A TYPE SYSTEM FOR JULIA

BENJAMIN CHUNG

May 2023 – version 1.2

ABSTRACT

The Julia programming language was designed to fill the needs of scientific computing by combining the benefits of productivity and performance languages. Julia allows users to write untyped scripts easily without needing to worry about many implementation details, as do other productivity languages. If one just wants to get the work done–regardless of how efficient or general the program might be—such a paradigm is ideal. Simultaneously, Julia also allows library developers to write efficient generic code that can run as fast as implementations in performance languages such as C or Fortran. This combination of user-facing ease and library developer-facing performance has proven quite attractive, and the language has increasing adoption.

With adoption comes combinatorial challenges to correctness. One of Julia's calling cards is multiple dispatch, a mechanism for solving the expression problem. On one hand, multiple dispatch allows many libraries to compose "out of the box:" for example, you can automatically differentiate an integrator merely by importing and using the appropriate packages. On the other hand, it creates bugs where one library expects features that another does not provide. Typing is one solution to this problem—mechanically ensuring that methods are used correctly—but would make Julia harder to use as a scripting language.

I developed a "best of both worlds" solution: gradual typing for Julia. My system forms the core of a gradual type system for Julia, laying the foundation for improving the correctness of Julia programs while not getting in the way of script writers. My framework allows methods to be individually typed or untyped, allowing users to write untyped code that interacts with typed library code and vice versa. Typed methods then get a soundness guarantee that is robust in the presence of both dynamically typed code and dynamically generated definitions. I additionally describe protocols, a mechanism for typing abstraction over concrete implementation that accommodates one common pattern in Julia libraries, and describe its implementation into my typed Julia framework.

ACKNOWLEDGMENTS

First, I want to thank my parents. I cannot sufficiently express my thanks for their love and support.

My colleagues Artem Pelenitsyn, Julia Belyakova, Francesco Zappa Nardelli, Celeste Hollenbeck, and Lionel Zoubritzky all played a vital part in the work that went into this thesis; this would not have been possible without you.

Leif Andersen, Ming-Ho Yee, Aviral Goel, and Andrew Cobb were an essential part of my degree. They helped me through both the most challenging and best parts of my tenure as a student and were always there for me when I needed support.

Jonathan Aldrich and Joshua Sunshine were key to me pursuing programming languages and research generally; they helped the undergraduate me discover a love of research and taught me how to communicate in an academic setting.

Several people both in and around the Julia team played a major part in this project. The whole collaboration is rooted in a chance encounter with Jean Yang, who I met flying back from SPLASH 2015 in Pittsburgh (after I already moved to Boston!) who was the introduction of my group to the Julia team. The Julia development team were very welcoming; in particular, I would like to thank Jeff Bezanson and Jameson Nash for advice, support, and making measured performance values obsolete extraordinarily quickly.

Last but not least, I want to thank my advisor Jan Vitek for his guidance, advice, support, and willingness to bear with my many, many side projects.

CONTENTS

1 INTRODUCTION 1
1.1 Thesis 8
2 THE JULIA LANUGAGE 10
2.1 Related work 13
2.2 Julia in action 15
2.3 Evaluating relative performance 18
2.4 The Julia programming language 20
2.4.1 Values, types, and annotations 20
2.4.2 Multiple dispatch 23
2.4.3 Metaprogramming 26
2.4.4 Discussion 33
2.5 Implementing Julia 35
2.5.1 Method specialization 36
2.5.2 Type inference 37
2.5.3 Method inlining 39
2.5.4 Object unboxing 39
2.6 Julia in Practice 39
2.6.1 Typeful programming 41
2.6.2 Multiple dispatch 42
2.6.3 Specializations 44
2.6.4 Impact on performance 45
3 SUBTYPING IN JULIA 47
3.1 Related Work 47
3.2 Formalization of Subtyping 48
3.3 Proof of Undecidability 53
3.3.1 System $F^{P}_{\leqslant}$ 53
3.3.2 Reduction from System $F^{P}_{\leqslant}$ to Julia subtyping 54
3.4 Undecidability in Practice 58
4 GRADUAL TYPING FOR JULIA 62
4.1 Related Work 73
4.1.1 Gradual Typing 73

4.1.2 Eval 76
4.1.3 Static Typing of Multiple Dispatch 77
4.2 A Core Calculus for Julia 79
4.2.1 Syntax 81
4.2.2 Semantics 81
4.3 Static Type System 85
4.3.1 Static Dispatch Resolution 87
4.3.2 Protocols 90
4.4 Typed Dynamic Semantics 93
5 TYPED JULIA IN PRACTICE 104
5.1 Implementation 104
5.1.1 Syntactic analysis 105
5.1.2 Semantic analysis 105
5.1.3 Scope analysis 107
5.1.4 Type analysis 109
5.2 Evaluation 111
5.2.1 Picking a method table 112
5.2.2 Protocols 115
5.2.3 Case Study 118
6 CONCLUSION 124
6.1 Future Work 125

1 INTRODUCTION

Computer programming languages provide abstractions for expressing our computational intent. Historically, languages have come in one of two flavors. First are productivity languages, designed to enable rapid and interactive changes to the system at the price of performance, facilitating the development of software systems. Second, performance languages are designed to allow users to write efficient code at the cost of higher development effort. Increasing performance often requires taking more control over how exactly a computation is performed which might otherwise be abstracted away. Of course, there are many dimensions to modern computer languages and the distinction between the two flavors can be blurred.

This thesis focuses on a feature that often distinguishes the two language flavors: static type checking. A statically type-checkable language can be augmented with a tool, a type checker, that catches potential errors before the program is run. Furthermore, a program that has been checked can often be more efficient as the compiler is able to rely on type information when making optimization decisions. Productivity languages frequently lack a type checker, while performance languages are much more wont to possess one. Why?

Most producitivity languages are untyped for two reasons. First, these languages often include features that are inherently difficult to check before execution of the program. Second, enforcing a static type system renders programs more rigid, inhibiting the very exploratory programming that productivity language users want.

I extend a productivity language with a static type system, but do so in a non-invasive way. In my system, statically typed code should coexist with untyped code allowing programmers to pick whether to write code that will be checked or to ignore the type system when it gets in the way. The idea is that developers of large systems, like libraries, can use static checking to provide some measure of confidence while users who just want to get their script working might not. Choice then allows the benefits of both productivity and performance languages within the same framework.

A number of criteria should hold in order to reap the benefits of static checking. First, there should be a clear and well-understood guarantee about which errors may not occur in the checked parts of the code. Additionally, no (or very few) performance regressions should occur due to typing. Finally, it should be possible to opt-in or opt-out of typing so as to provide the requisite agency to the developer.

