Decidable Subtyping for the Julia Language

Thesis Proposal

Julia Belyakova

Northeastern University
Khoury College of Computer Sciences
Boston, Massachusetts, USA
April 2023

Abstract

Julia is a dynamic, high-level, yet high-performance programming language for scientific computing. To encourage a high level of code reuse and extensibility, Julia is designed around symmetric multiple dynamic dispatch, which allows functions to have multiple implementations tailored to different argument types. To resolve multiple dispatch, Julia relies on a subtype relation over a complex language of run-time types and type annotations, which include parametric types, untagged unions that distribute over covariant tuples, and impredicative existential types. Notably, Julia's subtyping is undecidable. Unlike in statically typed languages with undecidable subtyping, e.g. Scala, where the undecidability manifests at compile time, in Julia, it leads to a run-time stack overflow error. This can happen at almost any point during the program execution, as subtyping is used to resolve every function call. In this thesis, I propose to develop a decidable subtype relation for the Julia language, such that migrating existing code would require minimal effort. In particular, to provide for decidability, I will restrict the language of type annotations. To evaluate the migration effort, I will check whether type annotations used in registered Julia packages conform to the proposed restriction.

CONTENTS

1

INTRODUCTION

Julia is a dynamic, high-level, yet high-performance programming language [Bezanson et al. 2017], originally designed for scientific computing. Julia aims to solve the two-language problem, combining the convenience of productivity languages such as Python and R, and performance of languages such as Fortran and C [Bezanson et al. 2018].

Notably, to encourage a high level of code reuse and extensibility, Julia is designed around *symmetric multiple dynamic dispatch*. Multiple dispatch allows a function to have multiple implementations, called methods, tailored to different argument types; at run time, a function call is dispatched to the most specific method for the given arguments. To deliver performance, Julia relies on a type-specializing just-in-time (JIT) compiler: with few exceptionsevery time a method is called with a new set of argument types, it is specialized for those types, creating a new method definition [Pelenitsyn et al. 2021]. Therefore, even for functions with a single user-defined method, more methods can accrue during the program execution, adding work for the dispatch mechanism.

Multiple dispatch is resolved with the use of a subtype relation. Namely, for every function call, subtyping is checked between the argument types and method type signatures, as well as between method type signatures. While most run-time value types are nominal, the language of type annotations is inspired by semantic subtyping and supports untagged unions that distribute over covariant tuples. In addition, Julia uses existential types to represent parametric method definitions, choosing an unusual combination of type language features.

Despite intentional simplifications in the type language, e.g. the lack of recursive bounds on type variables, subtyping in Julia is *undecidable* due to impredicative bounded existential types. Simply disallowing such types would not be practical, as Julia users rely on them to describe heterogeneous data. For example, `Vector{Vector{T} where T<:Number}` describes a vector of vectors of numeric values where inner vectors can have different element types. A more detailed account of Julia types and subtyping is provided in Chapter 3.

Although relying on undecidable subtyping is not unprecedented for a statically typed language, e.g. Scala [Hu and Lhoták 2019], the price of undecidability is higher in Julia, where it can manifest at almost any point during the program execution. Namely, the run-time system relies on undecidable subtyping to resolve function calls, process new method definitions, manipulate

Table 1: Statistics of issues on the Julia bug tracker: open/closed (March 2023)

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

data (e.g. when adding an element to a container), as well as during the JIT compilation. In practice, the undecidability leads to a run-time crash with a `StackOverflowError`. Such an issue can be particularly hard to debug, because neither the problematic subtyping query nor its origins are available to the user.

A number of issues related to subtyping have been reported on the Julia bug tracker. For example, `#41948`[1] reports a `StackOverflowError` caused by a function definition, which is likely linked to the undecidability; `#33137`[2] points out an inconsistency in subtyping; `#24166`[3] (now fixed) reports a problem with reflexivity and transitivity. Overall, there are 105 open/704 closed issues labeled with "types and dispatch" as of March 2023, with 13 open/138 closed being also labeled with "bug" (not every issue is properly labeled as a bug, e.g. the aforementioned `#24166`). Tab. 1 provides a few more data points for comparison: for example, there are 8 open/86 closed "codegen" bugs and 1 open/15 closed "GC" bugs. Thus, type-related concerns, including the undecidability of subtyping, are not purely theoretical and constitute a non-negligible portion of problems in the Julia implementation.

