Decidable Subtyping of Existential Types for Julia

JULIA BELYAKOVA, Purdue University, USA
BENJAMIN CHUNG, JuliaHub, USA
ROSS TATE, Independent Consultant, USA
JAN VITEK, Northeastern University, USA and Charles University, Czechia

Julia is a modern scientific-computing language that relies on multiple dispatch to implement generic libraries. While the language does not have a static type system, method declarations are decorated with expressive type annotations to determine when they are applicable. To find applicable methods, the implementation uses subtyping at run-time. We show that Julia's subtyping is undecidable, and we propose a restriction on types to recover decidability by stratifying types into method signatures over value types—where the former can freely use bounded existential types but the latter are restricted to use-site variance. A corpus analysis suggests that nearly all Julia programs written in practice already conform to this restriction.

CCS Concepts: • **Theory of computation** → *Type structures*; • **Software and its engineering** → **Data types and structures**; *Semantics*; *Just-in-time compilers*; **Language features**.

Additional Key Words and Phrases: Decidability, Subtyping, Julia

ACM Reference Format:
Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2024. Decidable Subtyping of Existential Types for Julia. *Proc. ACM Program. Lang.* 8, PLDI, Article 191 (June 2024), 24 pages. https://doi.org/10.1145/3656421

1 INTRODUCTION

Julia is a scientific-computing language carefully designed so that an implementation can generate efficient code for performance-critical abstractions. The central abstraction mechanism offered by the language is multiple dispatch with an expressive type-annotation language and a complex subtype relation. Multiple dispatch is a mechanism dating back to Lisp [Bobrow et al. 1986], which allows *generic functions* to have multiple implementations, called *methods*, tailored to different argument types. The following code snippet illustrates the expressive power of Julia's type-annotation language, in this case defining the binary "-" operator for various types:

```
- (x::BigInt, y::BigInt) = ...
- (x::T, y::T) where T <: Union{Int16, Int32} = ...
- (m::Missing, n::Number) = ...
- (A::AbstractArray{T,N}) where {T,N} = ...
```

Julia has both nominal type constructors, such as `Number` and `AbstractArray{T,N}`, and a variety of structural types. `Any` is a supertype of all types. Finite unions of types are written `Union{Int16, Int32}`. Tuple types such as `Tuple{String, Number}` are covariant in their element types. Finally,

Authors' addresses: Julia Belyakova, Purdue University, West Lafayette, USA, ybelyako@purdue.edu; Benjamin Chung, JuliaHub, Boston, USA, benjamin.chung@juliahub.com; Ross Tate, Independent Consultant, Ithaka, USA, research@rosstate.org; Jan Vitek, Northeastern University, Boston, USA and Charles University, Prague, Czechia, j.vitek@northeastern.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART191
https://doi.org/10.1145/3656421

bounded existential types, called union-all in Julia, are written `t where l<:T<:u`, and represent the union of types `t[t'/T]` for all valid instantiations `l<:t'<:u` of the type variable `T`.

While it has a type-annotation language, Julia does not have a static type system. Thus, subtyping is only used at run-time, in particular, to find an applicable method for each function call. The reference definition of subtyping lies in a highly optimized, and evolving, C implementation. Zappa Nardelli et al. [2018] give a faithful account; the departures from the implementation are believed to be bugs in the C code. Chung et al. [2019] argue that the complexity of the implementation comes in equal parts from the type language's expressive power and from efficiency concerns.

As we will demonstrate, Julia subtyping is undecidable. When a language has an undecidable static type system, compile-time errors can typically be fixed by adding annotations to the program as needed. For Julia, incomplete subtyping can incorrectly change the execution of a program, either during dynamic dispatch, or when adding new method definitions, or when generating code. When subtyping fails to terminate, the implementation raises an exception with little insight as to how to fix the offending source code. The subtyping algorithm is still under development, as these issues suggest: `#41948` (a `StackOverflowError` caused by a function definition), `#33137` (a problem with Julia's "diagonal rule"); `#24166` (a problem with reflexivity and transitivity); `#39099` (a problem with transitivity of variadic tuple arguments).

Our goal is to develop a clear formalization of the intended subtype relation—one that can be understood and adopted by programmers, and one for which an algorithm can be proved sound and complete. The breadth of Julia's type features makes this difficult to do all at once, so in this paper we focus on bounded existential types.

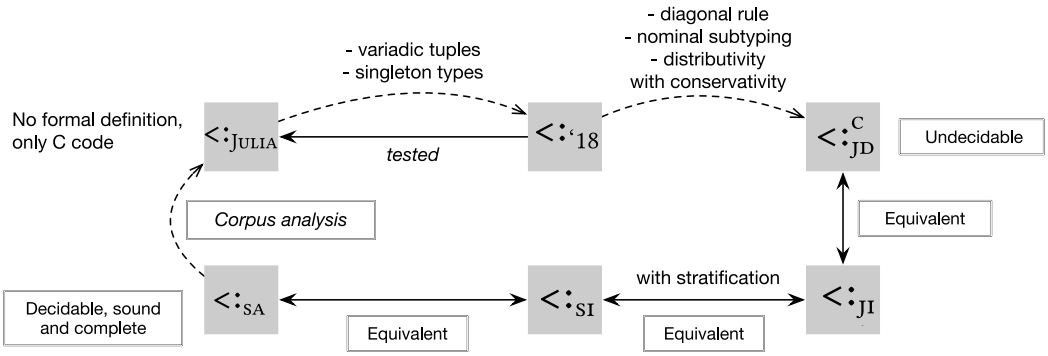

Fig. 1. Overview

Fig. 1 is a roadmap for this paper. We start with the implementation of subtyping, $<:_{\text{JULIA}}$, used in practice. The 2018 paper gave a formal definition of that relation, $<:_{'18}$, with some small differences (e.g. it ignored variadic tuples and singleton types). To validate $<:_{'18}$, the authors performed extensive testing of both definitions. The handful of differences between them were either ascribed to bugs—some of which were fixed by the Julia team—or to undefined behavior. Much of the effort was in establishing a better understanding of Julia's diagonal rule, which restricts how existential types can be instantiated in certain situations that were only informally described. While interesting, the feature is a layer on top of existential types, which we show here are challenging enough on their own. One of our contributions is that, by ignoring the diagonal rule in particular, we demonstrate that one can recover a much simpler and more familiar *declarative* formalization of Julia's subtyping. The language intends its types to approximate sets and its subtyping to approximate set inclusion, a transitive relation. Therefore, the declarative formalization, $<:^{\text{C}}_{\text{JD}}$, simply declares, with an explicit

rule, that subtyping is transitive. While this trivially captures the intended behavior, it also makes reasoning about subtyping difficult.[1] Thus, our second contribution is to establish an *invertible* formalization of subtyping, $<:_{\text{JI}}$, where only one rule can act on a given side at a given time. Invertible subtyping is proved equivalent to the declarative formalization of subtyping when restricted to *conservative* types, meaning types whose quantified type variables' lower bounds are subtypes of their corresponding upper bounds. At this point, it is rather easy to show that subtyping is undecidable. More specifically, we prove the undecidability of invertible subtyping between conservative types. Our proof proceeds by reduction of subtyping of one of Pierce [1992]'s deterministic fragments of System $F_{\leq}$ to invertible subtyping. We do this by translating $F^{P}_{\leq}$ types to conservative types, and by showing $F^{P}_{\leq}$ subtyping holds if and only if their translations are invertible *super*types (where the only-if direction relies critically on invertibility of the rules). The key insight is this flipping of subtyping into supertyping and, likewise, flipping upper-bounded universal quantification in $F^{P}_{\leq}$ to lower-bounded existential quantification in Julia.

To find a decidable yet practical fragment of Julia types, we conduct an empirical study, demonstrating that the types actually written by users are *stratified*. In particular, method type annotations can be stratified as *method signatures* that predicatively quantify over non-quantifying *value types*[2] with use-site variance, which Julia already has a shorthand for (e.g. `Vector{<:Number}`). The key observation is that use-site variance is the only employed application of impredicative existential quantification where an existential type variable is instantiated with an existential type. One nice property of the stratification is that, aside from checking conservativity of bounds, it is syntactic. Our corpus analysis of all the source code of 9,335 Julia packages finds only a handful of stratification violations in 16.5 million lines of code. We conclude with the definition of *algorithmic* subtyping, $<:_{\text{SA}}$, which we prove specifies a sound and complete algorithm for subtyping between conservative stratified types. The key insight is that stratification ensures that only one of the two types being compared contains flexible variables, i.e. existentially quantified variables that need to be instantiated, preventing major complications like recursive constraints. Combined with the above contributions, this provides the foundation for a sound and complete algorithm, upon which future work can expand to encompass the full feature set. Belyakova [2023] in her thesis discusses many of the missing features in the context of a restricted subtype relation that is shown to be decidable but not complete. The step-wise approach presented here is key to be able to prove that important property.

2 BACKGROUND ON JULIA

Julia is a high-level, dynamically typed programming language—originally designed for scientific computing—that addresses the, so-called, "two-language problem" by providing both productivity features and performance [Bezanson et al. 2018, 2017]. For productivity, the language provides garbage collection, dynamic typing, and multiple dispatch—resolved at run-time using subtyping. For performance, it relies on an optimizing compiler that specializes multiple dispatches to direct calls [Pelenitsyn et al. 2021]. Subtyping largely follows the combination of nominal subtyping for

[1] For example, type `Nothing` is declaratively equivalent to `Any where Any<:T<:Nothing`; demonstrated by opening the existential and applying transitivity to `Any <: Nothing` with `T` as the middle type, even though `T` never occurs within the body of the existential. This existential type is odd: its bounds are nonconservative, i.e. its lower bound is not a subtype of its upper bound.

[2] Value types are not to be confused with run-time types or, for those familiar with Julia terminology, concrete types.

user-defined nominal types and semantic subtyping[3] for covariant tuples, unions, and bounded existential types. While the language has no formal definition of its subtyping algorithm, Zappa Nardelli et al. [2018] attempted to reverse-engineer an algorithmic definition and test it empirically. That definition is mostly accurate, with most observed differences due to bugs in Julia.

```
abstract type Real <: Number end
struct Rational{T<:Integer} <: Real
  num :: T
  den :: T
end
mutable struct BitSet <: AbsSet{Int}
```

```
abstract type Ref{T} end
struct RefArray{
    T,
    A<:AbstractArray{T},
    R
  } <: Ref{T}
```

Fig. 2. Datatype declarations

Nominal types are induced by user-defined datatype declarations and constitute a single-parent inheritance hierarchy. Abstract types can be inherited from, and concrete types can be instantiated. Fig. 2 illustrates a definition of several datatypes. Both `Real` and `Ref{t}` are abstract; the remaining are concrete. Parametric types can have non-recursive lower and upper bounds on type variables, and they are invariant with respect to their type parameters. Thus, `Ref{t1}` is a subtype of `Ref{t2}` only if the arguments are equivalent. Tuples are immutable, and their type parameters are covariant. Type `Union{t…}` describes a union (not sum) of types. For instance, `Real` is a subtype of `Union{Number,String}`, and `Union{t1,t2}` is a subtype of `t` if all components are subtypes: `t1<:t` and `t2<:t`. Following the semantic-subtyping mindset, tuples distribute over unions. For example, `Tuple{String,Union{Int32,Int64}}` represents binary tuples where the first component is a string, and the second is either a 32- or 64-bit integer. Due to distributivity, this type is equivalent to `Union{Tuple{String,Int32},Tuple{String,Int64}}`. *Bounded existential types*, called union-all in Julia, have the form `t where l<:T<:u`, where the lower and upper bounds can be omitted and default to the bottom type—`Union{}`—and top type—`Any`—respectively. Existentials can model Java wildcards but are more expressive. Intuitively, they denote a union of `t[t'/T]` for all instantiations of the type variable `T` such that `l<:t'<:u`. Similarly to subtyping of union types, the intent is that

- `(t where l<:T<:u)<:t2` if `t[t'/T]<:t2` for all valid instantiations `t'` of `T`,
- `t1<:(t where l<:T<:u)` if there exists a valid instantiation `t'` with `t1<:t[t'/T]`.