JULIA. I have chosen the Julia language [10] as my concrete target language. Julia is a relatively young system with a combination of productivity and performance features. It was developed at MIT in the last decade by a small academic team, and has since become increasingly popular within scientific computing. Today, Julia is used in applications ranging from climactic modeling to numerical optimization, with a steadily growing user base. There are flies in the adoption and scale ointment, however.

Julia libraries have grown to be of considerable size, exceeding hundreds of thousands of lines in some cases. The developers and users of these libraries are frustrated by simple type bugs, unreliability, and confusing error messages. One selling point of Julia is that it allows easy composition of large libraries; however, when one composes libraries, one also composes their bugs. Composing libraries causes the potential for error to expand combinatorially; each inter-library interaction begets the opportunity for some new and exciting edge case to spring up. Julia, presently, has no formalized mechanism for checking or even describing these interactions. Consequently, it has gained some notoriety for being buggy due to one of its signature features[1].

Types are the standard answer for building abstraction over implementation. Abstract types, in particular, allow specification of some generic behavior that should hold for any potential instantiation of the type. Typing most untyped languages is difficult, though, for a notion of type must be added on top of the existing language concept. Julia is different.

In spite of being an untyped productivity language, Julia programs are full of types. Almost all methods in the most popular Julia libraries have at least on type annotated argument and the overwhelming majority are fully typed [11]. Moreover, most struct fields carry a type annotation. If Julia is untyped, why should programmers write all of these types?

Julia programmers write types because Julia's runtime uses them. Julia's implementation relies on types for two key applications: First, type annotated field writes are checked. These checks are performed each time the field is updated during pro-

1 https://yuri.is/not-julia/, https://github.com/JuliaStats/Distances.jl/issues/201

gram execution. Stored values can therefore be relied upon to be type-correct. In turn, this correctness guarantee lets compiled code safely read from typed fields. Second, types are used for dispatch, wherein Julia decides what implementation should be invoked from a given call site. Julia uses a mechanism called multiple dispatch to decide what method to invoke. Each Julia function can have multiple implementations, or methods, with each method having differently type annotated arguments. Julia will dynamically dispatch function invocations to the method with the most applicable type annotations. Julia therefore dynamically guarantees that arguments will be of their declared types.

These guarantees mean little, however, when they are not exposed to the programmer. As the scale of the Julia ecosystem increases, so does the likelihood that some argument, somewhere, is misused. It does not matter if the argument is a member of the statically-declared type annotation if the method body calls a function on it that does not exist. Thus, there may be cases where no applicable method exists for some call site. Only when code is exercised are these errors discovered, but not all code is exercised with all possible types during testing. Moreover, Julia's design allows users to introduce new types and thread them through existing code, further creating edge cases whereby invocations can go wrong. As a result, bugs in Julia code can hide from developers, hidden under a thin layer of passing tests and faulty unstated assumptions, only to jump out when a user is least expecting it. A static type system can help identify these issues ahead of time, improving reliability of Julia code and easing development.

COMPOSITION. One of Julia's selling points is the composition of libraries using multiple dispatch. Suppose we wanted to differentiate the function `f(x)=x*sin(x)+x*cos(x)` about `5`. We could differentiate the function's implementation on paper, but if `f` was even slightly complex this becomes impractical. We could also use finite differencing, but this is imprecise. In Julia, however, computing the derivative is as simple as:

```
> using ForwardDiff
> ForwardDiff.derivative(f, 5)
5.537670211431911
```

The library works by replacing the argument of `f` with a dual number. The dual number extends every operation on numbers to operate on the stored derivative value.

Consider the operation `x+y`, an invocation of the function `+` on the variables `x` and `y`. To evaluate this simple addition, Julia must select which method to invoke out of over 190 options:

```
# 190 methods for generic function "+":
[1] +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128,
    UInt16, UInt32, UInt64, UInt8} in Base
[2] +(c::Union{UInt16, UInt32, UInt8}, x::BigInt) in Base.GM
[3] +(c::Union{Int16, Int32, Int8}, x::BigInt) in Base.GMP
[4] +(c::Union{UInt16, UInt32, UInt8}, x::BigFloat) in Base.MPFR
[5] +(c::Union{Int16, Int32, Int8}, x::BigFloat) in Base.MPFR
...
```

The selection is based on two criteria. First, the dispatch algorithm selects methods that are *applicable*, in other word, methods whose argument type annotations include the types of the actual values provided. Second, the dispatch algorithm must, out of all applicable methods, find whichever one is the most *specific*, the one that most accurately describes the provided arguments. All arguments participate in dispatch; no preference is given to any single argument.

As an example, consider the case where both `x` and `y` are `Int64`. The following methods are all *applicable*:

```
[1] +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128,
    UInt16, UInt32, UInt64, UInt8} in Base
[2] +(a::Integer, b::Integer) in Base
[3] +(y::Integer, x::Rational) in Base
[4] +(x::T, y::T) where T<:Number in Base
[5] +(x::Number, y::Number) in Base
```

The algorithm chooses the first of these implementations as when the variable `T` is bound to `Int64`, the first method describes exactly the arguments at the call site. All other implementations handle some additional value types, and are thus less specific.

As the number of libraries in the Julia ecosystem increases, errors become more frequent. Julia relies on undocumented interfaces such as `+`. Programmers expect to be able to add two number-like-things together, but there is no specification for either what a number-like-thing is, or what + should actually do.

To illustrate this, let us consider a case where the intuition for what a number-like-thing is breaks down. Suppose that instead of using forward differentiation to calculate the derivative of `f`, we wanted to do it symbolically using computer algebra. We can do this in Julia by using the `Symbolics` library as follows:

```
> using Symbolics: variable, solve_for
> x = variable(:x)
> eqn = f(x)
x*cos(x) + x*sin(x)
> Symbolics.derivative(eqn, x)
x*cos(x) + cos(x) + sin(x) - x*sin(x)
```

Now, instead of passing a dual number to `f` for automatic differentiation, we pass the symbolic variable `x`. This symbolic variable accumulates the operations performed on it, constructing an equation describing the computation. We can then symbolically differentiate this equation to determine the derivative of the function as a whole. This can readily go wrong on a function like `g`:

```
function g(x)
   result = 0
   for i=1:x
     result += i
   end
   return result
end
```

Symbolically differentiating `g` with respect to `x` yields the error:

```
ERROR: TypeError: non-boolean (Num) used in boolean context
Stacktrace:
 [1] unitrange_last(start::Num, stop::Num)
   @ Base .\range.jl:294
 [2] UnitRange{Num}(start::Num, stop::Num)
   @ Base .\range.jl:287
 [3] Colon
   @ .\range.jl:5 [inlined]
 [4] Colon
   @ .\range.jl:3 [inlined]
 [5] f(x::Num)
   @ Main .\REPL[19]:3
```

This is an example of where the lack of agreement on what it means to be a number-like-thing bites us. Here, should all number-like-things be usable as iteration bounds? The function `g` implicitly assumes that yes, numbers should be usable as an iteration bound. Integers satisfy this unstated assumption but abstract variables such as those introduced by Symbolics cannot. Moreover, there is no way to specify that `g` takes arguments that can serve as loop bounds. This fundamental lack of agreement on what it should be possible to do to a number can create unexpected errors from deep within programs that are difficult to understand; why, in this example, is the error referring to using a non-boolean as a boolean even though the real issue is that we are trying to bound iteration with an algebraic variable? Deep inspection of the library's source code can answer this question, but most programmers would prefer not to have to do this to identify simple errors of this nature.