1 https://github.com/JuliaLang/julia/issues/41948
2 https://github.com/JuliaLang/julia/issues/33137
3 https://github.com/JuliaLang/julia/issues/24166

2

THESIS STATEMENT

Because of the undecidability of subtyping, which is an integral part of Julia's dynamic semantics, a Julia program can unexpectedly crash during the execution. One way to address this problem would be to use a decidable subtype relation in place of the existing, undecidable one. However, as subtyping impacts the set of valid programs and their semantics, this can have profound effects on user experience with the language.

My thesis is:

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

To validate the thesis, I will:

1. design a new subtype relation based on the one currently used by Julia and prove it decidable;
2. estimate the migration effort by conducting a static analysis of registered Julia packages and suggesting code migration strategies.

An implementation of the subtype relation and understanding its impact on Julia's performance are beyond the scope of this thesis and remain future work.

PRELIMINARY WORK. In my thesis research so far, I have collaborated on reconstructing a specification of Julia subtyping (OOPSLA 2018 [Zappa Nardelli et al. 2018]), and defined and mechanized a set-theoretic model of a subset of Julia types (FTfJP 2019 [Belyakova 2019]); more details about these efforts can be found in Chapter 4. I have also collaborated on modeling Julia's dynamic semantics, including `eval` [Belyakova et al. 2020] and the JIT compiler [Pelenitsyn et al. 2021], which illuminated the role of types and subtyping in the language.

3

BACKGROUND: DISPATCH AND SUBTYPING IN JULIA

3.1 OVERVIEW OF THE JULIA LANGUAGE

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

Multiple dynamic dispatch [Bobrow et al. 1986; Chambers 1992] is at the core of the Julia language. The dispatch mechanism allows a function, called *generic function*, to have multiple implementations, called *methods*, that are tailored to different argument types. For example, Fig. 1 shows several method definitions of function `(-)`, which is used as both unary negation (lines 1 and 5) and binary subtraction (lines 2–4). At *run time*[1], every function call in the program is dispatched to the *most specific applicable* method, which is determined based on the types of the argument values. If such a single best method does not exist, an error is raised. Section 3.2 describes this process in more detail, but for now, it suffices to know that dispatch resolution relies on **subtyping**. Notably, Julia does not allow the user to call a method of a generic function directly: the only way to reach a method is via the dispatch mechanism. Thus, whenever a method is called, its arguments are guaranteed to be subtypes of the declared method signature types.

Julia has a non-trivial **language of types**, as demonstrated in Fig. 1. In addition to nominal types such as `BigInt` and `Number`, the language also supports set-theoretic union types (e.g. `Union{Int16, ..., UInt128}` in line 3), covariant tuple types (e.g. `Tuple{String, Number}`), invariant parametric types (e.g. `Vector{T}`), and so-called union-all types, written with `where` (lines 3 and 5). Intuitively, a union-all type `t where T` represents a union of types `t[t'/T]` for all valid instantiations `t'` of the type variable `T`. Zappa Nardelli et al. [2018] provide a detailed

1 Statically typed languages often support *method overloading*. The difference between method overloading and method dispatch is that overloading is resolved at compile time, using static types of the arguments, whereas dispatch is resolved at run time, using the precise, run-time types of the arguments.

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

Figure 1: Several method definitions of generic function `(-)` in Julia

account of Julia types and their subtype relation, and we give a brief overview of subtyping in Section 3.3.

One of the key properties related to Julia types is the distinction between *concrete* and *abstract* types. A concrete type is a type that describes the representation of a value; for any value, its concrete type can be obtained with the `typeof` operator. Concrete types include primitive types, such as `Int128`, and composite `struct` types (either mutable or immutable). Any concrete type definition in Julia is final: a concrete type does not have non-empty subtypes other than itself. On the other hand, every concrete type has a single declared supertype, which can only be an abstract nominal type. Abstract nominal types can be used to create a nominal subtype hierarchy, but they cannot be instantiated with values. Other categories of abstract types include union types such as `Union{Int16, ..., UInt128}`, and union-all types such as `Vector{T} where T` (a shorthand for this type is simply `Vector`). Note, however, that every particular instantiation of `Vector`, e.g. `Vector{Number}`, is a concrete type. The bottom type represented with the empty union, `Union{}`, is neither concrete nor abstract: it has no values and is a subtype of all types.