For example, `Vector{Int32}` is a subtype of `Vector{T} where T<:Number` because `T` can be instantiated with `Int32`, and `Vector{T} where T<:Number` is a subtype of `Vector{S} where S` because for all valid instantiations `t'` of `T`, type variable `S` can be instantiated with the type `t'`. Tuples distribute over existential types; types `Tuple{Vector{T} where T}` and `Tuple{Vector{T}} where T` are equivalent.

Existential types are *impredicative*: quantifiers can appear anywhere in a type, and type variables can be instantiated with any type. Type `Vector{Ref{T} where T}` denotes a heterogeneous vector of references, whereas `Vector{Ref{S}} where S` denotes a union of homogeneous vectors of references. Thus, a vector containing integer references, `Vector{Ref{Int32}}`, is a subtype of the latter but not the former as the type arguments `Ref{Int32}` and `Ref{T} where T` are not equivalent; in particular, a `Ref{String}` could be put into a `Vector{Ref{T} where T}` but not into

[3]This is a misnomer, though, because semantic subtyping is subtyping that is complete with respect to a particular semantics [Castagna and Frisch 2005], which Julia fails to be. Rather, Julia includes expressive subtyping rules commonly associated with semantic subtyping, such as distributivity of tuples over unions.

a `Vector{Ref{Int32}}`. Top-level existential types appear in signatures of polymorphic method definitions. Recall that types are only used for multiple dispatch. To call a method, one provides arguments that inhabit the corresponding existential type. In the method body, the existential is implicitly unpacked, with the witness type being some valid instantiation induced by subtyping. Consider a function `f(v::Vector{T}) where T`: its signature is `Tuple{Vector{T}} where T`. For a call such as `f([5,7,5])`, represented with the type `Tuple{Vector{Int32}}`, dispatch resolution relies on tuple subtyping—which succeeds in this case—and instantiates `T` with `Int32`.

3 SPECIFICATION OF SUBTYPING

As there is no official definition of subtyping, we face the problem of choosing a baseline in our exploration of decidability of subtyping. One option is to use Zappa Nardelli et al. [2018], as it matches the implementation, but the complexity of their rules is daunting. Furthermore, being accurate to the implementation is not necessarily what one wants; the implementation has bugs, so it can be unclear whether accepting a program is intentional or accidental. Thus, we formalize declarative subtyping based on the intuitions provided by the Julia documentation. To keep the paper focused, we omit features not relevant to undecidability, such as distributivity of tuples, the diagonal rule, nominal inheritance, variadic tuples, "plain bits" values, and singleton types. Our paper lays the foundations that such extensions can be built upon and establishes a practical means to address the key source of undecidability.

		Julia syntax	
$\sigma ::=$	$\top$	`Any`	top
	$\bot$	`Union{}`	bottom
	X	X	type variable
	$\sigma \times \ldots \times \sigma$	`Tuple{`$\sigma \ldots \sigma$`}`	covariant tuple
	`C`$\langle\sigma, \ldots, \sigma\rangle$	`C{`$\sigma, \ldots, \sigma$`}`	invariant nominal constructor
	$\sigma \cup \sigma$	`Union{`σ, σ`}`	union
	$\exists \sigma_\ell <: \mathrm{X} <: \sigma_u.\ \sigma$	σ **where** $\sigma_\ell <: \mathrm{X} <: \sigma_u$	bounded existential

Fig. 3. Type grammar

For the remainder of the paper, we depart from the Julia type syntax and adopt a more standard notation. The type grammar is given by Fig. 3. The shorthand `C` is for nullary datatypes. Datatype declarations are implicit and do not restrict type parameters. A kind context is denoted Σ and is a, possibly empty, sequence of type variables with explicit bounds $\sigma_\ell <: \mathrm{X} <: \sigma_u$. As a shorthand, we omit $\bot$ lower bounds and $\top$ upper bounds.

Julia has a notion of type validity: it rejects types with unbound type variables. However, this notion is too permissive, as it allows a type variable to have a lower bound that is not a subtype of its upper bound; we call such bounds *nonconservative*. This causes algorithmic problems and is an unnecessary degree of freedom; our corpus analysis reveals that nonconservative bounds are not used in practice. Thus, we consider only types that are well-scoped and conservative. Fig. 4 formalizes type validity, which is parameterized by the subtype relation that enforces conservativity.

3.1 Declarative Subtyping

Our declarative formalization of subtyping is given in Fig. 5. It is parameterized by a validity predicate and includes explicit rules for reflexivity and transitivity. The latter make subtyping reflexive and transitive by definition. Each of the remaining rules is standard for the relevant feature. Observe

$$\frac{}{\Sigma \vdash_S \top} \qquad \frac{}{\Sigma \vdash_S \bot} \qquad \frac{\sigma_\ell <: \mathrm{X} <: \sigma_u \in \Sigma}{\Sigma \vdash_S \mathrm{X}} \qquad \frac{\forall i \in [1,n] \qquad \Sigma \vdash_S \sigma_i}{\Sigma \vdash_S \sigma_1 \times \ldots \times \sigma_n}$$

$$\frac{\forall i \in [1,n] \qquad \Sigma \vdash_S \sigma_i}{\Sigma \vdash_S \mathtt{C}\langle \sigma_1 \ldots \sigma_n \rangle} \qquad \frac{\Sigma \vdash_S \sigma_1 \qquad \Sigma \vdash_S \sigma_2}{\Sigma \vdash_S \sigma_1 \cup \sigma_2}$$

$$\frac{\Sigma \vdash_S \sigma_\ell \qquad \Sigma \vdash_S \sigma_u \qquad \Sigma \vdash \sigma_\ell <:_S \sigma_u \qquad \Sigma, \sigma_\ell <: \mathrm{X} <: \sigma_u \vdash_S \sigma}{\Sigma \vdash_S \exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma}$$

$$\frac{}{\vdash_S \cdot} \qquad \frac{\vdash_S \Sigma \qquad \Sigma \vdash_S \sigma_\ell \qquad \Sigma \vdash_S \sigma_u \qquad \Sigma \vdash \sigma_\ell <:_S \sigma_u}{\vdash_S \Sigma, \sigma_\ell <: \mathrm{X} <: \sigma_u}$$

Fig. 4. Type and kind-context validity: $\boxed{\Sigma \vdash_S \sigma}$ and $\boxed{\vdash_S \Sigma}$ for a given subtype relation $<:_S$

that unlike in $F_{\leq}$ where universal quantifications are only subtypes of universal quantifications, here existential quantifications are supertypes of arbitrary types provided appropriate instantiations exist. The validity predicate specifies the universe of types to consider when applying transitivity or instantiating existential types.

The parameterized judgments have common instantiations. $\Sigma \vdash^{\mathrm{ws}} \sigma$ denotes validity with the total subtype relation (i.e. not enforcing conservativity) and formalizes when a type is *well-scoped*. $\Sigma \vdash^{\mathrm{c}}_{\mathrm{JD}} \sigma$ and $\Sigma \vdash \sigma <:^{\mathrm{c}}_{\mathrm{JD}} \sigma$ denote the mutually-inductively defined conservativity predicate and declarative subtype relation quantifying over conservative types. Unless otherwise indicated, declarative subtyping henceforth ranges over conservative types.

$$\frac{}{\Sigma \vdash \sigma <:^V_{\mathrm{JD}} \sigma} \qquad \frac{\Sigma \vdash^V \sigma' \qquad \Sigma \vdash \sigma <:^V_{\mathrm{JD}} \sigma' \qquad \Sigma \vdash \sigma' <:^V_{\mathrm{JD}} \sigma''}{\Sigma \vdash \sigma <:^V_{\mathrm{JD}} \sigma''}$$

$$\frac{}{\Sigma \vdash \sigma <:^V_{\mathrm{JD}} \top} \qquad \frac{}{\Sigma \vdash \bot <:^V_{\mathrm{JD}} \sigma} \qquad \frac{\sigma_\ell <: \mathrm{X} <: \sigma_u \in \Sigma}{\Sigma \vdash \mathrm{X} <:^V_{\mathrm{JD}} \sigma_u} \qquad \frac{\sigma_\ell <: \mathrm{X} <: \sigma_u \in \Sigma}{\Sigma \vdash \sigma_\ell <:^V_{\mathrm{JD}} \mathrm{X}}$$

$$\frac{\forall i \in [1,n] \qquad \Sigma \vdash \sigma_i <:^V_{\mathrm{JD}} \sigma_i'}{\Sigma \vdash \sigma_1 \times \ldots \times \sigma_n <:^V_{\mathrm{JD}} \sigma_1' \times \ldots \times \sigma_n'} \qquad \frac{\forall i \in [1,n] \qquad \Sigma \vdash \sigma_i <:^V_{\mathrm{JD}} \sigma_i' \qquad \Sigma \vdash \sigma_i' <:^V_{\mathrm{JD}} \sigma_i}{\Sigma \vdash \mathtt{C}\langle \sigma_1 \ldots \sigma_n \rangle <:^V_{\mathrm{JD}} \mathtt{C}\langle \sigma_1' \ldots \sigma_n' \rangle}$$

$$\frac{\Sigma \vdash \sigma_1 <:^V_{\mathrm{JD}} \sigma' \qquad \Sigma \vdash \sigma_2 <:^V_{\mathrm{JD}} \sigma'}{\Sigma \vdash \sigma_1 \cup \sigma_2 <:^V_{\mathrm{JD}} \sigma'} \qquad \frac{}{\Sigma \vdash \sigma_i <:^V_{\mathrm{JD}} \sigma_1 \cup \sigma_2}$$

$$\frac{\Sigma, \sigma_\ell <: \mathrm{X} <: \sigma_u \vdash \sigma <:^V_{\mathrm{JD}} \sigma'}{\Sigma \vdash \exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma <:^V_{\mathrm{JD}} \sigma'} \qquad \frac{\Sigma \vdash^V \sigma_{\mathrm{X}} \qquad \Sigma \vdash \sigma_\ell <:^V_{\mathrm{JD}} \sigma_{\mathrm{X}} \qquad \Sigma \vdash \sigma_{\mathrm{X}} <:^V_{\mathrm{JD}} \sigma_u}{\Sigma \vdash \sigma[\mathrm{X} \mapsto \sigma_{\mathrm{X}}] <:^V_{\mathrm{JD}} \exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma}$$

Fig. 5. Declarative subtyping: $\boxed{\Sigma \vdash \sigma <:^V_{\mathrm{JD}} \sigma}$ for a given type-validity predicate $\vdash^V$

$$\frac{}{\Sigma \vdash \sigma <:_{\text{JI}} \top} \qquad \frac{}{\Sigma \vdash \bot <:_{\text{JI}} \sigma'}$$

$$\frac{}{\Sigma \vdash \mathrm{X} <:_{\text{JI}} \mathrm{X}} \qquad \frac{\sigma_\ell <: \mathrm{X} <: \sigma_u \in \Sigma \qquad \Sigma \vdash \sigma_u <:_{\text{JI}} \sigma'}{\Sigma \vdash \mathrm{X} <:_{\text{JI}} \sigma'} \qquad \frac{\sigma_\ell <: \mathrm{X} <: \sigma_u \in \Sigma \qquad \Sigma \vdash \sigma <:_{\text{JI}} \sigma_\ell}{\Sigma \vdash \sigma <:_{\text{JI}} \mathrm{X}}$$

$$\frac{\forall i \in [1,n] \qquad \Sigma \vdash \sigma_i <:_{\text{JI}} \sigma_i'}{\Sigma \vdash \sigma_1 \times \ldots \times \sigma_n <:_{\text{JI}} \sigma_1' \times \ldots \times \sigma_n'} \qquad \frac{\forall i \in [1,n] \qquad \Sigma \vdash \sigma_i <:_{\text{JI}} \sigma_i' \qquad \Sigma \vdash \sigma_i' <:_{\text{JI}} \sigma_i}{\Sigma \vdash \mathtt{C}\langle \sigma_1, \ldots, \sigma_n \rangle <:_{\text{JI}} \mathtt{C}\langle \sigma_1', \ldots, \sigma_n' \rangle}$$

$$\frac{\Sigma \vdash \sigma_1 <:_{\text{JI}} \sigma' \qquad \Sigma \vdash \sigma_2 <:_{\text{JI}} \sigma'}{\Sigma \vdash \sigma_1 \cup \sigma_2 <:_{\text{JI}} \sigma'} \qquad \frac{\Sigma \vdash \sigma <:_{\text{JI}} \sigma_i'}{\Sigma \vdash \sigma <:_{\text{JI}} \sigma_1' \cup \sigma_2'} \qquad \frac{\Sigma, \sigma_\ell <: \mathrm{X} <: \sigma_u \vdash \sigma <:_{\text{JI}} \sigma'}{\Sigma \vdash \exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma <:_{\text{JI}} \sigma'}$$

$$\frac{\Sigma \vdash_{\text{JI}} \sigma_{\mathrm{X}} \qquad \Sigma \vdash \sigma_\ell <:_{\text{JI}} \sigma_{\mathrm{X}} \qquad \Sigma \vdash \sigma_{\mathrm{X}} <:_{\text{JI}} \sigma_u \qquad \Sigma \vdash \sigma <:_{\text{JI}} \sigma'[\mathrm{X} \mapsto \sigma_{\mathrm{X}}]}{\Sigma \vdash \sigma <:_{\text{JI}} \exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma'}$$