Julia possesses no means of specifying or enforcing the existence or usage of abstractions. Programmers introduce generic notions (such as "this should act like `+`") but then have no way to ensure that every implementation of the notion is complete nor that usages are correct. As a result, this problem of composition running wild only grows with the size of the Julia ecosystem.

A large number of Julia packages have been written for describing implementations of various abstractions. None have gotten substantial adoption. I argue that this failure is in part because none of them actually solve the problem: they allow users to easily describe abstractions, but do not guarantee that implementations are correct to any standard nor that usages are safe. As a result, the benefit of using such a system is very limited.

The problem underlying these efforts to canonize some abstraction is that checking correctness of Julia code against any given standard is hard. Julia itself provides no mechanisms for source-level static analysis; the only tools on offer are for an intermediate language with little direct correspondence to source code. The only available source code analyzers are so deeply integrated into one particular code editor (as part of the Julia Visual Studio Code extension) and limited that they are not practical for this application.

I aim to address this gap by providing a framework for sound static type checking in Julia. I built a system that can type check Julia code that fits within the existing Julia type paradigms and can be extended to support various sorts of abstractions.

Moreover, I propose one particular approach to abstraction (which I refer to as protocols, more on this later). However, static typing is not a great fit for all use cases of Julia.

GRADUAL TYPING. A type system answers the question of "how can we ensure that independently-developed code can interoperate?" If both sides type checked their code and the type system is sound then composition should then be straightforward. The problem is that, in practice, not all code is typed.

As I mentioned earlier, Julia straddles the line between performance and productivity language. Some Julia developers such as scientists and analysts want to use it as a productivity language, geting results fast without worrying too much about if their code is correct. Other developers like library programmers want to have every assurance possible that their code will work in the "real world." Where the analyst might be annoyed by type errors the library developer might love the red squiggly line. Julia accommodates both use cases—and a type system for Julia should as well.

A type system that can meet the needs of both the productivity and performance developer is one that lets both typed and untyped code coexist. This mixing of soundly typed and untyped code is the popular concept of gradual typing. Gradual type systems allow soundly statically typed code to run in the same system as untyped code.

One concern is that a key Julia feature is performance. Gradually typed Julia, then, needs to be just as fast as base Julia if it is going to be useful particularly for library developers—and past gradual type systems [76] have had severe performance impacts. My type system for Julia needs to have minimal to no runtime impact.

As shown earlier, Julia code has a lot of semantically meaningful type annotations. Normally, these annotations are used for dispatch, not for static checking. The existence and semantics of type annotations then means that programmers have an intuition about what it means to inhabit a type in Julia, which is a critical issue for gradual typing. However, if I use Julia's existing type system, then I need to be able to reason about subtyping in Julia.

SUBTYPING. Julia's type language inlcudes nominal single subtyping as well as union, parametric invariant existential, and covariant tuple types. Subtyping of unions and tuples is distributive, including over parametric types, which Julia augments with the so-called diagonal rule.

All of these features cause Julia's type system to be theoretically challenging. I was able to prove that subtyping in Julia is undecidable by reduction from System $F_{<:}$ [65], showing that type checking in Julia is potentially nonterminating.

Undecidability of subtyping is not fatal for type checking. However, it illustrates that reasoning about and deriving useful properties from subtyping is difficult. The type system cannot rely on any specific definition of subtyping, instead using various approximations that aim to be sound and useful rather than complete. The type checker needs to be parametric over operations on types.

DYNAMIC CODE GENERATION. Dynamically generated code is another issue for a type system. Other untyped languages make extensive use of features like `eval` . Prior gradual type systems for such languages ignore this dynamism as it is very difficult to reason about code that does not yet exist. Moreover, dynamically checking the newly-generated code is frequently impractical.

Julia, like other dynamic lanugages, supports `eval` . Ideally, a type system for Julia should then be able to support dynamically typed code. The problem is that multiple dispatch allows functions to be extended with new methods anywhere and anywhen—including from `eval` . As a result, *no* call site is safe; any function call could have an untyped method slipped underneath it at any time. While uncommon in practice, a sound type system for Julia should be able to retain soundness even in the presence of `eval` .

1.1 THESIS

I posit that a static type system can be designed for Julia such that

- statically typed code can interoperate with untyped methods,
- static type annotations do not introduce new dynamic checks,
- dynamically generated code does not break the whole system.

My static type system will guarantee that statically typed methods do not go wrong while leaving the semantics of untyped code unchanged.

I additionally demonstrate one kind of abstraction for Julia programs: protocols. A protocol defines the interface of a function that must be implemented for all of the subtypes of the declared argument. Protocols demonstrate how one common pattern in Julia can be typed and how future work might use the type system to provide checked abstraction.

This work builds on a number of papers that I have co-written; papers in **bold** are directly used or extended in this work, while those in *italics* serve as background.

- *Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation*
  OOPSLA 2021
- **World Age in Julia: Optimizing Method Dispatch in the Presence of Eval**
  OOPSLA 2020
- *Julia's Efficient Algorithm for Subtyping Unions and Covariant Tuples*
  ECOOP 2019
- **Julia Subtyping: a Rational Reconstruction**
  OOPSLA 2018
- **Julia: Dynamism and Performance Reconciled by Design**
  OOPSLA 2018
- *KafKa: Gradual Typing for Objects*
  ECOOP 2018

2 THE JULIA LANUGAGE

Before I can examine too deeply how I should type Julia I first need to consider what Julia *is*, exactly. Superficially, of course, Julia is a programming language, but what was it designed to do, what are its distinguishing features, and how does the community take advantage of them? The answers to each of these questions are critical to answering how a type system should work.

The purpose of Julia is straightforward. Julia aims to bridge the gap between productivity and performance languages. Previously, a scientist might have written the interface to their library in Python and the backend in C++. In contrast, Julia aims to allow programmers to write both the easy-to-use interface and the high-performance implementation in a single homogeneous Julia codebase.