Fig. 2 provides several examples of user-defined concrete (on the left) and abstract (on the right) types. Parametric types (both abstract and concrete) can declare (non-recursive) lower and upper bounds on type variables; for example, type `Rational{T}` requires `T` to be a subtype of abstract `Integer`. Supertypes in type definitions are declared to the right of `<:`. For example, `Int128` is a subtype of `Signed`, and `Signed` is a subtype of `Integer`. If the supertype declaration is omitted, like in `AbstractSet{T}`, the supertype defaults to `Any`, which is a supertype of all types.

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

```
primitive type Int128 <: Signed 128
end

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

mutable struct
  BitSet <: AbstractSet{Int}
  ...
end
```

```
abstract type Signed <: Integer
end

abstract type AbstractSet{T}
end
```

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

3.2 MULTIPLE DISPATCH RESOLUTION

Method resolution for a dispatched function call `f(a1, a2, ...)` relies on tuple subtyping [Leavens and Millstein 1998] between run-time argument types and method type signatures. Namely, argument types are combined into a tuple of concrete types

$$\sigma = \texttt{Tuple\{typeof(a1), typeof(a2), ...\}},$$

whereas every method type signature is either a tuple of declared argument types if the method definition is not parametric, or a union-all of that tuple if the definition is parametric. For example, consider Fig. 1: method in line 1 is represented by the signature `Tuple{BigInt}`, line 4 by `Tuple{Missing,Number}`, and line 3 by `Tuple{T, T} where T <: Union{...}`.

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

1. Find all applicable methods, i.e., all method signatures τ of the function `f` that are supertypes of the argument type $\sigma <: \tau$. If no methods are applicable, a method-not-found error is raised.

2. Among the applicable methods $\{\tau_1, \ldots, \tau_n\}$, pick the method with the most specific type signature, i.e., τ_k such that $\forall i \in 1..n, \tau_k <: \tau_i$. If there is no such single best method, a method-is-ambiguous error is raised.[2]

Notably, Julia's dispatch mechanism is *symmetric*: during dispatch resolution, all arguments are given equal consideration as a part of subtyping on tuples.

2 Requested by language users over the years, Julia has additional specificity rules to resolve ambiguities in some cases, but those are not relevant to this work.

While there are ways to optimize dispatch resolution to limit the number of subtyping checks, the key takeaway is that in Julia, *subtyping* is checked *at run time* as a part of resolving multiple dynamic dispatch.

3.3 JULIA SUBTYPING

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

User-defined parametric structs are invariant in the type parameter regardless of whether the struct is mutable or immutable, meaning that `Name{t11, t12, ...}` is a subtype of `Name{t21, t22, ...}` only if the corresponding type arguments are equivalent, i.e., `t11 <:> t21, t21 <:> t22, ...`. Thus, the immutable invariant struct `Rational{Int}` is *not* a subtype of `Rational{Signed}`, while the covariant tuple `Tuple{Int} <: Tuple{Signed}`.

Abstract union-all types `t where tl<:T<:tu` are better known in literature as **bounded existential types**, which also model Java wildcards [Torgersen et al. 2004][3]. In what follows, we will call `t where tl<:T<:tu` an existential type; if lower (upper) bound on the type variable is omitted, it defaults to the bottom type `Union{}` (top type `Any`). Intuitively, an existential type denotes a union of `t[t'/T]` for all instantiations of the type variable `T` such that `tl <: t' <: tu`. Similarly to subtyping of finite union types, the intent is that:

- `(t where tl<:T<:tu) <: t2` if *for all* valid instantiations `t'`, it holds that `t[t'/T] <: t2`, and
- `t1 <: (t where tl<:T<:tu)` if *there exists* at least one `t'` such that `t1 <: t[t'/T]`.

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

Existential types in Julia are *impredicative*: existential quantifiers can appear anywhere in a type, and type variables can be instantiated with arbitrary

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

existential types. For example, type `Vector{Matrix{T} where T}` denotes a vector of matrices with arbitrary element types. In contrast, `Vector{Matrix{S}} where S` denotes a set of vectors where elements are matrices with the same element type. Thus, a vector of integer matrices `Vector{Matrix{Int}}` is a subtype of the latter—existential—type, because `S` can be instantiated with `Int`. But it is not a subtype of the former—invariant parametric—type `Vector{Matrix{T} where T}`, because type arguments `Matrix{Int}` and `Matrix{T} where T` are not equivalent. Note that because of the impredicativity, type arguments of invariant type constructors such as `Vector` are arbitrarily complex. Thus, even though any particular `Vector{t}` is a concrete type, checking subtyping for this type requires a subtyping check for an arbitrary `t`.