Fig. 6. Invertible subtyping: $\boxed{\Sigma \vdash \sigma <:_{\text{JI}} \sigma}$

3.2 Invertible Subtyping

When one knows $\sigma_1 \times \sigma_2$ is a subtype of $\sigma_1' \times \sigma_2'$, one would often like to deduce that σ_1 is a subtype of σ_1' and that σ_2 is a subtype of σ_2', conceptually inverting the rule for covariant tuples. However, this is not necessarily true. The kind context could contain a bounded variable $\sigma_1 \times \sigma_2 <: \mathrm{X} <: \sigma_1' \times \sigma_2'$, and transitivity could use X as its intermediate type without ever connecting the respective projections. So, without conservativity of bounds, transitivity makes such desirable subtyping inversions impossible. To this end, we restrict the type system to conservative types. With this restriction, our problematic bounded variable $\sigma_1 \times \sigma_2 <: \mathrm{X} <: \sigma_1' \times \sigma_2'$ is only valid if $\sigma_1 \times \sigma_2$ can be determined to be a subtype of $\sigma_1' \times \sigma_2'$ without X. From this proof of conservativity, one then hopes to extract the expected subtypings between the respective projections. Indeed, we prove that, when restricted to conservative types and kind contexts, declarative subtyping is equivalent to the invertible subtype relation $\Sigma \vdash \sigma <:_{\text{JI}} \sigma$, formalized in Fig. 6, which does not contain an explicit transitivity rule.

Theorem 3.1. *For any kind context Σ satisfying $\vdash_{\text{JI}} \Sigma$, and for any pair of types σ and σ' satisfying $\Sigma \vdash_{\text{JI}} \sigma$ and $\Sigma \vdash_{\text{JI}} \sigma'$, the following equivalence holds:*

$$\Sigma \vdash \sigma <:^{\text{c}}_{\text{JD}} \sigma' \iff \Sigma \vdash \sigma <:_{\text{JI}} \sigma'$$

Proof. The leftward implication is trivially proven by induction on $\Sigma \vdash \sigma <:_{\text{JI}} \sigma'$, with applications of reflexivity and transitivity rules of $<:^{\text{c}}_{\text{JD}}$. The bulk of the proof by far is in the rightward implication. First, we observe that, if its bounds are conservative, an existential type $\exists \sigma_\ell <: \mathrm{X} <: \sigma_u . \sigma$ is equivalent to the type $\exists \bot <: \mathrm{X}' <: \sigma_u . \sigma[\mathrm{X} \mapsto \sigma_\ell \cup \mathrm{X}']$. Then, for the simpler type space with only upper bounds, we define transitivity-eliminating reductions and show that repeatedly applying them necessarily terminates. In particular, we translate the system to Girard's cut-elimination reductions for proof nets of second-order classical linear logic (PN2) [Girard 1987], which handles termination of cut elimination in the presence of impredicativity. □

Invertible subtyping is derived from declarative subtyping by baking transitivity into each rule directly. It allows for an easy use of inversion: when given a subtype relation $\Sigma \vdash \sigma <:_{\text{JI}} \sigma'$, by design only one rule can "act on" a given side. For example, only the tuple rule is applicable to the subtyping $\vdash \sigma_1 \times \sigma_2 <:_{\text{JI}} \sigma_1' \times \sigma_2'$, guaranteeing the respective projections are necessarily subtypes, just as one would hope for. For other features formalized using left and right rules, one can often use induction to dig up an application of the desired one-sided rule. For example, one can easily prove that a union is a subtype of another type only if its components are. However, not all inversions are always possible; for example, a subtype of a union is not necessarily a subtype of either component of the union. Similarly, invertible subtyping is not quite syntax-directed; in particular, the right rule for existentials can apply in an infinite number of ways depending on the instantiating types. So, $\vdash$ `Int32` $<:_{\text{JI}} \exists X.X$ can be shown by instantiating X with `Int32`, or with `Int32` ∪ `Int64`, or with many other types. Nonetheless, invertible subtyping is easier to reason about than declarative subtyping. The equivalence is valuable for both proving undecidability and establishing completeness of our algorithm on restricted types.

4 UNDECIDABILITY OF JULIA SUBTYPING

This section provides a proof of undecidability of Julia subtyping. Consider this code:

```
Neg(T) = Ref{>:T}
Kappa(T) = Pair{Y, <:Neg(Y)} where Y>:T
const Theta = Pair{Z, <:Neg(Kappa(Z))} where Z
println(Kappa(Theta) <: Theta) # StackOverflowError
```

This valid Julia program causes a `StackOverflowError` when executed. A method with a parameter of type `Ref{<:Theta}` would similarly fail if called with a value of type `Ref{Kappa(Theta)}`. Furthermore, adding a second method with `Ref{Kappa(Theta)}` as parameter would also overflow, as Julia tries to prioritize method definitions by subtyping. The code fragment is a contravariant translation of Curien and Ghelli [1990]'s gadget for non-termination in $F_{\leq}$. This singular example does not prove undecidability, but our translation does. Our proof proceeds by showing that System $F_{\leq}^{P}$'s subtyping is equivalent under a translation into our subtyping calculus, thereby demonstrating undecidability by reduction.

4.1 System $F_{\leq}^{P}$

We remove arrow types from $F_{\leq}$ to form the reduced $F_{\leq}^{P}$ as described by Pierce [1992]. Pierce shows that subtyping in $F_{\leq}^{P}$ is equivalent to subtyping in $F_{\leq}$ and, therefore, that $F_{\leq}^{P}$ is undeccidable. Fig. 7 gives the grammar of $F_{\leq}^{P}$, where types are restricted depending on whether they occur in a negative or positive position. Kind contexts Γ^{-} are (possibly empty) sequences of upper-bounding type-variable declarations $\alpha \leq \tau^{-}$ that restrict upper bounds to only be negative types.

τ^{+} ::=	*Positive types*	τ^{-} ::=	*Negative types*
Top	top	α	type variable
$\neg\tau^{-}$	negative negation	$\neg\tau^{+}$	positive negation
$\forall\alpha \leq \tau^{-}.\tau^{+}$	positive quantification	$\forall\alpha \leq Top.\tau^{-}$	negative quantification

Fig. 7. $F_{\leq}^{P}$ grammar

Fig. 8 gives the subtyping rules of $F^P_{\leq}$. As in this paper, the rules treat α-equivalent types as implicitly equivalent. As such, the rule $F^P_{\leq}$-All does not require the two types to use the same variable name; if the variable names are different, they are implicitly α-renamed.

$$\frac{}{\Gamma^- \vdash \tau^- \leq \mathit{Top}} \qquad \frac{\alpha{\leq}\tau^- \in \Gamma^- \qquad \Gamma^- \vdash \tau^- \leq \tau^+}{\Gamma^- \vdash \alpha \leq \tau^+}$$

$$F^P_{\leq}\text{-All}\ \frac{\Gamma^-, \alpha{\leq}\tau_2^- \vdash \tau_1^- \leq \tau^+}{\Gamma^- \vdash \forall \alpha{\leq}\mathit{Top}.\tau_1^- \leq \forall \alpha{\leq}\tau_2^-.\tau^+} \qquad F^P_{\leq}\text{-Neg}\ \frac{\Gamma^- \vdash \tau_2^- \leq \tau_1^+}{\Gamma^- \vdash \neg\tau_1^+ \leq \neg\tau_2^-}$$

Fig. 8. Subtyping for $F^P_{\leq}$: $\boxed{\Gamma^- \vdash \tau^- \leq \tau^+}$

4.2 From $F^P_{\leq}$ to Julia

For this reduction, we shall use the invertible subtype relation of Fig. 6 and show that there exists a suitable contravariant translation $[\![\,]\!]$ of $F^P_{\leq}$ types and environments. Our translation is defined in Fig. 9. We use the nominal type constructors $\texttt{Neg}\langle\sigma\rangle$ and $\texttt{All}\langle\sigma,\sigma\rangle$ to create invariant contexts that force type equivalence. For simplicity, we treat $F^P_{\leq}$ type variables as Julia type variables.

$$\tau^+ \;\left|\begin{array}{ll} [\![\mathit{Top}]\!] & = \bot \\ [\![\neg\tau^-]\!] & = \texttt{Neg}\langle[\![\tau^-]\!]\rangle \\ [\![\forall\alpha{\leq}\tau^-.\tau^+]\!] & = \exists[\![\tau^-]\!]{<:}\alpha.\,\exists\alpha'{<:}[\![\tau^+]\!].\,\texttt{All}\langle\alpha,\alpha'\rangle \end{array}\right.$$

$$\tau^- \;\left|\begin{array}{ll} [\![\alpha]\!] & = \alpha \\ [\![\neg\tau^+]\!] & = \exists[\![\tau^+]\!]{<:}\alpha.\,\texttt{Neg}\langle\alpha\rangle \\ [\![\forall\alpha{\leq}\mathit{Top}.\tau^-]\!] & = \exists\alpha.\,\exists\alpha'{<:}[\![\tau^-]\!].\,\texttt{All}\langle\alpha,\alpha'\rangle \end{array}\right.$$

$$\Gamma^- \;\left|\begin{array}{ll} [\![\cdot]\!] & = \cdot \\ [\![\Gamma^-, \alpha{\leq}\tau^-]\!] & = [\![\Gamma^-]\!], [\![\tau^-]\!]{<:}\alpha \end{array}\right.$$

Fig. 9. Contravariant translation from $F^P_{\leq}$ to Julia

The key insight of the translation is that subtyping of universally quantified types is dual to subtyping of existentially quantified types. The translation flips an $F^P_{\leq}$ judgment $\Gamma^- \vdash \tau_1^- \leq \tau_2^+$ into $[\![\Gamma^-]\!] \vdash [\![\tau_2^+]\!] <:_{\text{JI}} [\![\tau_1^-]\!]$, replacing upper-bounded universal quantification with lower-bounded existential quantification. The translation targets invertible subtyping, rather than declarative subtyping. We made this choice since, for the translation to transfer undecidability, we need both that translation preserves subtyping ($\tau_1^- \leq \tau_2^+$ implies $[\![\tau_2^+]\!] <:_{\text{JI}} [\![\tau_1^-]\!]$) and that translation reflects subtyping ($[\![\tau_2^+]\!] <:_{\text{JI}} [\![\tau_1^-]\!]$ implies $\tau_1^- \leq \tau_2^+$). Showing that the translation reflects subtyping relies heavily on invertibility. This then leads us into the proof of undecidability.