The fact that Julia delivers on its promise of having both performance and ease of use is surprising. Dynamic languages like Python or R typically suffer from at least an order of magnitude slowdown over C and often more. Fig. 1 illustrates that Julia is indeed a dynamic language. Just as in in Python, one can declare a `Node` datatype containing two untyped fields, `val` and `nxt`, and an untyped `insert` function that takes a sorted list and performs an ordered insertion. While this code will be optimized by the Julia compiler, it is not going to run fast without some additional programmer intervention.

The key to performance in Julia lies in the synergy between language design, implementation techniques and programming style. Julia's design was carefully tailored so that a very small team of language implementers could create an efficient compiler. The key to this relative ease is to leverage the combination of language features and programming idioms to reduce overhead, but what language properties enable easy compilation to fast code?

Language design: Julia includes a number of features that are common to many productivity languages, namely dynamic types, optional type annotations, reflection, dynamic code loading, and garbage collection. A slightly less common feature

This work was previously published in OOPSLA 2018 [11] as Julia: Dynamism and performance reconciled by design. This section is an updated version of that presentation.

```
mutable struct Node
    val
    nxt
end

function insert(list, elem)
    if list isa Void
        return Node(elem, nothing)
    elseif list.val > elem
        return Node(elem, list)
    end
    list.nxt = insert(list.nxt, elem)
    list
end
```

Figure 1: Linked list

is symmetric multiple dispatch [13]. In Julia a function can have multiple implementations, called methods, distinguished by the type annotations added to parameters of the function. At run-time, a function call is dispatched to the most specific method applicable to the types of the arguments. Type annotations can be attached to datatype declarations as well, in which case they are checked whenever typed fields are assigned to. The language design does impose limits on some of those features, for instance the `eval` function does not run in local scope, but instead is evaluated at the top-level. Another significant choice for optimizations is the difference between concrete and abstract types: the former can have fields and can be instantiated while the latter can be extended by subtypes.

Language implementation: Performance in Julia does not arise from great feats of compiler engineering: Julia's implementation is simpler than that of many dynamic languages. The Julia compiler has three main optimizations that are performed on a high-level intermediate representation; native code generation is then delegated to the LLVM compiler infrastructure. The optimizations performed in Julia are (1) *method inlining* which devirtualizes multi-dispatched calls and inline the call target; (2) *object unboxing* to avoid heap allocation; and (3) *method specialization* where code is special-cased to its actual argument types. The compiler does not support the kind of speculative compilation and deoptimizations common in dynamic language implementations, but supports dynamic code loading from the interpreter and with `eval()`.

The synergy between language design and implementation is in evidence in the interaction between the three optimizations. Each call to a function that has, as arguments, a combination of concrete types not observed before triggers specialization. A data-flow analysis algorithm uses the type of the arguments (and if these are user-defined types, the declared type of their fields) to approximate the types of all variables in the specialized function. This enables both unboxing and inlining. The specialized method is added to the function's dispatch table so that future calls with the same combination of argument types can reuse the generated code.

Programming style: To assist the implementation, Julia programmers need to write idiomatic code that can be compiled effectively. Programmers are keenly aware of the optimizations that the compiler performs and shape their code accordingly. For instance, adding type annotations to fields of datatypes is viewed as good practice as it provides information to the compiler to estimate the size of instances and may allow unboxing. Another good practice is to write methods that are *type stable*. A method is type stable if, when it is specialized to a set of concrete types, data-flow analysis can assign concrete types to all variables in the function. This property should hold for all specializations of the same method. Type instability can stem from methods that can return values of different types, from assignment of different types to the same variable depending on branches of the function, or from functions that cannot devirtualized and analyzed.

Julia's design for (easy implementation of) performance is critical to understanding how the language itself and code written in Julia might be typed. I will provide an overview of this design and how it facilitates performance by synergizing the design with the compilation pipeline. This close coupling provides good performance (between 0.9x and 6.1x of optimized C code on a small suite of 10 benchmarks) and makes many of the choices in type checking obvious. Towards this last point, I will consider a corpus of 50 popular Julia projects on GitHub to examine how Julia's features and their underlying design choices are exercised by the broader community and discuss how they lend themselves towards type-ability.

2.1 RELATED WORK

SCIENTIFIC COMPUTING LANGUAGES. R [68] and MATLAB [56] are the two languages superficially closest to Julia. Both languages are dynamically typed, garbage collected, vectorized and offer an integrated development environment focused on a read-eval-print loop. However, the languages' attitudes towards vectorization differ. In R and MATLAB, vectorized functions are more efficient than iterative code whereas the contrary stands for Julia. In this context I use "vectorization" to refer to code that operates on entire vectors[1], so for instance in R, all operations are implicitly vectorized. The reason vectorized operations are faster in R and MATLAB is that the implicit loop they denote is written in a C library, while source-level loops are interpreted and slow. In comparison, Julia can compile loops very efficiently, as long as type information is present.

While there has been much research in compilation of R [45, 82, 1] and MATLAB [22, 27], both languages are far from matching the performance of Julia. The main difference, in terms of performance, between MATLAB or R, and Julia comes from language design decisions. MATLAB and R are more dynamic than Julia, allowing, for example, reflective operations to inspect and modify the current scope and arbitrary redefinition of functions. Other issues include the lack of type annotations on data declarations, crucial for unboxing in Julia.

Other languages have targeted the scientific computing space, most notably IBM's X10 [20] and Oracle's Fortress [74]. The two languages are both statically typed, but differ in their details. X10 focuses on programming for multicore machines that have partitioned global addressed spaces; its type system is designed to track the locations of values. Fortress, on the other hand, had multiple dispatch like Julia, but never reached a stage where its performance could be evaluated due to the complexity of its type system. In comparison, Julia's multi-threading is still in its infancy, and it does not have any support for partitioned address spaces.

MULTIPLE DISPATCH. Multiple dispatch goes back to [13] and is used in languages such as CLOS [28], Perl [69] and R [19]. Lifting explicit programmatic type tests into dispatch requires an expressive annotation sublanguage to capture the same logic; expressiveness that has created substantial research challenges. Researchers have struggled with how to provide expressiveness while ensuring

1 This discussion should not be confused with hardware-level vectorization, e.g. SIMD operations, which are available to Julia at the LLVM level.

type soundness. Languages such as Cecil [49] and Fortress [4] are notable for their rich type systems; but, as mentioned in Guy Steele's retrospective talk, finding an efficient, expressive and sound type system remains an open challenge.[2] The language design trade-off seems to be that programmers want to express relations between arguments that require complex types, but when types are rich enough, static type checking becomes difficult. The Fortress designers were not able to prove soundness, and the project ended before they could get external validation of their design. Julia side-steps many of the problems encountered in previous work on typed programming languages with multiple dispatch. It makes no attempt to statically ensure invocation soundness or prevent ambiguities, falling back to dynamic errors in these cases.

