &= \llbracket \tau_1 \rrbracket_w \cup \llbracket \tau_2 \rrbracket_w \\ \llbracket N\{\tau_1, \dots, \tau_n\} \rrbracket_0 &= \{N\{\tau'_1, \dots, \tau'_n\}\} \\ \llbracket N\{\tau_1, \dots, \tau_n\} \rrbracket_{w+1} &= \{N\{\tau'_1, \dots, \tau'_n\} \mid \forall i. \llbracket \tau'_i \rrbracket_w = \llbracket \tau_i \rrbracket_w\} \end{aligned}$$

Subtyping $\tau <: \tau'$

$$\dots \quad \frac{\tau_i <: \tau'_i \quad \tau'_i <: \tau_i}{N\{\tau_1, \dots, \tau_n\} <: N\{\tau'_1, \dots, \tau'_n\}}$$

Figure 32: Semantic interpretation and decidable subtyping for invariant constructors

However, because of invariant constructors, an interpretation cannot be simply a set of value type tags¹. Consider the following example:

$$\begin{aligned} &\llbracket \exists \text{Ref}\{\text{Int}\} <: X <: \text{Ref}\{\text{Int}\}. \text{Vector}\{X\} \rrbracket_1 \\ &= \\ &\bigcup_{s \in \llbracket \text{Ref}\{\text{Int}\} \rrbracket_1} \llbracket \text{Vector}\{X\} \rrbracket_1^{[X \mapsto s]} \\ &= \\ &\{ \text{Vector}\{\tau\} \mid \llbracket \tau \rrbracket_0 = \llbracket \text{Ref}\{\text{Int}\} \rrbracket_1 \} \\ &= \\ &\{ \text{Vector}\{\tau\} \mid \llbracket \tau \rrbracket_0 = \{ \text{Ref}\{\tau'\} \mid \llbracket \tau' \rrbracket_0 = \{\text{Int}\} \} \} \\ &= \\ &\{ \text{Vector}\{\tau\} \mid \llbracket \tau \rrbracket_0 = \{ \text{Ref}\{\text{Int}\}, \text{Ref}\{\text{Int} \cup \perp\}, \dots \} \} \end{aligned}$$

Note that there are no closed types τ such that

$$\llbracket \tau \rrbracket_0 = \{ \text{Ref}\{\text{Int}\}, \text{Ref}\{\text{Int} \cup \perp\}, \text{Ref}\{\text{Int} \cup \text{Int}\}, \dots \},$$

because the 0-interpretation of any Ref-type includes arbitrary $\text{Ref}\{\tau'\}$, not just Int-interpreted τ' . Therefore, the existential type

$$\exists \text{Ref}\{\text{Int}\} <: X <: \text{Ref}\{\text{Int}\}. \text{Vector}\{X\}$$

is not a supertype of

$$\text{Vector}\{\text{Ref}\{\text{Int}\}\},$$

which does not match Julia subtyping. The crux of the problem is that X can be interpreted in a smaller w than it was instantiated with.

¹ Tags are assumed to be closed, i.e. do not contain free variables.

Grammar

$$\begin{aligned} \tau, l, u &::= \dots \mid X \mid \exists l <: X <: u. \tau && \text{Type} \\ \sigma &::= \dots \text{ where } FV(\sigma) = \emptyset && \text{Concrete value type} \end{aligned}$$