Theorem 4.1 (Undecidability of Subtyping). *For any valid kind contexts Γ^- and any pair of negative type τ^- and positive type τ^+ valid in Γ^-, the following equivalence holds:*

$$\Gamma^- \vdash \tau_1^- \leq \tau_2^+ \iff [\![\Gamma^-]\!] \vdash [\![\tau_2^+]\!] <:_{\text{JI}} [\![\tau_1^-]\!]$$

Because the former is undecidable, this implies that the latter is also undecidable.

Proof. Consider the conclusion of rule $F^P_{\leq}$-All: $\Gamma^- \vdash \forall\alpha{\leq}\mathit{Top}.\tau_1^- \leq \forall\alpha{\leq}\tau_2^-.\tau^+$. The goal is to prove this holds if and only if the dual judgment holds: $[\![\Gamma^-]\!] \vdash [\![\forall\alpha{\leq}\tau_2^-.\tau^+]\!] <:_{\text{JI}} [\![\forall\alpha{\leq}\mathit{Top}.\tau_1^-]\!]$. Expanding the translation begets the following judgment:

$$[\![\Gamma^-]\!] \vdash \exists[\![\tau_2^-]\!]{<:}\alpha.\, \exists\alpha'{<:}[\![\tau^+]\!].\, \texttt{All}\langle\alpha,\alpha'\rangle <:_{\text{JI}} \exists\alpha.\, \exists\alpha'{<:}[\![\tau_1^-]\!].\, \texttt{All}\langle\alpha,\alpha'\rangle$$

The universal quantification is translated into an existentially quantified term of the same variable with upper bounds flipped into (translated) lower bounds, and with another variable with an upper bound in order to encode covariance. Because of the unique inversion properties of nominal constructors, having the body $\texttt{All}\langle\alpha,\alpha'\rangle$ for the right existential implies α and α' are necessarily instantiated with the corresponding arguments in the left type.[4] Thus the above judgment holds if and only if those instantiations satisfy their bounds in the appropriate kind context. Only α' has a bound, which invertibility then quickly implies is satisfied if and only if in the kind context $[\![\Gamma^-]\!], [\![\tau_2^-]\!]{<:}\alpha$ (since we can drop α' once it no longer occurs in either type), the following holds: $[\![\Gamma^-]\!], [\![\tau_2^-]\!]{<:}\alpha \vdash [\![\tau^+]\!] <:_{\text{JI}} [\![\tau_1^-]\!]$. This is the translation of the premise of $F^P_{\leq}$-All. Translation for negated types follows the same concept (dualization into existentially-bounded variables) but is simplified since there is no reference to the introduced type variable from within the translated terms. Rule $F^P_{\leq}$-Neg concludes with $\Gamma^- \vdash \neg\tau_1^+ \leq \neg\tau_2^-$, which translates to $[\![\Gamma^-]\!] \vdash \texttt{Neg}\langle[\![\tau_2^-]\!]\rangle <:_{\text{JI}} \exists[\![\tau_1^+]\!]{<:}\alpha.\, \texttt{Neg}\langle\alpha\rangle$. By reasoning similar to $F^P_{\leq}$-All, invertibility implies this holds if and only if α is instantiated with $[\![\tau_2^-]\!]$, though this time with the additional requirement that the instantiation satisfies its lower-bound constraint.[5] Thus this holds if and only if $[\![\Gamma^-]\!] \vdash [\![\tau_1^+]\!] <:_{\text{JI}} [\![\tau_2^-]\!]$ holds, which is the translation of the premise of $F^P_{\leq}$-Neg. □

Therefore, $F^P_{\leq}$ subtyping holds if and only if the dualized translated version holds. By Pierce [1992], we can conclude that our subset of Julia's subtype relation is undecidable. Chung [2023] generalizes undecidability onto the broader Julia subtype relation when considering the other subtyping features as described by Zappa Nardelli et al. [2018].

5 STRATIFYING EXISTENTIAL TYPES FOR JULIA-IN-PRACTICE

We identify a subset of Julia types within which subtyping is decidable and to which existing programs already conform. In particular, one can stratify types into method signatures over value types, where quantification in method signatures can only use value types for bounds, and quantification within value types is restricted to use-site variance. In the next section, we demonstrate that this stratification indeed makes subtyping decidable. We start with an empirical study of Julia programs to show they already conform to this stratification.

[4] Technically, invertibility only directly implies the instantiations are equivalent to the corresponding arguments. However, one can extend the intermediate type and proof system and the reduction process employed in the proof of Th. 3.1 to furthermore eliminate such equivalences and ensure direct instantiations. This extension adds a specialized form of existential quantification for precisely the above pattern, with its right rule restricted in the desired manner. Due to invariance of nominal constructors, transitivity-elimination reduction can be extended to accommodate this restriction while staying in line with PN2. And because α and α' have no lower bounds, there is no need to replace them with a union to integrate conservativity. Altogether, this means we can get a proof in the desired form for induction.

[5] This lower bound requires extending the aforementioned intermediate specialized quantifier to support either upper or lower bounds. So long as only one side is present, we still avoid the complications caused by nonconservative bounds.

$\psi ::=$	$(\tau, \ldots, \tau)$	tuple of method parameters
	$\exists \tau <: \mathrm{X} <: \tau.\, \psi$	polymorphism
	$\psi \cup \psi$	multipurposing
	$\bot$	exclusion
$\tau ::=$	$\top$	top
	$\bot$	bottom
	X	type variable
	$\tau \times \ldots \times \tau$	covariant tuple
	$\mathtt{C}\langle \tau \ll \tau, \ldots, \tau \ll \tau \rangle$	nominal constructor with use-site variance
	$\tau \cup \tau$	union
$\Theta ::=$	$\cdot$	empty kind context
	$\Theta, \tau <: \mathrm{X} <: \tau$	bounded variable

Fig. 10. Stratified types

5.1 Stratified Types

The stratified grammar is defined in Fig. 10. A method signature, ψ, is a union of existential quantifications of lists of value types as its method parameters. Thus, ψ can represent generic method signatures for the purpose of multiple dispatch. A value type, τ, is limited to use-site variance, rather than full existential quantification, but can use all other type constructors freely. A type argument $\tau_\ell \ll \tau_u$ for a nominal constructor indicates that the run-time argument must be a supertype of τ_ℓ and a subtype of τ_u. When a type argument is of the form $\tau \ll \tau$, we abbreviate it as τ. Upper and lower bounds, for both existentially quantified variables and type arguments, are always value types.

Julia already provides syntactic support for use-site variance. For the type `Vector{T} where T`, one can instead simply write `Vector`. Similarly, one can write types `Vector{<:Number}` and `Vector{>:Int32}` for, respectively, `Vector{T} where T<:Number` and `Vector{T} where T>:Int32`.

Crucially for the decidability of subtyping, our restriction rules out types where existential types inside invariant constructors do not match use-site variance, e.g., `Ref{Pair{T, T} where T}`. In particular, the `Theta`[6] type from Ghelli's looping gadget is not expressible as a stratified type because, in the existential `Kappa(Z)`[7], variable `Y` occurs twice under the invariant `Pair` constructor. Thus, `Kappa(Z)` (and, consequently, `Neg(Kappa(Z))`) is not encodable as use-site variance and does not constitute a value type, so it is not allowed as a bound in `Theta`.

In addition to syntactic stratification, we also require stratified types and kind contexts to be conservative, where lower bounds must be subtypes of upper bounds (for both variables and type arguments of datatypes). The validity judgments $\boxed{\Theta \vdash_S \psi}$, $\boxed{\Theta \vdash_S \tau}$, and $\boxed{\vdash_S \Theta}$ are all defined in the obvious manner, parameterized by a subtype relation on just value types (since method signatures do not occur in bounds).

There is an obvious syntactic definition for when an existential type corresponds to a nominal constructor with use-site variance (for example, $\exists \bot <: \mathrm{X} <: \top.\, \mathtt{Ref}\langle \mathrm{X} \rangle$ is $\mathtt{Ref}\langle \bot \ll \top \rangle$). This then extends to a syntactic correspondence between Julia types and value types, and then between Julia types and method signatures as well. Although, technically, multiple Julia types can correspond to a given stratified type, all of these types are trivially equivalent; so, as an abuse of notation, we treat

[6] `const Theta = Pair{Z, <:Neg(Kappa(Z))} where Z`
[7] `Kappa(T) = Pair{Y, <:Neg(Y)} where Y>:T`

stratified types as a subset of Julia types. We say a type conforms to stratification if there exists a stratified type that it corresponds to. We say a type conforms to stratification up to equivalence if it is equivalent—*according to Julia's entire feature set*—to some type that conforms to stratification. In particular, one can use distributivity to either pull a nested existential up to the top level or to push an existential down inside a tuple so that it exemplifies use-site variance. This is why method signatures have unions, whereas in practice just the method parameters would have unions that would be automatically distributed up to the signature level. We discuss such equivalences in more detail below.

5.2 Empirical Evaluation

To estimate the potential impact of imposing this stratification on existing programs, we conducted a corpus analysis over nearly all of the packages listed in the official Julia registry. Out of more than 2 million type annotations, only 4 do not conform to stratification up to equivalence. All type annotations have conservative bounds.

5.2.1 Methodology. The corpus used in our analysis is the entire *General Registry*, which is the default source of packages used by Julia programs. The list of packages, obtained from JuliaHub, contained 9,383 entries as of 2023-05-20. Out of those, 9,335 packages were successfully downloaded. Some entries were not valid registered packages, or were duplicates, or were no longer publicly available. The resulting corpus has 172K files with 16.5M lines of code as reported by CLOC 1.9.0. Our analysis code is written in Julia 1.8.5. It extracts type annotations from all `.jl` files in the corpus and reports annotations that do not conform to stratification up to equivalence. The extraction uses the Julia parser and the MacroTools.jl package for convenient pattern matching over abstract syntax trees. Source code and the results of the analysis are publicly available.[8]

5.2.2 Results. Some types do not directly conform to stratification, but they do up to equivalence. Such types are not flagged by the analysis because their equivalent rewriting could be automated. At the method-signature level, if a method parameter has an existential at a distributive location, we first distribute the existential up to the signature level. Then, at the value-type level, we employ equivalences from the following two categories:

- An existential variable essentially encoding use-site variance but separated from its binding by distributive constructors. Some examples are `Tuple{Vector{T}} where T` and `Union{Vector{T}, Missing} where T`. These types are equivalent to the stratified types `Tuple{Vector{T} where T}` and `Union{Vector{T} where T, Missing}`.
- An existential variable used completely unnecessarily as a single component of a tuple. For example, `Tuple{T} where T<:u` is equivalent to `Tuple{u}`. This category is already automatically rewritten by Julia into the equivalent existential-free form.

The analysis was run on our corpus. One package failed to process; we manually confirmed its types conform to stratification up to equivalence. There were 206 packages with at least one file failing to parse; such files were ignored. In the remaining files, the analysis identified a total of 2,283,011 type annotations. Out of these, 1,887 were not processed because a type-variable binding contained a macro or quoted expression, and 26,385 were partially processed due to a macro or quoted expression in a part of the type. Of the 2,281,124 fully-or-partially analyzable type annotations, 2,281,117 were identified as conforming to stratification up to equivalence, and 7 annotations were flagged as potentially problematic. Three of these seven annotations were false positives related to `Vararg`. Variadic arguments are represented as `Vararg` in `Tuple` types. For example, `Tuple{Vararg{Int32}}` stands for a tuple of arbitrarily many integers. According to

[8] https://github.com/prl-julia/julia-sub

Julia subtyping, `Vararg` is covariant in its type argument, whereas the analysis reported it as if it were invariant.

Of the remaining four type annotations, two are instances of the same non-conforming type that can be rewritten into a semantically equivalent type that conforms to stratification. The original type is the following from package Muon.jl:

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

This type describes dictionaries of arbitrary key type whose elements are either arrays of numbers, arrays of elements that are either of homogeneous numeric type or missing, or an arbitrary data frame. This type has the following semantically equivalent conforming type, but the equivalence is not derivable according to Julia subtyping:

```
Dict{ K, Union { AbstractArray{ <:Number},
                 AbstractArray{ Missing << Union{Missing,Number} }
                 DataFrame } }
```

The other two remaining types do not conform to stratification even up to semantic equivalence. The first type is the following from package Alicorn.jl:

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

Here we have an existential quantifier inside a nominal constructor, and its variable occurs more than once so that it expresses more than just use-site variance. This type requires that the array contains tuples where the type of the first projection matches the second projection's element type. The second type is the following from package UnitfulEquivalences.jl:

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

Here `T` and `U`, used by `Quantity`, are quantified outside of the containing `Level`.