STATIC TYPE INFERENCE. At heart, despite the allure of types and the optimizations they allow, type inference for untyped programs is difficult. Flow typing tries to propagate types through the program at large, but sacrifices soundness in the process. Soft typing [31] applies Hindley-Milner type inference to untyped programs, enabling optimizations. This approach has been applied practically in Chez Scheme [81]. However, Hindley-Milner type inference is too slow to use on practically large code bases. Moreover, many language features (such as subtyping) are incompatible with it. Constraint propagation or dataflow type inference systems are a commonly used alternative to Hindley-Milner inference. These systems work by propagating types in a data flow analysis [3]. No unification is needed, and it is therefore much faster and more flexible than soft typing. Several inference systems based on data flow have been proposed for JavaScript [21], Scheme [71], and others.

DYNAMIC TYPE INFERENCE FOR JIT OPTIMIZATIONS. Feeding dynamic type information into a type propagation type inference system is not a technique new to Julia. The first system to use dataflow type inference inside a JIT compiler was RATA [53]. RATA relies on abstract interpretation of dynamically-discovered intervals, kinds, and variations to infer extremely precise types for JavaScript code; types which enable JIT optimizations. The same approach was then used by Hackett [41], which used a simplified type propagation system to infer types for more general JavaScript code, providing performance improvements. In comparison to dynamic type inference systems for JavaScript, Julia's richer type annotations and multiple

2 JuliaCon 2016, https://www.youtube.com/watch?v=EZD3Scuv02g.

dispatch allow it to infer more precise types. Another related project is the StaDyn [34] language. StaDyn was designed specifically with hybrid static and dynamic type inference in mind. However, StaDyn does not have many of Julia's features that enable precise type inference, including typed fields and multiple dispatch.

DYNAMIC LANGUAGE IMPLEMENTATION. Modern dynamic language implementation techniques can be traced back to the work on the Self language, that pioneered the ideas of run-time specialization and deoptimization [43]. These ideas were then transferred into the Java HotSpot compiler [63]; in HotSpot, static type information can be used to determine out object layout, and deoptimization is used when inlining decisions were invalidated by newly loaded code. Implementations of JavaScript have increased the degree of specialization, for instance allowing unboxed primitive arrays at the more complex guards and potentially wide-ranging deoptimization [82].

2.2 JULIA IN ACTION

To introduce Julia, let's consider an example function. This code started as an attempt to replicate the R language's multi-dimensional summary function. For explanatory reason, I shortened the code somewhat, the shortened version simply computes the sum of a vector. Just like the R function that inspired it, the Julia code is polymorphic over vectors of integer, float, boolean, and complex values. Furthermore, since R supports missing values in every data type, I encode `NA` s in Julia.[3]

3 Since Julia v1.0 support for missing values is native, this example shows how it could be encoded.

```
function vsum(x)
    sum = zero(x)
    for i = 1:length(x)
        @inbounds v = x[i]
        if !is_na(v)
            sum += v
        end
    end
    sum
end
```

Figure 2: Compute vector sum

```
    push    %rbp
    mov     %rsp, %rbp
    mov     (%rdi), %rcx
    mov     8(%rdi), %rdx
    xor     %eax, %eax
    test    %rdx, %rdx
    cmove   %rax, %rdx
    movl    $1, %esi
    movabs  $0x8000000000000000, %r8
    jmp     L54
    nopw    %cs:(%rax,%rax)
L48:add     %rdi, %rax
    inc     %rsi
L54:dec     %rsi
    nopl    (%rax)
L64:cmp     %rsi, %rdx
    je      L83
    mov     (%rcx,%rsi,8), %rdi
    inc     %rsi
    cmp     %r8, %rdi
    je      L64
    jmp     L48
L83:pop     %rbp
    ret
    nopw    %cs:(%rax,%rax)
```

Figure 3: `@code_native vsum([1])` (X86-64)

```
zero(::Array{T}) where {T<:AbstractFloat} =
    0.0
zero(::Array{T}) where {T<:Complex} =
    complex(0.0,0.0)
zero(x) = 0

is_na(x::T) where T = x == typemin(T)

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

Figure 4: `zero` yields the zero matching the element type, by default the integer `0`. `is_na` checks for missing values encoded as the smallest element of a type (returned by the builtin function `typemin`). `typemin` is extended with a method to return the smallest complex value

```
primitive type RBool 8 end

RBool(x::UInt8) = reinterpret(RBool, x)
convert(::Type{T},x::RBool) where{T<:Real}
    = T(reinterpret(UInt8,x))

const T  = RBool(0x1)
const F  = RBool(0x0)
const NA = RBool(0xff)

typemin(::Type{RBool}) = NA

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

Figure 5: `RBool` is a an 8-bit primitive type representing boolean values extended with a missing value `NA`. The constructor takes an 8-bit unsigned integer. Conversion casts any number into an `RBool` by `reinterpret` ing the in-memory representation as a `RBool`. A new method is added to `typemin` to return `NA`

Fig. 2 shows how to sum values of a vector `x` of any type. As the Julia syntax is straightforward, little explanation is required to understand the programmer's intent. In this case, type annotations are not needed for the compiler to optimize the code, so I omit them. Variables are lexically scoped; an initial assignment defines them. Fig. 3 is the output of `@code_native(vsum([1]))` (a call to the function with a vector of integers). It shows the x86 machine code generated for the specialized method. It is noteworthy that the generated machine code does not contain object allocation or method invocation, nor does it invoke any language runtime components. The machine code is similar to code one would expect to be emitted by a C compiler.

Type stability is key to performant Julia code, allowing the compiler to optimize using types. An expression is type stable if, in a given type context, it always returns a value of the same type. Function `vsum(x)` always returns a value that is either of the same type as the element type of `x` (for floating point and complex vectors) or `Int64`. For the call `vsum([1])`, the method returns an `Int64`, as its argument is of type `Array{Int64,1}`. When presented with such a call, the Julia compiler specializes the method for that type. Specialization provides enough information to determine that all values manipulated by the computation are of the same type, `Int64`. Thus, no boxing is required; moreover, all calls are devirtualized and inlined. The `@inbounds` macro elides array bounds checking.

Type stability may require cooperation from the developer. Consider variable `sum`: its type has to match the element type of `x`. In our case, `sum` must be appropriately initialized to support any of the possible argument types integer, float, complex or boolean. To ensure type stability, the programmer leverages dispatch and specialization with the definition of the function `zero` shown in Fig. 4. It dispatches on the type of its argument. If the argument is an array containing subtypes of float, the function returns float `0.0`. Similarly, if passed an array containing complex numbers, the function returns a complex zero. In all other cases, it returns integer `0`. All three methods are trivially type stable, as they always return the same value for the same types.