Existential types serve at least two distinct purposes in the Julia language. First, parametric types with existential parameters, such as `Vector{Matrix{T} where T}`, are useful for representing heterogeneous data. Second, top-level existential types, such as `Tuple{T, Vector{T}} where T`, represent type signatures of parametric method definitions. It may be surprising that Julia uses existential rather than universal types, but recall that the primary purpose of types is to serve multiple dispatch. In Julia, it is impossible to directly invoke a parametric method definition and provide it with a type argument. Instead, the method is being dispatched to if subtyping for the corresponding existential type succeeds. Then, in the body of the method, the existential type is implicitly unpacked, with the witness type being some valid instantiation induced by subtyping. Consider the following code snippet as an example:

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

Because `[5, 7, 5]` is a `Vector{Int}` and `Tuple{Vector{Int}}` is a subtype of the existential `Tuple{Vector{T}} where T`, as witnessed by the instantiation of `T` with `Int`, the call `f([5, 7, 5])` dispatches to the method in line 1, and `T` in the body of the method becomes `Int`. However, when multiple instantiations of the variable are possible, Julia sometimes gives up on assigning the witness type. In the example below, subtyping succeeds for the call `g(5.0)`, because `Tuple{Bool} <: Tuple{T} where T>:Int`, as witnessed by multiple instantiations of `T`, e.g. `Any` or `Union{Int, Bool}`. But rather than pick one instantiation, Julia throws an error.

```
> g(x::T) where T>:Int = begin
    println(x)
    println(T)
  end

> g(true)
true
ERROR: UndefVarError: T not defined
```

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

```
> g(5)
5
Int
```

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

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

If it were possible to instantiate `T` with an abstract type such as `Integer`, the method could be called with a pair of a signed and unsigned integer. This would be an incorrect implementation of equality, for the same bit representation corresponds to different numbers when interpreted with and without the sign. To prevent such behavior, the diagonal rule states: if a type variable appears in the type (1) only covariantly and (2) more than once, it can be instantiated only with concrete types.[4] Thus, the type signature of `(==)` above, `Tuple{T, T} where T<:Number`, represents a restricted existential type: it is a union of tuples `Tuple{t, t}` where `t` is a concrete subtype of `Number`. The same rule applies in line 3 of Fig. 1: built-in integer subtraction `sub_int` is guaranteed to be called only with primitive integers of the same concrete type.

3.4 UNDECIDABILITY OF JULIA SUBTYPING

It is not unusual for a statically typed programming language to have undecidable subtyping, as witnessed by Java and Scala [Grigore 2017; Hu and Lhoták 2019]. In practice, the undecidability means that the compiler might

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

$$
\begin{aligned}
[\![Top]\!] &= \texttt{Union\{\}} \\
[\![\alpha]\!] &= \alpha \\
[\![\forall \alpha \leq \tau . \tau']\!] &= \texttt{Tuple\{Ref\{}\alpha\texttt{\},}[\![\tau']\!]\texttt{\}} \ \texttt{where}\ \alpha\ \texttt{>:}[\![\tau]\!]
\end{aligned}
$$

Figure 3: Encoding of $F^N_{\leq}$ types in Julia

not terminate on some programs. Although undesirable, such property can be acceptable if it manifests rarely and allows for an expressive type system.

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

Despite the intentional simplifications in the type language, Julia subtyping is in fact *undecidable*. As discovered by Ben Chung and Ross Tate, Julia can encode system $F^N_{\leq}$, which is known to be undecidable [Pierce 1992]. Fig. 3 shows the encoding[5], with $\tau_1 \leq \tau_2$ in $F^N_{\leq}$ defined as $[\![\tau_2]\!] <: [\![\tau_1]\!]$ in Julia.

In practice, the undecidability of Julia subtyping manifests itself with a `StackOverflowError`. The reason is that internally, Julia relies on a dedicated stack to resolve subtyping and terminates the program when the stack reaches a certain limit.