To check for conservative bounds, we extracted all type annotations that explicitly declare both a lower and an upper bound on at least one variable. There were only 9 such annotations, all of which we inspected manually and found to be conservative.

In sum, our analysis shows that the types programmers write conform to our proposed stratification, or do so at least up to equivalences that are easy to recognize.

5.3 Stratified Subtyping

With our stratification empirically justified, we proceed to define invertible subtyping on stratified types, which we use as a bridge to invertible subtyping on arbitrary types. Fig. 11 gives the rules for method signatures, and Fig. 12 gives the rules for value types. Subtyping itself has become stratified; not only is method-signature subtyping layered over value-type subtyping, but for method signatures, the left rules are layered over the right rules, taking advantage of the stratification boundary to signal when to move from left to right. The right existential rule now only permits instantiation with value types τ, which we prove is faithful to subtyping between the corresponding

$$\frac{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} \psi'}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:_{\mathrm{SI}} \psi'} \qquad \frac{\forall i \in [1,n] \qquad \Theta \vdash \tau_i <:_{\mathrm{SI}} \tau_i'}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} (\tau_1', \ldots, \tau_n')}$$

$$\frac{}{\Theta \vdash \bot <:_{\mathrm{SI}} \psi'}$$

$$\frac{\Theta \vdash \psi_1 <:_{\mathrm{SI}} \psi' \qquad \Theta \vdash \psi_2 <:_{\mathrm{SI}} \psi'}{\Theta \vdash \psi_1 \cup \psi_2 <:_{\mathrm{SI}} \psi'} \qquad \frac{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} \psi_i'}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} \psi_1' \cup \psi_2'}$$

$$\frac{\Theta, \tau_\ell <: \mathrm{X} <: \tau_u \vdash \psi <:_{\mathrm{SI}} \psi'}{\Theta \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u . \psi <:_{\mathrm{SI}} \psi'} \qquad \frac{\Theta \vdash_{\mathrm{SI}} \tau_{\mathrm{X}} \qquad \Theta \vdash \tau_\ell <:_{\mathrm{SI}} \tau_{\mathrm{X}} \qquad \Theta \vdash \tau_{\mathrm{X}} <:_{\mathrm{SI}} \tau_u \qquad \Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} \psi'[\mathrm{X} \mapsto \tau_{\mathrm{X}}]}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SI}} \exists \tau_\ell <: \mathrm{X} <: \tau_u . \psi'}$$

Fig. 11. Stratified invertible subtyping for signatures: $\boxed{\Theta \vdash \psi <:_{\mathrm{SI}} \psi}$ (left), $\boxed{\Theta \vdash (\tau, \ldots) <:^{\mathrm{R}}_{\mathrm{SI}} \psi}$ (right)

$$\frac{}{\Theta \vdash \tau <:_{\mathrm{SI}} \top} \qquad \frac{}{\Theta \vdash \bot <:_{\mathrm{SI}} \tau}$$

$$\frac{}{\Theta \vdash \mathrm{X} <:_{\mathrm{SI}} \mathrm{X}} \qquad \frac{\tau_\ell <: \mathrm{X} <: \tau_u \in \Theta \qquad \Theta \vdash \tau_u <:_{\mathrm{SI}} \tau'}{\Theta \vdash \mathrm{X} <:_{\mathrm{SI}} \tau'} \qquad \frac{\tau_\ell <: \mathrm{X} <: \tau_u \in \Theta \qquad \Theta \vdash \tau <:_{\mathrm{SI}} \tau_\ell}{\Theta \vdash \tau <:_{\mathrm{SI}} \mathrm{X}}$$

$$\frac{\forall i \in [1,n] \qquad \Theta \vdash \tau_i <:_{\mathrm{SI}} \tau_i'}{\Theta \vdash \tau_1 \times \ldots \times \tau_n <:_{\mathrm{SI}} \tau_1' \times \ldots \times \tau_n'} \qquad \frac{\forall i \in [1,n] \qquad \Theta \vdash \tau_i^{\ell\prime} <:_{\mathrm{SI}} \tau_i^{\ell} \qquad \Theta \vdash \tau_i^{u} <:_{\mathrm{SI}} \tau_i^{u\prime}}{\Theta \vdash \mathtt{C}\langle \tau_1^{\ell} \ll \tau_1^{u} \ldots \tau_n^{\ell} \ll \tau_n^{u} \rangle <:_{\mathrm{SI}} \mathtt{C}\langle \tau_1^{\ell\prime} \ll \tau_1^{u\prime} \ldots \tau_n^{\ell\prime} \ll \tau_n^{u\prime} \rangle}$$

$$\frac{\Theta \vdash \tau_1 <:_{\mathrm{SI}} \tau' \qquad \Theta \vdash \tau_2 <:_{\mathrm{SI}} \tau'}{\Theta \vdash \tau_1 \cup \tau_2 <:_{\mathrm{SI}} \tau'} \qquad \frac{\Theta \vdash \tau <:_{\mathrm{SI}} \tau_i'}{\Theta \vdash \tau <:_{\mathrm{SI}} \tau_1' \cup \tau_2'}$$

Fig. 12. Stratified invertible subtyping for value types: $\boxed{\Theta \vdash \tau <:_{\mathrm{SI}} \tau}$

Julia types. The rule for nominal constructors now directly supports use-site variance without existential types. In particular, it ensures that, for each type argument, the left-hand use-site range is contained within the right-hand use-site range. For example, the subtyping $\vdash \mathtt{C}\langle \bot \ll \mathtt{Int32} \rangle <:_{\mathrm{SI}} \mathtt{C}\langle \bot \ll \top \rangle$ holds, whereas the subtyping $\nvdash \mathtt{C}\langle \bot \ll \mathtt{Int32} \rangle <:_{\mathrm{SI}} \mathtt{C}\langle \mathtt{Int32} \ll \top \rangle$ does not.

Lemma 5.1. *For any kind context Θ satisfying $\vdash_{\mathrm{SI}} \Theta$, and for any pair of value types τ and τ' satisfying $\Theta \vdash_{\mathrm{SI}} \tau$ and $\Theta \vdash_{\mathrm{SI}} \tau'$, the following equivalence holds:*

$$\Theta \vdash \tau <:_{\mathrm{JI}} \tau' \iff \Theta \vdash \tau <:_{\mathrm{SI}} \tau'$$

Theorem 5.2. *For any kind context Θ satisfying $\vdash_{\mathrm{SI}} \Theta$, and for any pair of method signatures ψ and ψ' satisfying $\Theta \vdash_{\mathrm{SI}} \psi$ and $\Theta \vdash_{\mathrm{SI}} \psi'$, the following equivalence holds:*

$$\Theta \vdash \psi <:_{\mathrm{JI}} \psi' \iff \Theta \vdash \psi <:_{\mathrm{SI}} \psi'$$

In addition to bridging invertible subtyping across Julia types and stratified types, Th. 3.1 extends that bridge to declarative subtyping. Consequently, we know that invertible subtyping for

conservative stratified types is reflexive and transitive, which will be useful for proving that our algorithm is complete.

6 DECIDING SUBTYPING FOR STRATIFIED EXISTENTIAL TYPES

At last, we present our algorithm for deciding subtyping of stratified types. Invertible subtyping brought us substantially closer than declarative subtyping by limiting the rules that can apply to any pair of types. Stratification brought us closer still by making recursive search terminating, as we illustrate next. The final step, addressed here, is to deal with the fact that certain rule applications still involve significant choices. In particular, the right existential rule for method signatures has to conjure an instantiating type. It is important to understand, though, that the algoritm is not particularly surprising; it is, for the most part, what one would expect for a constraint-collecting algorithm. It also aligns with Julia's implementation, aside from various heuristics, e.g. for picking which path to explore first. What is new is the insight that stratification and conservativity offer guarantees that ensure decidability and completeness. That is, stratification and conservativity explain why Julia's *existing* algorithm works well in practice.

In the following, we differentiate between *rigid* type variables, whose bounds are given by the kind context Θ, and *flexible* variables, on which constraints are collected and eventually solved to produce a corresponding instantiating type. The terminology carries over to types: a *rigid* (resp. *flexible*) type is a one that has only rigid (resp. flexible) type variables.

6.1 Backtracking Proof Search

Invertible subtyping for value types is decidable. The rules in Fig. 12 prescribe a backtracking proof search algorithm which is trivially sound and complete, *provided it terminates.* This latter condition is critical. While $F^P_{\leq}$ satisfies the requirements for backtracking proof search, the search can fail to terminate. In $<:_{\text{JI}}$-subtyping of value types, termination is ensured by a simple decreasing measure.

Lemma 6.1. *For any kind context Θ satisfying $\vdash^{\text{ws}} \Theta$, and for any pair of value types τ and τ' satisfying $\Theta \vdash^{\text{ws}} \tau$ and $\Theta \vdash^{\text{ws}} \tau'$, the following is decidable:*

$$\Theta \vdash \tau <:_{\text{SI}} \tau'$$

Consequently, we can simply define algorithmic subtyping on value types $\boxed{\Theta \vdash \tau <:_{\text{SA}} \tau}$ as invertible subtyping on value types $<:_{\text{SI}}$.

6.2 Marshalling Type Variables

Because of stratification, complexities typically associated with rigid and flexible variables (such as determining how to solve recursive constraints on flexible variables) are absent from our system.

Recall that stratified subtyping layers the left method-signature rules over the right method-signature rules, which, in turn, are layered over value-type subtyping. Since the only algorithmic rules that would need to introduce rigid or flexible variables for Julia are, respectively, the left and right existential rules, this means we can introduce all rigid variables, and then all flexible variables, and then proceed to value subtyping wherein neither get introduced. Consequently, only rigid variables occur in the left method signature and only flexible variables occur in the right method signature, which is a valuable property. While contravariance might cause left and right to swap, if we track such directionality, we can know whether a variable is rigid or flexible—and what we should do with it—simply by knowing the correct direction and which side of the subtyping the variable is occurring on. Furthermore, when the variable is flexible and we need to collect the other side as a constraint on that variable, the constraint is known to contain only rigid variables. In particular, this means that the constraint cannot be recursive. Thus, we can exploit stratification

$$\frac{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} \psi' \rightsquigarrow \varnothing}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:_{\mathrm{SA}} \psi'} \qquad \frac{\forall i \in [1, n] \qquad \Theta \vdash \tau_i <:^{\rightarrow}_{\mathrm{SA}} \tau_i' \rightsquigarrow \mathrm{K}_i}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} (\tau_1', \ldots, \tau_n') \rightsquigarrow \mathrm{K}_1 \cup \ldots \cup \mathrm{K}_n}$$

$$\frac{}{\Theta \vdash \bot <:_{\mathrm{SA}} \psi'}$$

$$\frac{\Theta \vdash \psi_1 <:_{\mathrm{SA}} \psi' \qquad \Theta \vdash \psi_2 <:_{\mathrm{SA}} \psi'}{\Theta \vdash \psi_1 \cup \psi_2 <:_{\mathrm{SA}} \psi'} \qquad \frac{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} \psi_i' \rightsquigarrow \mathrm{K}}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} \psi_1' \cup \psi_2' \rightsquigarrow \mathrm{K}}$$

$$\frac{\Theta, \tau_\ell <: \mathrm{X} <: \tau_u \vdash \psi <:_{\mathrm{SA}} \psi'}{\Theta \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u . \psi <:_{\mathrm{SA}} \psi'} \qquad \frac{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} \psi' \rightsquigarrow \mathrm{K_X} \quad \Theta \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u . \mathrm{K_X} \rightsquigarrow \mathrm{K}}{\Theta \vdash (\tau_1, \ldots, \tau_n) <:^{\mathrm{R}}_{\mathrm{SA}} \exists \tau_\ell <: \mathrm{X} <: \tau_u . \psi' \rightsquigarrow \mathrm{K}}$$

Fig. 13. Algorithmic subtyping for signatures: $\boxed{\Theta \vdash \psi <:_{\mathrm{SA}} \psi}$ (left) and $\boxed{\Theta \vdash (\tau, \ldots, \tau) <:^{\mathrm{R}}_{\mathrm{SA}} \psi \rightsquigarrow \mathrm{K}}$ (right)

to carefully marshal rigid and flexible variables and thereby prevent all major complexities for constraint solving.