Missing values also require attention. Each primitive type needs its own representation—yet the code for checking whether a value is missing must remain type stable. This can be achieved by leveraging dispatch. I add a function `is_na(x)` that returns true if `x` is missing. I select the smallest value in each type to use as its missing value (obtained by calling the builtin function `typemin`).

The solution outlined so far fails for booleans, as their minimum is `false`, which I can't steal. Fig. 5 shows how to add a new boolean data type, `RBool`. Like Julia's boolean, `RBool` is represented as an 8-bit value; but like R's boolean, it has three values, true, false and missing. Defining a new data type entails providing a constructor and a conversion function. Since our data type has only three useful values, we enumerate them as constants. I then add a method to `typemin` to return `NA`. Finally, since the loop adds booleans to integers, I need to extend addition to integer and `RBool`, this is done by interpreting true as 1 and false as zero.

2.3 EVALUATING RELATIVE PERFORMANCE

Figure 6: Time spent on implementations

Julia has to be fast to compete against other languages used for scientific computing, but it also has to be easy to develop and maintain. Programming languages are notoriously expensive propositions in terms of the level of expertise required during development and the effort required to achieve production-quality outcomes. Fig. 6 shows a very rough estimate of the the person-years invested in several language implementations. These blunt approximations were obtained using commit histories: two commits made by the same developer in one week were counted as one person-week of effort. While approximate, this figure suggests that performance comes at a substantial cost in engineering. For example, V8 for JavaScript and HotSpot for Java, two high-performance implementations, have nearly two person-centuries invested into their respective implementations. Even PyPy, an academic project, has over one century of work by our metric. Given the difference in implementation effort, the fact that Julia's performance is competitive is surprising.

To estimate the languages' relative performance, I selected 10 small programs for which implementations in C, JavaScript, and Python are available in the programming language benchmark game (PLBG) suite [38]. The suite consists of small but non-trivial benchmarks which stress either computational or memory performance. I started with PLBG programs written by the Julia developers and fixed some performance anomalies. The benchmarks are written in an idiomatic style, using the same algorithms as the C benchmarks. Their code is largely untyped, with type annotations only appearing on structure fields. Over the 10 benchmark programs, 12 type annotations appear, all on structs and only in the nbody, binary_trees, and knucleotide. The `@inbounds` macro eliding bounds checking is the only low-level optimization used, leveraged only in revcomp. Using the PLBG methodology, I measured the size of the programs by removing comments and duplicate whitespace characters, then performing the minimal GZip compression. The combined size of all the benchmarks is 6 KB for Julia, 7.4 KB for JavaScript, 8.3 KB for Python and 14.2 KB for C.

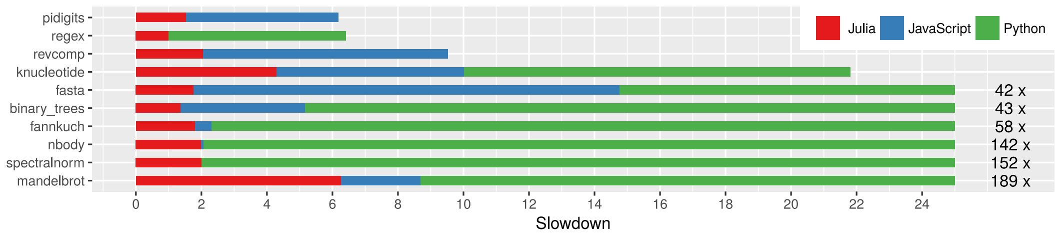


Figure 7: Slowdown of Julia, JavaScript, and Python relative to C

Fig. 7 compares the performance of the four languages with the results normalized to the running time of the C programs. Measurements were obtained using Julia v0.6.2, CPython 3.5.3, V8/Node.js v8.11.1, and GCC 6.3.0 -O2 for C, running on Debian 9.4 on a Intel i7-950 at 3.07GHz with 10GB of RAM. All benchmarks ran single threaded. No other optimization flags were used.

The results show Julia consistently outperforming Python and JavaScript (with the exception of spectralnorm). Julia is mostly within 2x of C. Slowdowns are likely due to memory operations. Like other high level dynamically-typed programming languages, Julia relies on a garbage collector to manage memory. It prohibits the kind of explicit memory management tricks that C allows. In particular, it allocates structs on the heap. Stack allocation is only used in limited circumstances. Moreover, Julia disallows pointer arithmetic.

Three programs fall outside of this range: two programs (knucleotide and mandelbrot) have slowdowns greater than 2x over C, while one (regex) is faster than C. The knucleotide benchmark was written for clarity over performance; it makes heavy use of abstractly-typed struct fields (which cause the values they denote to be boxed). In the case of mandelbrot, the C code is manually vectorized to compute the fractal image 8 pixels at a time; Julia's implementation, however, computes one pixel at a time. Finally, regex, which was within the margin of error of C, simply calls into the same regex library C does.

Julia is fast on tiny benchmarks, but this may not be representative of real-world programs. I lack the benchmarks to gauge Julia's performance at scale. Some libraries have published comparisons. JuMP, a large embedded domain specific language for mathematical optimization, is one such library. JuMP converts numerous problem types (e.g. linear, integer linear, convex, and nonlinear) into standard form for solvers. When compared to equivalent implementations in C++, MATLAB, and Python, JuMP is within 2x of C++. For comparison, MATLAB libraries are between 4x and 18x slower than C++, while Python's optimization frameworks are at least 70x slower than C++ [54]. This provides some evidence that Julia's performance on small benchmarks may carry over to larger programs.

2.4 THE JULIA PROGRAMMING LANGUAGE

The designers of Julia set out to develop a language specifically for the needs of scientific computation, and they chose a finely tuned set of features to support this use case. Antecedent languages, like R and MATLAB, illustrate scientific programmers' desire to write high-level scripts, which motivated Julia's adoption of an optionally typed surface language. Likewise, these languages drove home the importance of flexibility: programmers regularly extend core language functionalities to fit their needs. Julia provides this extensibility mechanism through multiple dispatch.

2.4.1 *Values, types, and annotations*

2.4.1.1 *Values*

Values can be either instances of *primitive types*, represented as sequences of bits, or *composite types*, represented as a collection of fields holding values. Logically, every value is tagged by its full type description; in practice, however, tags are often elided when they can be inferred from context. Composite types are immutable by default, thus assignment to their fields is not allowed. This restriction is lifted when the `mutable` keyword is used.