Domain

$$\hat{\Sigma} = \{\sigma^k \mid \sigma \in \Sigma, k \in \mathbb{N}\}$$

Interpretation $\llbracket \cdot \rrbracket_w^\eta$

$$\begin{aligned} \llbracket \top \rrbracket_w^\eta &= \{\sigma^k \mid k \leq w\} \\ \llbracket \perp \rrbracket_w^\eta &= \emptyset \\ \llbracket \tau_1 \times \tau_2 \rrbracket_w^\eta &= \{(\sigma_1 \times \sigma_2)^k \mid \sigma_1^k \in \llbracket \tau_1 \rrbracket_w^\eta, \sigma_2^k \in \llbracket \tau_2 \rrbracket_w^\eta\} \\ \llbracket \tau_1 \cup \tau_2 \rrbracket_w^\eta &= \llbracket \tau_1 \rrbracket_w^\eta \cup \llbracket \tau_2 \rrbracket_w^\eta \\ \llbracket N\{\tau_1, \dots, \tau_n\} \rrbracket_0^\eta &= \{N\{\tau_1', \dots, \tau_n'\}^0\} \\ \llbracket N\{\tau_1, \dots, \tau_n\} \rrbracket_{w+1}^\eta &= \{N\{\tau_1', \dots, \tau_n'\}^{w+1} \mid \forall i. \llbracket \tau_i' \rrbracket_w^\eta = \llbracket \tau_i \rrbracket_w^\eta\} \cup \llbracket N\{\tau_1, \dots, \tau_n\} \rrbracket_w^\eta \\ \llbracket \exists l <: X <: u. \tau \rrbracket_w^\eta &= \bigcup_{0 \leq w' \leq w} \bigcup_{\llbracket l \rrbracket_{w'}^\eta \subseteq \llbracket u \rrbracket_{w'}^\eta} \llbracket \tau \rrbracket_{w'}^{\eta[X \mapsto s]} \\ \llbracket X \rrbracket_w^\eta &= \{\sigma^k \mid \sigma^k \in \eta(X) \text{ and } k \leq w\} \end{aligned}$$

Figure 33: Semantic interpretation for existential types

In the spirit of [Ahmed et al. 2003], the domain of interpretation can be redefined as a set of indexed tags instead of just tags, as presented in Figure 33. In this case, whenever a variable is interpreted in w , all tags originally produced by the potentially higher interpretation are removed. It is easy to see by induction on τ that

$$w' \leq w \implies \llbracket \tau \rrbracket_{w'}^\eta \subseteq \llbracket \tau \rrbracket_w^\eta.$$

The semantic interpretation can be used to examine soundness of decidable syntactic subtyping presented in Chapter 4.1, as well as equivalent rewritings discussed in Chapter 5.2.

Satisfy the restriction

```

1 Pair{T, T} where T
2 Tuple{T, Tuple{T, Int}} where T
3 Tuple{Ref{T}} where T
4 Tuple{T, Ref{T}} where T
5 Tuple{Ref{Tuple{T}}}} where T
6 Vector{Union{T, Int}} where T
7 Ref{Ref{T} where T} where T
8 Ref{Pair{T, S} where S} where T
9 Pair{S, Pair{Int, T} where T<:S} where S
10 Tuple{S, Pair{T, S} where T<:S} where S
11 Tuple{Ref{T} where T>:S} where S
12 Ref{T} where T<:(Ref{S} where S)
13 Tuple{T} where T<:Ref{Ref{<:Any}}
```

Do not satisfy the restriction

```

1 Ref{Pair{T, T} where T}
2 Ref{Tuple{Pair{T, T} where T}}
3 Tuple{T} where T>:(Pair{S,S} where S)
4 Vector{Vector{Union{T, Int}} where T}
5 Vector{Ref{Tuple{T}} where T}
```

Figure 34: Test cases for the analysis of type annotations

```

Error: Couldn't process expression
e =
  :($ (Expr(:$, :d))->begin
    #= none:54 =#
    Base.axes($ (Expr(:$, :arraysym)), $(Expr(:$, :d)))
  end)
err =
  ArgumentError: Not a function definition: :($ (Expr(:$, :d))->begin
    #= none:54 =#
    Base.axes($ (Expr(:$, :arraysym)), $(Expr(:$, :d)))
  end)
```

Figure 35: An example of a parsing error

A.3 EVALUATION


```

Error: Couldn't process type annotation
  tastr = "Tuple{Union{Document, Node}} where \$(esc.(P)...)"
  err = AssertionError: Unsupported lb-var-ub format

Error: Unsupported Expr type annotation
  ty = :(typeof((year, month, day, yearmonth, ...))...)

Error: Couldn't process type annotation
  tastr = "(Tuple{A} where Base.IteratorSize(A)::Base.SizeUnknown) where A"
  err = AssertionError: Unsupported lb-var-ub format

Error: Couldn't process type annotation
  tastr = "(((Tuple{(\$T_nameparam){\$N, \$M, \$FT}} ... \$(T_params...))"
  err = AssertionError: Unsupported lb-var-ub format

Error: Couldn't process type annotation
  tastr = "Tuple{Union{map((T->beginn  #= none:302 =#n ...))...}}"
  err = Base.Meta.ParseError("missing comma or ) in argument list")

```

Figure 36: Type annotation processing errors