Fig. 13 presents the algorithmic subtyping rules for method signatures. The left rules, interpreted using backtracking proof search, introduce rigid variables on the left until reaching method parameters. The right rules are then also interpreted using backtracking proof search, but with the constraint set K as an output of the search. These right rules introduce flexible variables on the right until reaching method parameters, at which point they defer to constraint-collecting value-type subtyping *with the direction superscript ($\rightarrow$) indicating the flexible variables are on the right.* After those constraints are collected, the respective backtracking proof search returns the resulting constraint set. Furthermore, the right existential rules each solve the constraints for the flexible variables they introduced—in reverse order—using the constraint-solving algorithm discussed in Section 6.4. Due to explicit flexible-variable bounds possibly referring to previously introduced flexible variables, constraint resolution can result in more constraints on those previously introduced variables.

Theorem 6.2. *For any pair of method signatures ψ and ψ' satisfying $\cdot \vdash^{\mathrm{WS}} \psi$ and $\cdot \vdash^{\mathrm{WS}} \psi'$, the following is decidable:*

$$\cdot \vdash \psi <:_{\mathrm{SA}} \psi'$$

Theorem 6.3. *For any pair of method signatures ψ and ψ' satisfying $\cdot \vdash_{\mathrm{SA}} \psi$ and $\cdot \vdash_{\mathrm{SA}} \psi'$, the following equivalence holds:*

$$\cdot \vdash \psi <:_{\mathrm{SI}} \psi' \iff \cdot \vdash \psi <:_{\mathrm{SA}} \psi'$$

6.3 Directed Constraint Collection

After method-signature subtyping takes care of introducing, and later solving, all variables, only value types are needed to determine the constraints flexible variables need to satisfy. The constraint sets K collected during this process are finite sets of constraints of either the form $\mathrm{X} \geq \tau_\ell$ or $\mathrm{X} \leq \tau_u$, where X is a flexible variable and τ_ℓ and τ_u are rigid types. In order to maintain these invariants, constraint-collecting subtyping is directed. A direction δ is either $\leftarrow$ or $\rightarrow$, with the arrow pointing from the side with the rigid type to the side with the flexible type. In the case of contravariance, the direction is reversed, denoted $-\delta$, in the obvious manner. Using these new concepts, the rules for constraint-collecting algorithmic subtyping for value types are presented in Fig. 14.

$$\frac{}{\Theta \vdash \tau <:^{\delta}_{\text{SA}} \top \rightsquigarrow \varnothing} \qquad \frac{}{\Theta \vdash \bot <:^{\delta}_{\text{SA}} \tau \rightsquigarrow \varnothing}$$

$$\frac{}{\Theta \vdash \mathrm{X} <:^{\leftarrow}_{\text{SA}} \tau \rightsquigarrow \mathrm{X} \leq \tau} \qquad \frac{}{\Theta \vdash \tau <:^{\rightarrow}_{\text{SA}} \mathrm{X} \rightsquigarrow \mathrm{X} \geq \tau}$$

$$\frac{\tau_\ell <: \mathrm{X} <: \tau_u \in \Theta \qquad \Theta \vdash \tau_u <:^{\rightarrow}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}}{\Theta \vdash \mathrm{X} <:^{\rightarrow}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}} \qquad \frac{\tau_\ell <: \mathrm{X} <: \tau_u \in \Theta \qquad \Theta \vdash \tau <:^{\leftarrow}_{\text{SA}} \tau_\ell \rightsquigarrow \mathrm{K}}{\Theta \vdash \tau <:^{\leftarrow}_{\text{SA}} \mathrm{X} \rightsquigarrow \mathrm{K}}$$

$$\frac{\forall i \in [1,n] \qquad \Theta \vdash \tau_i <:^{\delta}_{\text{SA}} \tau_i' \rightsquigarrow \mathrm{K}_i}{\Theta \vdash \tau_1 \times \ldots \times \tau_n <:^{\delta}_{\text{SA}} \tau_1' \times \ldots \times \tau_n' \rightsquigarrow \mathrm{K}_1 \cup \ldots \cup \mathrm{K}_n}$$

$$\frac{\forall i \in [1,n] \qquad \Theta \vdash \tau_i^{\ell\prime} <:^{-\delta}_{\text{SA}} \tau_i^{\ell} \rightsquigarrow \mathrm{K}_i^{\ell} \qquad \Theta \vdash \tau_i^{u} <:^{\delta}_{\text{SA}} \tau_i^{u\prime} \rightsquigarrow \mathrm{K}_i^{u}}{\Theta \vdash \mathtt{C}\langle \tau_1^{\ell} \ll \tau_1^{u} \ldots \tau_n^{\ell} \ll \tau_n^{u} \rangle <:^{\delta}_{\text{SA}} \mathtt{C}\langle \tau_1^{\ell\prime} \ll \tau_1^{u\prime} \ldots \tau_n^{\ell\prime} \ll \tau_n^{u\prime} \rangle \rightsquigarrow \mathrm{K}_1^{\ell} \cup \mathrm{K}_1^{u} \cup \ldots \cup \mathrm{K}_n^{\ell} \cup \mathrm{K}_n^{u}}$$

$$\frac{\Theta \vdash \tau_1 <:^{\delta}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}_1 \qquad \Theta \vdash \tau_2 <:^{\delta}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}_2}{\Theta \vdash \tau_1 \cup \tau_2 <:^{\delta}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}_1 \cup \mathrm{K}_2} \qquad \frac{\Theta \vdash \tau <:^{\delta}_{\text{SA}} \tau_i \rightsquigarrow \mathrm{K}}{\Theta \vdash \tau <:^{\delta}_{\text{SA}} \tau_1 \cup \tau_2 \rightsquigarrow \mathrm{K}}$$

Fig. 14. Constraint-collecting algorithmic subtyping for value types: $\boxed{\Theta \vdash \tau <:^{\delta}_{\text{SA}} \tau \rightsquigarrow \mathrm{K}}$

These rules have a clear correspondence with (non-constraint-collecting) algorithmic (i.e. invertible) subtyping. Most of them simply furthermore propagate the constraints collected from the premises. The only interesting rules are those for variables. If the direction points to the variable, then the corresponding constraint on that necessarily flexible variable is generated—this is the only way in which constraints are introduced. Otherwise, if the direction points away from the variable, the appropriate bound on that necessarily rigid variable is employed.

In order to discuss the properties of constraint collection formally, we need to introduce notions of assignments, substitutions, and satisfaction. An assignment θ is a finite partial mapping of type variables X to value types $\theta(\mathrm{X})$. Assignments extend to substitutions on value types $\tau[\theta]$ in the obvious manner. An assignment between kind contexts is one that satisfies $\boxed{\vdash \theta : \Theta_F \to \Theta_R}$ if, for each $\tau_\ell <: \mathrm{X} <: \tau_u \in \Theta_F$, a corresponding type $\theta(\mathrm{X})$ exists and is conservative in Θ_R (i.e. $\Theta_R \vdash_{\text{SA}} \theta(\mathrm{X})$ holds) and lies between its substituted bounds (i.e. $\Theta_R \vdash \tau_\ell[\theta] <:_{\text{SA}} \theta(\mathrm{X})$ and $\Theta_R \vdash \theta(\mathrm{X}) <:_{\text{SA}} \tau_u[\theta]$ hold), where Θ_F and Θ_R bind flexible and rigid variables, respectively.

A constraint set is valid, $\boxed{\Theta_R \vdash \mathrm{K} \dashv \Theta_F}$, when the bound variable in each constraint is declared in Θ_F and the constraining type in each constraint is rigid (i.e. conservative in Θ_R). An assignment θ from Θ_F to Θ_R satisfies a constraint set, $\boxed{\Theta_R \vdash_{\text{SA}} \theta \dashv \mathrm{K}}$, if for each constraint the value type assigned to the variable by θ is an algorithmic supertype/subtype of the constraining type in Θ_R.

Lemma 6.4. *For any kind contexts Θ_R and Θ_F satisfying $\vdash_{\text{SA}} \Theta_R$ and $\vdash_{\text{SA}} \Theta_F$, for any pair of value types τ and τ' satisfying $\Theta_R \vdash_{\text{SA}} \tau$ and $\Theta_F \vdash_{\text{SA}} \tau'$, the following hold:*

Marshalling *Any constraint set* K *satisfying* $\Theta_R \vdash \tau <:^{\rightarrow}_{\text{SA}} \tau' \rightsquigarrow \mathrm{K}$ *is valid, i.e.* $\Theta_R \vdash \mathrm{K} \dashv \Theta_F$ *holds.*

Soundness *For any constraint set* K *satisfying* $\Theta_R \vdash \tau <:^{\rightarrow}_{\text{SA}} \tau' \rightsquigarrow \mathrm{K}$, *any assignment θ satisfying* $\vdash \theta : \Theta_F \to \Theta_R$ *and* $\Theta_R \vdash_{\text{SA}} \theta \dashv \mathrm{K}$ *also satisfies* $\Theta_R \vdash \tau <:_{\text{SA}} \tau'[\theta]$.

$$\frac{\begin{array}{c}\forall \mathrm{X} \geq \tau_\ell^{\prime} \in \mathrm{K}, \mathrm{X} \leq \tau_u^{\prime} \in \mathrm{K}. \qquad \Theta \vdash \tau_\ell^{\prime} <:_{\mathrm{SA}} \tau_u^{\prime} \\ \forall \mathrm{X} \geq \tau_\ell^{\prime} \in \mathrm{K}. \qquad \Theta \vdash \tau_\ell^{\prime} <:^{\rightarrow}_{\mathrm{SA}} \tau_u \rightsquigarrow \mathrm{K}^{\ell}_{\tau_\ell^{\prime}} \qquad \forall \mathrm{X} \leq \tau_u^{\prime} \in \mathrm{K}. \qquad \Theta \vdash \tau_\ell <:^{\leftarrow}_{\mathrm{SA}} \tau_u^{\prime} \rightsquigarrow \mathrm{K}^{u}_{\tau_u^{\prime}} \\ \mathrm{K}_\ell = \{\mathrm{X}^{\prime} \geq \tau_\ell^{\prime} \in \mathrm{K} \mid \mathrm{X}^{\prime} \neq \mathrm{X}\} \qquad \mathrm{K}_u = \{\mathrm{X}^{\prime} \leq \tau_u^{\prime} \in \mathrm{K} \mid \mathrm{X}^{\prime} \neq \mathrm{X}\}\end{array}}{\Theta \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}_\ell \bigcup \mathrm{K}_u \bigcup_{\mathrm{X} \geq \tau_\ell^{\prime} \in \mathrm{K}} \mathrm{K}^{\ell}_{\tau_\ell^{\prime}} \bigcup_{\mathrm{X} \leq \tau_u^{\prime} \in \mathrm{K}} \mathrm{K}^{u}_{\tau_u^{\prime}}}$$

Fig. 15. Constraint solving: $\boxed{\Theta \vdash \exists \tau <: \mathrm{X} <: \tau.\, \mathrm{K} \rightsquigarrow \mathrm{K}}$

Completeness *For any assignment θ satisfying $\vdash \theta : \Theta_F \to \Theta_R$, if $\Theta_R \vdash \tau <:_{\mathrm{SA}} \tau'[\theta]$ holds then there exists a constraint set K such that $\Theta_R \vdash \tau <:^{\rightarrow}_{\mathrm{SA}} \tau' \rightsquigarrow \mathrm{K}$ and $\Theta_R \vdash_{\mathrm{SA}} \theta \dashv \mathrm{K}$ hold.*

Computability *The set of constraint sets K satisfying $\Theta_R \vdash \tau <:^{\rightarrow}_{\mathrm{SA}} \tau' \rightsquigarrow \mathrm{K}$ is finite and computably enumerable.*

Likewise for the opposite direction ($\leftarrow$), though with τ and τ' satisfying $\Theta_F \vdash_{\mathrm{SA}} \tau$ and $\Theta_R \vdash_{\mathrm{SA}} \tau'$ instead.