2.4.1.2 *Types declarations*

Programmers can declare three kinds of types: *abstract types*, *primitive types*, and *composite types*. Types can be parametrized by bounded type variables and have a single supertype. The type `Any` is the root of the type hierarchy, or the greatest supertype (top). *Abstract* types cannot be instantiated; *concrete* types can.

```
abstract type Number end

abstract type Real <: Number end

primitive type Int64 <: Signed 64 end

struct Polar{T<:Real} <: Number
    r::T
    t::T
end
```

The code shown is an extract of Julia's numeric tower. `Number` is an abstract type with no declared supertype, which means `Any` is its supertype. `Real` is also abstract but has `Number` as its supertype. `Int64` is a primitive type with `Signed` as its supertype; it is represented in 64 bits. The struct `Polar{T<:Real}` is a subtype of `Number` with two fields of type `T` bounded by `Real`. Run-time checks ensure that values stored in these fields are of the declared type. When types are omitted from field declarations, fields can hold values of `Any` type. Julia does not make a distinction between reference and value types as Java does. Concrete types can be manipulated either by value or by reference; the choice is left to the implementation. Abstract types, however, are always manipulated by reference. It is noteworthy that composite types do not admit subtypes; therefore, types such as `Polar` are final and cannot be extended with additional fields.

2.4.1.3 *Type annotations*

Julia offers a rich type annotation language to express constraints on fields, parameters, local variables, and method return types [84]. The `::` operator ascribes a type to a definition. The annotation language includes union types, written `Union{A, ...}`; tuple types, written `Tuple{A, ...}`; iterated union types, written `TExp where A<:T<:B`; and singleton types, written `Type{T}` or `Val{V}`. The distinguished type `Union{}`, with no argument, has no value and acts as the bottom type.

Union types are abstract types which include, as values, all instances of their arguments. Thus, `Union{Integer, String}` denotes the set of all integers and strings. Tuple types describe the types of the elements that may be instantiated within a given tuple, along with their order. They are parametrized, immutable types. Additionally, they are *covariant* in their parameters. The last parameter of a tuple type may optionally be the special type `Vararg`, which denotes any number of trailing elements.

Julia provides iterated union types to allow quantification over a range of possible instantiations. For example, the denotation of a polar coordinate represented using a subtype `T` of real numbers is `Polar{T} where Union{}<:T<:Real`. Each **`where`** clause introduces a single type variable. The type application syntax `A{B}` requires `A` to be a **`where`** type, and substitutes `B` for the outermost type variable in `A`. Type variable bounds can refer to outer type variables. For example,

```
Tuple{T, S} where S<:AbstractArray{T} where T<:Real
```

refers to 2-tuples whose first element is some `Real`, and whose second element is an array whose element type is the type of the first tuple element, `T`.

A singleton type is a special kind of abstract type, `Type{T}`, whose only instance is the object `T`.

2.4.1.4 *Subtyping*

In Julia, the subtyping relation between types, written `<:`, is used in run-time casts, as well as method dispatch. Semantic subtyping partially influenced Julia's subtyping [33], but practical considerations caused Julia to evolve in a unique direction. Julia has an original combination of *nominal subtyping*, *union types*, *iterated union types*, *covariant* and *invariant* constructors, and *singleton types*, as well as the *diagonal rule*. Parametric types are invariant in their parameters because this allows the Julia compiler to perform optimizations dependent on the memory representation of values. Arrays of dissimilar values box each of their arguments, for consistent element size, under type `Array{Any}`. However, if all the values are statically determined to be of the same kind, they are stored inside of the array itself. Tuple types represent both tuples of values and function arguments. They are covariant as this allows Julia to compute dispatch using subtyping of tuples. Subtyping of union types is asymmetrical but intuitive. Whenever a union type appears on the left-hand side of a judgment, as in `Union{T1, ...} <: T`, all the types `T1, ...` must be subtypes of `T`. In contrast, if a union type appears on the right-hand side, as in `T <: Union{T1, ...}`, then only one type, `Ti`, needs to be a supertype of `T`. Covariant tuples are distributive with respect to unions, so `Tuple{Union{A,B}, C} <: Union{Tuple{A,C}, Tuple{B,C}}`. The iterated union construct `TExp where A<:T<:B`, as with union types, must have either a "forall" or an "exist" semantics, according to whether the union appears on the left or right of a subtyping judgment. Finally, the *diagonal rule* states that if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. For example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`, where `T` ranges over all concrete types. The details of subtyping are intricate and the interactions between features can be surprising, described in the paper [84].

2.4.1.5 *Dynamically-checked type assertions*

Type annotations in method arguments are guaranteed by the language semantics. A method executes only if all of its arguments have types that match their declarations. However, Julia allows type annotations elsewhere in the program, these act as checked type assertions. For example, to guarantee that variable `x` has type `Int64`, the assertion `x::Int64` can be inserted into its declaration. Likewise, functions can assert a return type, as in `f()::Int = ...` for example. Fields and expressions can also be annotated. These annotations check the type of the expression's or field's value. If that type is not a subtype of the declared type, Julia will try to convert it to the declared type. If this conversion fails, an exception is thrown.

2.4.2 *Multiple dispatch*

Julia uses multiple dispatch extensively, allowing extension of functionality by means of overloading. Each function (for example `+`) can consist of an arbitrarily large number of methods (in the case of `+`, 180). Each of these methods declares what types it can handle, and Julia will dispatch to whichever method is most specific for a given call. As hinted at with addition, multiple dispatch is omnipresent. Virtually every operation in Julia involves dispatch. New methods can be added to existing functions, extending them to work with new types.

2.4.2.1 *Example*

```
struct Dual{T}
    re::T
    dx::T
end

function Base.:(+)(a::Dual{T},b::
    Dual{T}) where T
    Dual{T}(a.re+b.re, a.dx+b.dx)
end
function Base.:(*)(a::Dual{T},b::
    Dual{T}) where T
    Dual{T}(a.re*b.re, a.dx*b.re+b.
        dx*a.re)
end
function Base.:(/)(a::Dual{T},b::
    Dual{T}) where T
    Dual{T}(a.re/b.re, (a.dx*b.re-a.
        re*b.dx)/(b.re*b.re))
end
```

Consider forward differentiation, a technique that allows derivatives to be calculated for arbitrary programs. It is implemented threading a value together with its derivative through a program. In many languages, the code being differentiated would have to be aware of forward differentiation as the dual numbers need new definitions of arithmetic. Multiple dispatch allows to implement a library that works for existing functions, as I can simply extend arithmetic operators. Suppose I want to compute the derivative of `f(a,b) = a*b/(b+b*a+b*b)` about `a`, with `a = 1` and `b = 3`. Forward differentiation works by introducing a concept of dual numbers as shown in the example. Dual numbers consist of a real component (the actual value being computed) and the derivative of that number (`dx`, in the example). Differentiation is then performed by implementing the chain rule for whatever operation is then performed. In the case of addition, for instance, the real and derivative components of the two dual numbers are simply added. Multiplication, on the other hand scales the derivatives of the terms by the opposing real component to determine the derivative of the final value.