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

4

PRELIMINARY WORK

A RECONSTRUCTION OF JULIA SUBTYPING. This work is the result of a collaborative effort to provide a readable specification of Julia subtyping. Initially, an incomplete (and soon outdated) definition of subtyping existed only in [Bezanson 2015], with the actual implementation of subtyping (∼3000 lines of heavily optimized C code) being the only reference point.

In [Zappa Nardelli et al. 2018], we define subtyping in the form of a judgment

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

Here, E is a type variable environment that contains two kinds of variables, forall (also called left) and exist (also called right). A forall variable is added to the environment when an existential type appears on the left of the subtyping judgment, as in `(Vector{X} where X) <: t'`. Such variables never change in the environment, which corresponds to the intuition that for a left-hand side existential type, subtyping should hold for all possible instantiations of the type variable. However, when an existential type appears on the right, as in `t <: Vector{Q} where Q`, the variable is added as an exist variable. During subtype checking, such variables may accrue subtype constraints in addition to their declared bounds; if all the constraints are consistent, subtyping succeeds, which corresponds to the intuition that for a right-hand side existential type, subtyping holds if there exists a valid instantiation of the type variable. Exist-variables with updated bounds are recorded in the output environment E'. For example, if `Q` is an exist variable with bounds `Union{}<:Q<:Any`, then

$$\mathtt{Q}^{\mathtt{Any}}_{\mathtt{Union\{\}}} \vdash \mathtt{Int} <: \mathtt{Q} \vdash \mathtt{Q}^{\mathtt{Any}}_{\mathtt{Int}},$$

i.e., `Int<:Q` results in updating `Q` to `Int<:Q<:Any`.

Having reconstructed the specification of Julia subtyping, we were able to identify its undecidability. Furthermore, this work highlighted several other problems, e.g. the lack of transitivity, which I plan to address in the thesis.

TAG-BASED SEMANTIC SUBTYPING FOR NON-EXISTENTIAL TYPES. Julia's subtype relation was inspired by semantic subtyping. The description of the original Julia design [Bezanson 2015] suggested an intuitive interpretation of types as sets of values, but the interpretation is not well defined. Furthermore,

the treatment of invariant parametric types in the interpretation does not match the subtype relation. In particular, for concrete nominal `Name`,

$$[\![\texttt{Name\{}\tau\texttt{\}}]\!] = \{x \mid \texttt{typeof}(x) = \texttt{Name\{}\tau\texttt{\}}\}$$

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

In [Belyakova 2019], I proposed and mechanized in Coq a semantic interpretation of (a subset of) Julia types τ as sets of type tags σ (i.e. concrete types) rather than values. For the type language of non-parametric nominal types, tuples, and unions, a decidable syntactic subtyping based on disjunctive normal form coincides with the set inclusion of interpretations $[\![\tau]\!] \subseteq [\![\tau']\!]$.

Following up on this work, I extended the interpretation to support invariant type constructors such as `Vector{...}`. To account for the issue of multiple syntactic representations of types, in the new system, types are given an indexed interpretation $[\![\cdot]\!]_n$. In particular, a concrete type constructor `Name{...}` is interpreted as

$$\begin{array}{lcl} [\![\texttt{Name\{}\tau\texttt{\}}]\!]_{n+1} & = & \{\texttt{Name\{}\tau'\texttt{\}} \mid [\![\tau']\!]_n = [\![\tau]\!]_n\} \\ [\![\texttt{Name\{}\tau\texttt{\}}]\!]_0 & = & \{\texttt{Name\{}\tau'\texttt{\}}\}, \end{array}$$

and subtyping is defined as

$$\tau <: \tau' \quad \equiv \quad \forall n.\, [\![\tau]\!]_n \subseteq [\![\tau']\!]_n.$$

Despite the more complex indexed interpretation, the subtype relation has an equivalent, decidable syntactic definition $\tau \leq \tau'$ where $\texttt{Name\{}\tau\texttt{\}} \leq \texttt{Name\{}\tau'\texttt{\}}$ holds if $\tau \leq \tau'$ and $\tau' \leq \tau$. This extension is also mechanized in Coq.