6.4 Constraint Solving

If constrained subtyping succeeds and generates a constraint set K, the constraints on the most recently introduced unsolved flexible variable X are then solved by employing backtracking proof search on the rule for the judgment $\Theta \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}'$ given in Fig. 15. To understand the design of this rule, it is important to be mindful of where rigid and flexible variables can and cannot occur. All types that are constraining variables in K necessarily contain only rigid variables. On the other hand, the bounds τ_ℓ and τ_u of X necessarily contain only flexible variables (and do not contain X).

Each premise of the rule in Fig. 15 corresponds to a step of the (backtracking) algorithm:

(1) For each pair of (necessarily rigid) collected lower bound $\tau_\ell^{\prime}$ and collected upper bound $\tau_u^{\prime}$ on X in K, fail unless $\tau_\ell^{\prime}$ is a subype of $\tau_u^{\prime}$, since transitivity implies this must hold for any instantiation of X.
(2) For each (necessarily rigid) collected lower bound $\tau_\ell^{\prime}$ on X in K, let $\mathrm{K}^{\ell}_{\tau_\ell^{\prime}}$ be a constraint set collected from checking that $\tau_\ell^{\prime}$ is a subtype of the (necessarily flexible) given upper bound τ_u of X, which again transitivity implies must hold for any instantiation of X.
(3) For each (necessarily rigid) collected upper bound $\tau_u^{\prime}$ on X in K, let $\mathrm{K}^{u}_{\tau_u^{\prime}}$ be a constraint set collected from checking that the (necessarily flexible) given lower bound τ_ℓ of X is a subtype of $\tau_u^{\prime}$, which again transitivity implies must hold for any instantiation of X.
(4) Let K_ℓ be the set of (necessarily rigid) collected lower bounds on variables other than X, whose rigidity ensures they do not contain X.
(5) Let K_u be the set of (necessarily rigid) collected upper bounds on variables other than X, whose rigidity ensures they do not contain X.

Then the algorithm returns the union of all the constructed constraint sets. One might be surprised that the algorithm never actually constructs a satisfying instantiation of X. This is because stratification and consistency were able to ensure that none of the remaining constraints contain X, and so instantiating it would have no effect on the constraint set. Nonetheless, instantiating X with the union of its given and collected lower bounds necessarily satisfies its constraints (relying on the fact that consistency ensures its given lower bound is a subtype of its given upper bound), and as such the following holds.

Lemma 6.5. *For any kind contexts Θ_R and Θ_F satisfying $\vdash_{\text{SA}} \Theta_R$ and $\vdash_{\text{SA}} \Theta_F$, for any pair of value types τ_ℓ and τ_u satisfying $\Theta_F \vdash_{\text{SA}} \tau_\ell$ and $\Theta_F \vdash_{\text{SA}} \tau_u$ and $\Theta_F \vdash \tau_\ell <:_{\text{SA}} \tau_u$, for any type variable X not declared by either Θ_R or Θ_F, and for any constraint set K satisfying $\Theta_R \vdash \mathrm{K} \dashv \Theta_F, \tau_\ell <: \mathrm{X} <: \tau_u$, the following hold:*

Marshalling *Any constraint set K' satisfying $\Theta_R \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}'$ is valid without X, i.e. $\Theta_R \vdash \mathrm{K}' \dashv \Theta_F$ holds.*

Soundness *For any constraint set K' satisfying $\Theta_R \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}'$ and any assignment θ satisfying $\vdash \theta : \Theta_F \to \Theta_R$, if $\Theta_R \vdash_{\text{SA}} \theta \dashv \mathrm{K}'$ holds then there exists a value type τ_{X} satisfying $\Theta_R \vdash_{\text{SA}} \tau_{\mathrm{X}}$ and $\Theta_R \vdash \tau_\ell[\theta] <:_{\text{SA}} \tau_{\mathrm{X}}$ and $\Theta_R \vdash \tau_{\mathrm{X}} <:_{\text{SA}} \tau_u[\theta]$ such that $\Theta_R \vdash_{\text{SA}} \theta, \mathrm{X} \mapsto \tau_{\mathrm{X}} \dashv \mathrm{K}$ holds.*

Completeness *For any assignment θ satisfying $\vdash \theta : \Theta_F \to \Theta_R$ and any value type τ_{X} satisfying $\Theta_R \vdash_{\text{SA}} \tau_{\mathrm{X}}$ and $\Theta_R \vdash \tau_\ell[\theta] <:_{\text{SA}} \tau_{\mathrm{X}}$ and $\Theta_R \vdash \tau_{\mathrm{X}} <:_{\text{SA}} \tau_u[\theta]$, if $\Theta_R \vdash_{\text{SA}} \theta, \mathrm{X} \mapsto \tau_{\mathrm{X}} \dashv \mathrm{K}$ holds then there exists a constraint set K' satisfying $\Theta_R \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}'$ and $\Theta_R \vdash_{\text{SA}} \theta \dashv \mathrm{K}'$.*

Computability *The set of constraint sets K' satisfying $\Theta_R \vdash \exists \tau_\ell <: \mathrm{X} <: \tau_u.\, \mathrm{K} \rightsquigarrow \mathrm{K}'$ is finite and computably enumerable.*

6.5 Example of Constraint Collection and Solving

To illustrate how the algorithm works more concretely, consider the subtyping between signatures $(\texttt{String}, \texttt{Ref}\langle\texttt{Int}\rangle)$ and $\exists \bot <: \mathrm{X} <: \top.\, \exists \bot <: \mathrm{Y} <: \mathrm{X}.\, (\mathrm{X}, \texttt{Ref}\langle\mathrm{Y}\rangle)$. This subtyping should hold because X and Y can be instantiated with value types $\texttt{String} \cup \texttt{Int}$ and $\texttt{Int}$. The following derivation illustrates the generation of the necessary constraints on type variables X and Y.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \texttt{String} <:^{\rightarrow}_{\text{SA}} \mathrm{X} \rightsquigarrow \mathrm{X} \geq \texttt{String}} \quad \dfrac{\dfrac{}{\vdash \texttt{Int} <:^{\rightarrow}_{\text{SA}} \mathrm{Y} \rightsquigarrow \mathrm{Y} \leq \texttt{Int}} \quad \dfrac{}{\vdash \mathrm{Y} <:^{\leftarrow}_{\text{SA}} \texttt{Int} \rightsquigarrow \mathrm{Y} \geq \texttt{Int}}}{\vdash \texttt{Ref}\langle\texttt{Int}\rangle <:^{\rightarrow}_{\text{SA}} \texttt{Ref}\langle\mathrm{Y}\rangle \rightsquigarrow \{\mathrm{Y} \geq \texttt{Int}, \mathrm{Y} \leq \texttt{Int}\}}}{\vdash (\texttt{String}, \texttt{Ref}\langle\texttt{Int}\rangle) <:^{\text{R}}_{\text{SA}} (\mathrm{X}, \texttt{Ref}\langle\mathrm{Y}\rangle) \rightsquigarrow \{\mathrm{X} \geq \texttt{String}, \mathrm{Y} \geq \texttt{Int}, \mathrm{Y} \leq \texttt{Int}\}}\,(b)}{\vdash (\texttt{String}, \texttt{Ref}\langle\texttt{Int}\rangle) <:^{\text{R}}_{\text{SA}} \exists \bot <: \mathrm{Y} <: \mathrm{X}.\, (\mathrm{X}, \texttt{Ref}\langle\mathrm{Y}\rangle) \rightsquigarrow \{\mathrm{X} \geq \texttt{String}, \mathrm{X} \geq \texttt{Int}\}}\,(a)}{\vdash (\texttt{String}, \texttt{Ref}\langle\texttt{Int}\rangle) <:^{\text{R}}_{\text{SA}} \exists \bot <: \mathrm{X} <: \top.\, \exists \bot <: \mathrm{Y} <: \mathrm{X}.\, (\mathrm{X}, \texttt{Ref}\langle\mathrm{Y}\rangle) \rightsquigarrow \varnothing}}{\vdash (\texttt{String}, \texttt{Ref}\langle\texttt{Int}\rangle) <:_{\text{SA}} \exists \bot <: \mathrm{X} <: \top.\, \exists \bot <: \mathrm{Y} <: \mathrm{X}.\, (\mathrm{X}, \texttt{Ref}\langle\mathrm{Y}\rangle)}$$

Since there is no existential quantification on the left, subtyping starts by opening right existential types (X first, then Y) until method parameters are reached. For value types, directed constraint-collecting subtyping $<:^{\rightarrow}_{\text{SA}}$ generates constraints on flexible variables X and Y. Initially, flexible variables appear only on the right of the judgment, and thus, $<:^{\rightarrow}_{\text{SA}}$ used for tuple components points to the right; furthermore, constraints themselves are free from flexible variables due to constraining types coming from the opposite side of the judgment. Whenever constraint-collecting subtyping hits an invariant constructor, flexible variables move to the opposite side in one half of checking the required equivalence, which is why left-facing $<:^{\leftarrow}_{\text{SA}}$ is used for $\vdash \mathrm{Y} <:^{\leftarrow}_{\text{SA}} \texttt{Int} \rightsquigarrow \mathrm{Y} \geq \texttt{Int}$.

Once all constraints induced by subtyping of value types are collected, they are solved for the innermost flexible variable, Y in the example, incorporating its explicitly given bounds. Here, *(b)* denotes a constraint-solving step for the innermost variable Y:

$$\dfrac{\vdash \texttt{Int} <:_{\text{SA}} \texttt{Int} \qquad \vdash \bot <:^{\leftarrow}_{\text{SA}} \texttt{Int} \rightsquigarrow \varnothing \qquad \vdash \texttt{Int} <:^{\rightarrow}_{\text{SA}} \mathrm{X} \rightsquigarrow \mathrm{X} \geq \texttt{Int}}{\vdash \exists \bot <: \mathrm{Y} <: \mathrm{X}.\, \{\mathrm{X} \geq \texttt{String}, \mathrm{Y} \geq \texttt{Int}, \mathrm{Y} \leq \texttt{Int}\} \rightsquigarrow \{\mathrm{X} \geq \texttt{String}, \mathrm{X} \geq \texttt{Int}\}}$$

It checks that the (single) collected lower bound is a subtype of the (single) collected upper bound, the given lower bound is a subtype of the (single) collected upper bound, and the (single) collected

lower bound is a subtype of the given upper bound. In the first check, both types are necessarily rigid, so no contraint-collection is performed. The second check succeeds without additional constraints. But the third check imposes a new constraint on X, which is added to the collection of preexisting constraints on X. These checks confirm that Y can be correctly instantiated as the union of its given and (single) collected lower bound: $\bot \cup \texttt{Int}$.

Next, *(a)* denotes a constraint-solving step for the outermost variable X:

$$\frac{\vdash \texttt{String} <:^{\rightarrow}_{\text{SA}} \top \rightsquigarrow \varnothing \qquad \vdash \texttt{Int} <:^{\rightarrow}_{\text{SA}} \top \rightsquigarrow \varnothing}{\vdash \exists \bot <: \text{X} <: \top.\, \{\text{X} \geq \texttt{String}, \text{X} \geq \texttt{Int}\} \rightsquigarrow \varnothing}$$

It checks that the (two) collected lower bounds are subtypes of the given upper bound, which both succeed without additional constraints (as would always be the case when solving the final flexible variable), and then it has no more checks to perform because there are no collected upper bounds on X. These checks confirm that X can be correctly instantiated as the union of its given and collected lower bounds: $\bot \cup \texttt{String} \cup \texttt{Int}$.

After the outermost variable, the invariants granted by stratification and consistency ensure the resulting constraint set is empty. So, if this point is reached by the backtracking proof search, the subtyping between method signatures necessarily holds.

6.6 Sound and Complete Subtyping Algorithm

Putting all the pieces together, we show that algorithmic subtyping for method signatures provides a sound and complete decision procedure for declarative subtyping on conservative stratified types.

Theorem 6.6 (Decidability of Conservative and Stratified Julia). *For any pair of method signatures ψ and ψ' satisfying $\cdot \vdash_{\text{SA}} \psi$ and $\cdot \vdash_{\text{SA}} \psi'$, the following is decidable:*

$$\cdot \vdash \psi <:^{\text{C}}_{\text{JD}} \psi'$$

Proof. By Th. 3.1, Th. 5.2, and Th. 6.3, the above subtyping holds iff $\cdot \vdash \psi <:_{\text{SA}} \psi'$. By Th. 6.2, the latter is decidable. □

7 RELATED WORK

Designing decidable subtyping for production languages is challenging. Recent results include proofs of undecidability for Java generics by Grigore [2017] and Scala 3 by Hu and Lhoták [2019]. Expressiveness and decidability exist in a trade-off space. Users may prefer more expressive types even if the compiler may fail, as long as failures are rare cases. Failures that can manifest at run-time are more serious. Mainstream languages with subtyping usually restrict run-time subtype queries [Kennedy and Pierce 2007]. This is not so in Julia, as its full subtype relation is exercised at run-time.

Aiming for decidability, Julia's designers deliberately avoided features already established to be problematic, such as F-bounded polymorphism, contravariant nominal constructors, and multiple inheritance. Based on those restrictions, Bezanson [2015] conjectured decidability. However, he did point out that the combination of nominal constructors and contravariance in lower bounds of existential types is akin to the source of undecidability in $F_{\leq}$, which we have now formally established.

System $F_{\leq}$. Introduced by Cardelli et al. [1991], System $F_{\leq}$ combines System F and subtyping. As already mentioned, $F_{\leq}$ provides bounded universal quantification, whereas Julia provides bounded existential quantification. This difference is dual in nature and so not particularly impactful upon algorithmic concerns. However, there is another much more algorithmically-impactful difference: subtyping in $F_{\leq}$ is restricted so that the only subtypes/supertypes of universally quantified types

are universally quantified types—that is, quantifications must align. There is a good reason for this difference: unlike Julia, $F_{\leq}$ supports explicit type application, which only makes sense if quantifications stay aligned. Regardless of the reason, this difference means $F_{\leq}$ avoids a major challenge that Julia faces: whereas Julia needs to find a suitable instantiating type for a quantified variable, in $F_{\leq}$ that instantiating type is explicitly restricted to be the quantified variable of the other type. This makes $F_{\leq}$'s invertible subtype truly syntax-directed, whereas Julia must resort to collecting and solving constraints on flexible variables.

Yet, despite $F_{\leq}$'s restrictive subtyping, Pierce [1992] proved it can encode the halting problem for two-counter machines [Hopcroft and Ullman 1990] and therefore is undecidable. Yet there are variations of $F_{\leq}$ for which subtyping is decidable.

Restricting Subtyping. Kernel $F_{\leq}$ [Cardelli and Wegner 1985] forces even more alignment between subtypes: not only must quantifications align, they must have the identical bounds as well. This restriction is decidable, though rather limiting. Hu and Lhoták [2019] and Mackay et al. [2019] have shown that one can relax this by splitting the kind context into left and right parts so that, effectively, each bound is only used by the type that introduced it. However, this causes subtyping to no longer satisfy transitivity.

Restricting Types. Instead of restricting subtyping, Mackay et al. [2020] achieved decidability while retaining transitivity by instead restricting types. In fact, their solution is to stratify types into impredicative and predicative layers. The impredicative layer has more restrictive subtyping, akin to Kernel $F_{\leq}$, whereas the predicative layer has more restrictive types. It is interesting that we arrived at a similar stratification though for a very different reason. In particular, we discovered that stratification eliminated the complexities Julia faced in constraint solving by, in particular, entirely avoiding the possibility of recursive constraints. On the other hand, Mackay et al. [2020] still rely on the quantification alignment inherent in $F_{\leq}$ and as such have no concern for constraint solving. This difference in concerns explains the difference in where stratification is imposed in the two systems.

Beyond $F_{\leq}$, this approach of restricting types has been applied to prior practical systems as well. Kennedy and Pierce [2007] identified three decidable fragments of undecidable subtyping in the context of nominal inheritance and variance: the fragments can be obtained by restricting either contravariance, expansive class tables, or multiple-instantiation inheritance. Greenman et al. [2014] proposed a material-shape separation for Java generics that recovers decidability: it limits F-bounded polymorphism to the subset of types, called shapes, used exclusively as constraints. As we have, they conducted a corpus analysis to demonstrate that this restriction would be compatible with how programmers were using types in practice. Mackay et al. [2019] extend the material-shape separation to path-dependent and recursive types.

Bounded Existentials. In Java, a variable of type `List<? extends Number>` can be assigned any list whose elements are a subtype of `Number`. The "?" is known as a wildcard, and this wildcard-typed list effectively represents $\exists$`X<:Number.List<X>`. The wildcard mechanism of Java generics [Torgersen et al. 2004] is an encoding of use-site variance [Igarashi and Viroli 2002; Krab Thorup and Torgersen 1999], which is another widely used restricted form of bounded existential types. There have been multiple formalizations of Java wildcards [Cameron et al. 2008; Torgersen et al. 2005], though they focused on type soundness rather than decidability. 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. Wehr and Thiemann [2009] identified multiple undecidable subtype relations for bounded existential types in formal models inspired by Java. Tate et al. [2011] highlighted multiple sources of non-termination in Java

subtyping, e.g. recursive constraints on type variables and wildcards in the inheritance hierarchy. With some practical restrictions, they provide a terminating subtyping algorithm, though this has been mostly superseded by the aforementioned material-shape separation [Greenman et al. 2014].

Rather than encode use-site variance, Morrisett et al. [1998]'s typed assembly language (TAL) uses constrained existential types to track that the unknown exact type of a closure can be passed to the code pointer extracted from that closure. Early works required manually coercing between existential types, but Tate et al. [2010] developed a system with decidable subtyping for and even inferability of its existential types. However, the algorithmic framework they developed [Tate et al. 2008] relies heavily on rigidly structured types, and they illustrate how even $\bot$-like types (such as the type of a null pointer) cause fundamental problems due to violating this structure.

Subtype Constraints. Constraint generation and solving techniques are used in type inference. Most famously, equality constraints and unification were employed by Hindley [1969] and Damas and Milner [1982] in the context of a functional language with parametric polymorphism. With subtyping, equality constraints become subtype constraints. For example, Aiken and Wimmers [1993] extended Damas–Hindley–Milner inference with subtyping for recursive, union, and intersection types, and gave an algorithm for solving a system of constraints with restricted union and intersection types. Trifonov and Smith [1996] considered polymorphic types with explicit recursive constraints on type variables: they studied a corresponding subtype relation and provided its decidable approximation. Later, Pottier [1998] demonstrated how to improve the performance of inference with subtype constraints. Bourdoncle and Merz [1997] used a restricted form of constrained polymorphic types in a language with multi-methods and decidable subtyping. Castagna et al. [2015] dealt with subtype constraints for set-theoretic types with negation types. Chandra et al. [2016] and Chaudhuri et al. [2017] tackled flow-sensitive type inference with unusual constraint languages going beyond typical subtype constraints.

8 CONCLUSION

Decidability of Julia subtyping can be recovered by restricting types to a stratified grammar for the core of Julia's type-annotation language, most importantly, bounded existential types. This restriction is practical, as the vast majority of Julia programs already conform to it. However, the formalism presented here is still incomplete; it is missing types such as variadic arguments, and it is missing rules such as distributivity. More work needs to be done to develop a sound and complete algorithm for Julia's entire feature set.

ACKNOWLEDGMENTS

We thank the PLDI and POPL reviewers for their suggestions and feedback. This work was supported by the Czech Ministry of Education, Youth and Sports under program ERC-CZ, grant agreement LL2325, NSF grants CCF-1910850, CNS-1925644, and CCF-2139612, as well as the GACR EXPRO grant 23-07580X.

REFERENCES

Alexander Aiken and Edward L. Wimmers. 1993. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*. https://doi.org/10.1145/165180.165188

Julia Belyakova. 2023. *Decidable Subtyping of Existential Types for the Julia Language.* Ph. D. Dissertation. Northeastern University. https://onesearch.library.northeastern.edu/permalink/01NEU_INST/87npqb/cdi_proquest_journals_2853689755

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

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 (2018). https://doi.org/10.1145/3276490

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

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/28697.28700

Francois Bourdoncle and Stephan Merz. 1997. Type checking higher-order polymorphic multi-methods. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263743

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

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An Extension of System F with Subtyping. In *Theoretical Aspects of Computer Software (TACS)*. https://doi.org/10.1007/3-540-54415-1_73

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (1985). https://doi.org/10.1145/6041.6042

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Principles and Practice of Declarative Programming (PPDP)*. https://doi.org/10.1145/1069774.1069793

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

Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type inference for static compilation of JavaScript. https://doi.org/10.1145/2983990.2984017

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). https://doi.org/10.1145/3133872

Benjamin Chung. 2023. *A type system for Julia.* Ph. D. Dissertation. Northeastern University. https://arxiv.org/abs/2310.16866

Benjamin Chung, Francesco Zappa Nardelli, and Jan Vitek. 2019. Julia's Efficient Algorithm for Subtyping Unions and Covariant Tuples. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.24

Pierre-Louis Curien and Giorgio Ghelli. 1990. Coherence of subsumption. In *Colloquium on Trees in Algebra and Programming (CAAP)*. https://doi.org/10.1007/3-540-52590-4_45

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/582153.582176

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* (1987). https://doi.org/10.1016/0304-3975(87)90045-4

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-Bounded Polymorphism into Shape. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594308

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

Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* (1969). https://doi.org/10.2307/1995158

John Hopcroft and Jeffrey Ullman. 1990. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley.

Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of D<: And Its Decidable Fragments. *Proc. ACM Program. Lang.* 4, POPL (2019). https://doi.org/10.1145/3371077

Atsushi Igarashi and Mirko Viroli. 2002. On Variance-Based Subtyping for Parametric Types. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/3-540-47993-7_19

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

Kresten Krab Thorup and Mads Torgersen. 1999. Unifying Genericity. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/3-540-48743-3_9

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2019. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.* 4, POPL (2019). https://doi.org/10.1145/3371134

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems (APLAS)*. https://doi.org/10.1007/978-3-030-64437-6_7

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/268946.268954

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 (2021). https://doi.org/10.1145/3485527

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

Franc ois Pottier. 1998. A framework for type inference with subtyping. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/289423.289448

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

Ross Tate, Juan Chen, and Chris Hawblitzel. 2008. *A Flexible Framework for Type Inference with Existential Quantification*. Technical Report MSR-TR-2008-184. https://www.microsoft.com/en-us/research/publication/a-flexible-framework-for-type-inference-with-existential-quantification/

Ross Tate, Juan Chen, and Chris Hawblitzel. 2010. Inferable object-oriented typed assembly language. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1806596.1806644

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

Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. 2005. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*. https://homepages.inf.ed.ac.uk/wadler/fool/program/final/14/14_Paper.pdf

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)*. https://doi.org/10.1145/967900.968162

Valery Trifonov and Scott Smith. 1996. Subtyping constrained types. In *Static Analysis Symposium (SAS)*. https://doi.org/10.1007/3-540-61739-6_52

Stefan Wehr and Peter Thiemann. 2009. On the Decidability of Subtyping with Bounded Existential Types. In *Programming Languages and Systems (ESOP)*. https://doi.org/10.1007/978-3-642-10672-9_10

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 (2018). https://doi.org/10.1145/3276483

Received 2023-11-16; accepted 2024-03-31