5

RELATED WORK

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

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

5.1 DECIDABILITY OF SUBTYPING

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

Typically, undecidability is shown by a reduction from a known undecidable problem. For example, the halting problem for Turing machines is used to prove the undecidability of Java generics [Grigore 2017] as well as System $F_{<:}$ [Pierce 1992]. $F_{<:}$ [Cardelli et al. 1991] is a language with bounded universal types, where the undecidability of subtyping is caused by the combination of contravariance and rebounding of type variables in the context. Proofs by reduction from $F_{<:}$ are commonly used to establish undecidability in languages with some form of bounded quantification. For instance, the undecidability of DOT [Hu and Lhoták 2019] is shown by reducing a variant of $F_{<:}$ to $D_{<:}$, a restriction of DOT featuring type members and path-dependent types. A similar

approach is used by Mackay et al. [2019] for Wyv_{core}, another simplification of DOT featuring path-dependent and recursive types. In the case of Julia, the undecidability of subtyping also follows from its power to encode a variant of $F_{<:}$, as discussed in Section 3.4.

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

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

Identifying decidable fragments of undecidable type systems and subtype relations remains an important challenge. In some cases, it might be possible to recover decidability by restricting the system in a way that does not affect most practical programs. For example, the material-shape separation for Java generics [Greenman et al. 2014] enables decidability of subtyping by limiting F-bounded polymorphism (i.e., recursive constraints on type variables) to the subset of types, called shapes, that are used exclusively as constraints; this separation appears to be in agreement with an industry-wide practice. Mackay et al. [2019] extend the material-shape separation to the context of path-dependent and recursive types. Earlier, Kennedy and Pierce [2007] identified three decidable fragments of undecidable subtyping in the context of nominal inheritance and variance without F-bounded polymorphism: the fragments can be obtained by restricting either contravariance, or expansive class tables, or multiple instantiation inheritance. Aiming for decidable subtyping, Julia designers deliberately restricted the language to *not* support F-bounded polymorphism, contravariant constructors, and multiple inheritance. Based on those restrictions, Bezanson [2015] conjectured the decidability of subtyping but pointed out that the combination of invariant constructors and contravariance in lower bounds of existential types is akin to the source of undecidability in $F_{<:}$; the conjecture relied on the fact that left- and right-hand side variables

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

are treated assymetrically in Julia. Still, Julia subtyping is powerful enough to encode the undecidable $F_{<:}$.

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

5.2 SEMANTIC SUBTYPING

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

However, semantic subtyping still needs a decision procedure, and that is where the complexity typically comes from, especially if efficiency matters. For example, in the Julia language, semantic subtyping of union and tuple types relies on a space-efficient algorithm that lazily explores the disjunctive normal form [Chung et al. 2019]. In [Frisch et al. 2002], which supports not only union, but also intersection and negation types, subtyping $t \leq s$ is checked via the emptiness test $t \wedge \neg s \approx \bot$ and also relies on the disjunctive normal form. Related to Julia's treatment of right-hand side existential variables as unification variables, Castagna et al. [2015] generate and solve subtype constraints as a part of type inference process in their set-theoretic framework with union, intersection, and negation types. Schimpf et al. [2023] successfully apply the same framework to typing Erlang. However, the approach to constraint solving does not translate to Julia due to the presence of invariant type constructors.

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

5.3 JAVA WILDCARDS

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

Subtyping Java wildcards is challenging and, as shown by Grigore [2017], undecidable. Wehr and Thiemann [2009] identify multiple undecidable subtype relations for bounded existential types in formal models inspired by Java. Tate et al. [2011] highlight multiple sources of non-termination in Java subtyping, e.g. the presence of recursive constraints on type variables and wildcards being allowed in the inheritance hierarchy; by making practical restrictions on the Java language, they provide a terminating subtyping algorithm. Julia lacks all the Java features that were linked to non-termination of subtyping but supports union and existential types that are not present in Java's surface language.

6

PROPOSED WORK

DECIDABLE SUBTYPING. I will show that the decidability of subtyping can be achieved for a type language similar to the one currently used by Julia, with additional restrictions on existential types. The restricted grammar of types is shown in Fig. 4. The key idea is to distinguish between more expressive ψ and less expressive τ, with invariant parametric types $N\{\ldots\}$ being allowed to have only τ as type arguments. In particular:

- type signature ψ, meant to represent method type signatures, is allowed to have a more expressive, Julia-style form of existential types where bound type variables can have multiple occurrences anywhere in the signature; to encode the diagonal rule, existential variables can be declared concrete (when kind k is c), meaning that they can be instantiated only with concrete types;
- type τ, meant to represent data, is allowed to have only a limited, Java wildcards-style form of existential types where type variables occur exactly once in an invariant position and are bound immediately outside the invariant costructor; namely, $N\{\tau_l <: <: \tau_u\}$ represents an existential type $\exists \tau_l <: X <: \tau_u . N\{X\}$.

In addition to being decidable, the proposed subtype relation is intended to be reflexive and transitive, as is typical for such a relation. Furthermore, building on the indexed, tag-based interpretation of types, I will extend it to account for type variables, and explore the relationship between the extended model and the decidable, syntactic subtype relation.

EVALUATION OF MIGRATION EFFORT. To evaluate the effort required for migrating existing code to a language with the new subtype relation, I will conduct a static analysis of type annotations in the official registry of Julia packages (about 9000 packages as of March 2023). In addition, I will manually inspect a random sample of packages containing type annotations that do not conform to the new type language and propose possible migration strategies. Thus, for example, `Vector{Dict{T,T} where T}` (a vector of dictionaries where keys and values are of the same type) is not expressible under the restriction due to multiple occurrences of `T`. Such a type can be replaced

$$
\begin{array}{lcll}
\psi & ::= & \bot \mid \top \mid \psi \times \psi \mid \psi \cup \psi \mid N\{v,\ldots\} \mid \exists \tau <: \overset{k}{X} <: \tau.\psi \mid X & \text{type signature} \\
\tau & ::= & \bot \mid \top \mid \tau \times \tau \mid \tau \cup \tau \mid N\{v,\ldots\} \mid X & \text{type} \\
v & ::= & \tau <:< \tau & \text{restricted existential variable} \\
k & ::= & a \mid c & \text{variable kind (any } \mid \text{ concrete)}
\end{array}
$$

Figure 4: Restricted grammar of Julia types

with a more permissive `Vector{Dict{<:Any,<:Any}}`, which is a shorthand for `Vector{Dict{K,V} where K where V}`. Then, the equality between the types of keys and values can be checked at run time for each dictionary in the vector. As a second example, consider `Vector{Vector{Union{T, Int}} where T}`, which is not expressible due to `T` not being an immediate argument of the inner `Vector`. Under Julia's subtyping, this type is equivalent to `Vector{Vector{>:Int}}` (a shorthand for `Vector{Vector{T} where T>:Int}`), which is expressible in the new system.

PAPER SUBMISSION. I intend to submit a POPL 2024 paper on the formalization and evaluation of the decidable subtype relation.

SCHEDULE. I am currently working on proving the decidability, reflexivity, and transitivity of the new subtype relation, and intend to complete proofs in May 2023. After that, I will perform the evaluation and study an extension of the tag-based semantic interpretation for types with type variables. I will concurrently work on writing the thesis and the POPL 2024 paper throughout May and June, and will finish the thesis in July.

Table 2: Schedule

	May	June	July	August
proofs & evaluation	X	X		
paper	X	X		
thesis	X	X	X	
defense				X

BIBLIOGRAPHY

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. https://doi.org/10.1007/978-3-319-30936-1_14 (cited on p. 13)

Davide Ancona and Andrea Corradi. 2016. Semantic Subtyping for Imperative Object-Oriented Languages. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2016)*. https://doi.org/10.1145/2983990.2983992 (cited on p. 16)

Julia Belyakova. 2019. Decidable Tag-based Semantic Subtyping for Nominal Types, Tuples, and Unions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2019)*. Article 3. https://doi.org/10.1145/3340672.3341115 (cited on pp. 3 and 12)

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428275 (cited on p. 3)

Jeff Bezanson. 2015. *Abstraction in technical computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/99811 (cited on pp. 10, 11, 13, and 14)

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 120 (Oct 2018). https://doi.org/10.1145/3276490 (cited on pp. 1 and 4)

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017). https://doi.org/10.1137/141000671 (cited on pp. 1 and 4)

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-Oriented Programming. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1986)*. https://doi.org/10.1145/28697.28700 (cited on p. 4)

Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *European Conference on Object-Oriented Programming*

(ECOOP 2008). https://doi.org/10.1007/978-3-540-70592-5_2 (cited on p. 16)

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*. https://doi.org/10.1007/3-540-54415-1_73 (cited on p. 13)

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec 1985). https://doi.org/10.1145/6041.6042 (cited on p. 15)

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction. In *Symposium on Principles of Programming Languages (POPL 2015)*. https://doi.org/10.1145/2676726.2676991 (cited on p. 15)

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *International Conference on Functional Programming (ICFP 2011)*. https://doi.org/10.1145/2034773.2034788 (cited on p. 16)

Craig Chambers. 1992. Object-oriented multi-methods in Cecil. In *European Conference on Programming Languages (ECOOP 1992)*. https://doi.org/10.1007/BFb0053029 (cited on p. 4)

Benjamin Chung, Francesco Zappa Nardelli, and Jan Vitek. 2019. Julia's Efficient Algorithm for Subtyping Unions and Covariant Tuples (Pearl). In *European Conference on Programming Languages (ECOOP 2019, Vol. 134)*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.24 (cited on p. 15)

Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2016. Semantic Subtyping for Objects and Classes. *Comput. J.* 60, 5 (Dec 2016). https://doi.org/10.1093/comjnl/bxw080 (cited on p. 16)

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic subtyping. *Symposium on Logic in Computer Science* (2002). https://doi.org/10.1109/LICS.2002.1029823 (cited on pp. 15 and 16)

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sep 2008). https://doi.org/10.1145/1391289.1391293 (cited on p. 15)

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-Bounded Polymorphism into Shape. In *Conference on Programming Language Design and Implementation (PLDI 2014)*. https://doi.org/10.1145/2594291.2594308 (cited on p. 14)

Radu Grigore. 2017. Java Generics Are Turing Complete. In *Symposium on Principles of Programming Languages (POPL 2017)*. https://doi.org/10.1145/3009837.3009871 (cited on pp. 9, 13, and 16)

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2000. Regular Expression Types for XML. In *International Conference on Functional Programming (ICFP 2000)*. https://doi.org/10.1145/351240.351242 (cited on p. 15)

Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of D<: And Its Decidable Fragments. *Proc. ACM Program. Lang.* 4, POPL, Article 9 (Dec 2019). https://doi.org/10.1145/3371077 (cited on pp. 1, 9, 13, and 15)

Atsushi Igarashi and Mirko Viroli. 2002. On Variance-Based Subtyping for Parametric Types. In *European Conference on Object-Oriented Programming (ECOOP 2002)*. https://doi.org/10.1007/3-540-47993-7_19 (cited on p. 16)

Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD 2007)*. https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/ (cited on p. 14)

Kresten Krab Thorup and Mads Torgersen. 1999. Unifying Genericity. In *European Conference on Object-Oriented Programming (ECOOP 1999)*. https://doi.org/10.1007/3-540-48743-3_9 (cited on p. 16)

Gary T. Leavens and Todd D. Millstein. 1998. Multiple Dispatch as Dispatch on Tuples. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1998)*. https://doi.org/10.1145/286936.286977 (cited on p. 6)

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2019. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.* 4, POPL 2020, Article 66 (Dec 2019). https://doi.org/10.1145/3371134 (cited on pp. 14 and 15)

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems (APLAS 2020)*. https://doi.org/10.1007/978-3-030-64437-6_7 (cited on p. 15)

Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 150 (Oct 2021). https://doi.org/10.1145/3485527 (cited on pp. 1 and 3)

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Symposium on Principles of Programming Languages (POPL 1992)*. https://doi.org/10.1145/143165.143228 (cited on pp. 10 and 13)

Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-theoretic Types for Erlang. https://doi.org/10.48550/arXiv.2302.12783 arXiv:2302.12783 [cs.PL] (cited on p. 15)

Daniel Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It?. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2008)*. https://doi.org/10.1145/1449764.1449804 (cited on p. 16)

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Conference on Programming Language Design and Implementation (PLDI 2011)*. https://doi.org/10.1145/1993498.1993570 (cited on pp. 10 and 16)

Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. 2005. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL 2005)*. https://homepages.inf.ed.ac.uk/wadler/fool/program/final/14/14_Paper.pdf (cited on p. 16)

Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. 2004. Adding Wildcards to the Java Programming Language. In *Symposium on Applied Computing (SAC 2004)*. https://doi.org/10.1145/967900.968162 (cited on pp. 7 and 16)

Stefan Wehr and Peter Thiemann. 2009. On the Decidability of Subtyping with Bounded Existential Types. In *Programming Languages and Systems*. https://doi.org/10.1007/978-3-642-10672-9_10 (cited on p. 16)

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (Oct 2018). https://doi.org/10.1145/3276483 (cited on pp. 3, 4, and 11)