I can implement forward differentiation in Julia very easily by overloading arithmetic. As seen in the example, I can simply add new definitions for the same operators that are used for all other arithmetic operations. Since we covered all of the operations used in the function `f`, I can now figure out the derivative of `f` by simply calling it with dual numbers: `f(Dual(1.0,1.0), Dual(3.0,0.0)).dx` yields `0.16`.

2.4.2.2 *Semantics*

Dispatching on a function `f` for a call with argument type `T` consists in picking a method m from all the methods of `f`. The selection filters out methods whose types are not a supertype of `T` and takes the method whose type `T'` is the most specific of the remaining ones. In contrast to single dispatch, every position in the tuples `T` and `T'` have the same role—there is no single receiver position that takes precedence. Specificity is required to disambiguate between two or more methods which are all supertypes of the argument type. It extends subtyping with extra rules, allowing comparison of dissimilar types as well. The specificity rules are defined by the implementation and lack a formal semantics. In general, `A` is more specific than `B` if `A != B` and either `A <: B` or one of a number of special cases hold:

(a) `A = R{P}` and `B = S{Q}`, and there exist values of `P` and `Q` such that `R <: S`. This allows us to conclude that `Array{U} where U` is more specific than `AbstractArray{String}`.

(b) Let `C` be the non-empty meet (approximate intersection) of `A` and `B`, and `C` is more specific than `B` and `B` is not more specific than `A`. This is used for union types: `Union{Int32, String}` is more specific than `Number` because the meet, `Int32`, is clearly more specific than `Number`.

(c) `A` and `B` are tuple types, `A` ends with a `Vararg` type and `A` would be more specific than `B` if its `Vararg` was expanded to give it the same number of elements as `B`. This tells us that `Tuple{Int32,Vararg{Int32}}` is more specific than `Tuple{Number,Int32,Int32}`.

(d) `A` and `B` have parameters and compatible structures, `A` provides a consistent assignment of non-`Any` types to replace `B`'s type variables, regardless of the diagonal rule. This means that `Tuple{Int, Number, Number}` is more specific than `Tuple{R,S,S} where {R, S<:R}`.

(e) `A` and `B` have parameters and compatible structures and `A`'s parameters are equal or more specific than `B`'s. As a consequence, `Tuple{Array{R} where R, Number}` is more specific than `Tuple{AbstractArray{String}, Number}`.

One interesting feature is dispatch on type objects and on primitive values. For example, the Base library's `nutple` function is defined as a set of methods dispatching on the value of their second argument. Thus a call to `ntuple(id, Val{2})` yields `(1,2)` where `id` is the identity function. The `@_inline_meta` macro is used to force inling.

```
ntuple(f, ::Type{Val{0}}) = (
    @_inline_meta; ())
ntuple(f, ::Type{Val{1}}) = (
    @_inline_meta; (f(1),))
ntuple(f, ::Type{Val{2}}) = (
    @_inline_meta; (f(1), f(2)
    ))
```

2.4.3 *Metaprogramming*

Julia provides various features for defining functions at compile-time and run-time and has a particular definition of visibility for these definitions.

2.4.3.1 *Macros*

Macros provide a way to generate code and reduce the need for `eval()`. A macro maps a tuple of arguments to an expression which is compiled directly. Macro arguments may include expressions, literal values, and symbols. The example on the right shows the definition of the `assert` macro which either returns `nothing` if the assertion is true or throws an exception with an optional message provided by the user. The `:(...)` syntax denotes quotation, that is the creation of an expression. Within it, values can be interpolated: `$x` will be replaced by the value of `x` in the expression. Once defined, this macro can then be used to make assertions like `@assert 1 + 1 == 2`.

```
macro assert(ex, msgs...)
    msg_body = isempty(msgs) ?
        ex : msgs[1]
    msg = string(msg_body)
    return :($ex ? nothing
        : throw(AssertionError($
            msg)))
end
```

Another form of macro available in Julia is the string macro. String macros allow static compilation of string literals. One example in the Julia standard library is regular expressions: `r".*"` defines a regular expression that matches a string of any characters and length, for example. String literal macros are implemented very much like normal macros: they are only distinguished by a `_str` suffix. As an example, the regular expression macro is defined as `macro r_str(p) Regex(p) end`. A string macro implementation is then simply passed the string literal which it can then analyze or otherwise process as part of expansion.

Macros in Julia are unhygenic: macro developers can easily bind to and introduce new external syntactic forms if they so wish. Julia implements a sort of superficial hygine, wherein macro-introduced symbols are by default analyzed and rewritten with generated unique identifiers. However, this system can be opted out of using the `esc` expression form. Macros, as a result, can introduce new forms. For instance, the JuMP library introduces the `@variable` macro which defines a new optimization variable and binds it into scope. As an example, if I wanted to introduce a variable `x` to a model with an upper bound of 2, I could do it with `@variable(model, x <= 2)`. The variable `x` will now be in scope and be inititalized with the desired constraint.

Macros have found a wide range of use cases in Julia. A few common patterns are:

- **Sugar**: Macros like `@assert` or `@debug` encapsulate some simple but extremely common and otherwise tedious operation, such as asserting that an expression is true or logging a message at debug level.
- **Semantic**: Macros such as `@inline` mark expressions with metadata to alter how they are compiled. Similarly, macros such as `@.` modify the semantics of the expression they're given. The `@.` macro, for instance, automatically vectorizes the expression it's given.
- **DSL**: As seen in JuMP, another use case of macros is to define DSLs. DSLs in Julia can sometimes reuse the existing Julia grammar (as in the case of JuMP) or deviate wholly from it (as seen in the example of the regular expression macro).

2.4.3.2 *Reflection*

Julia provides methods for run-time introspection. The names of fields may be interrogated using `fieldnames()` and their types, with `fieldtype()`. Types are themselves represented as a structure called `DataType`. The direct subtypes of any `DataType` may be listed using `subtypes()`. The internal representation of a `DataType` is important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if `T` is stored with C-compatible alignment. The builtin function `fieldoffset(T::DataType, i::Integer)` returns the offset for field `i` relative to the start of the type. The methods of any function may be listed using `methods()`. The method dispatch table may be searched for methods accepting a given type using `methodswith()`.

More powerful is the `eval()` function which takes an expression object and evaluates it in the global scope of the current module. For example `eval(:(1+2))` will take the expression `:(1+2)` and evaluate it yielding the expected result.

```
for op in (:+, :*, :&, :|)
    eval(:($op(a,b,c) = $op($op
        (a,b),c)))
end
```

When combined with an invocation to the parser, any arbitrary string can be evaluated, so for instance `eval(parse("function id(x) x end"))` adds an identity method. One important difference from languages such as JavaScript is that `eval()` does not have access to the current scope. This is crucial for optimizations as it means that local variables are protected from interference. The `eval()` function is sometimes used as part of code generation. Here for example is a generalization of some of the basic binary operators to three arguments. This generates four new methods of three arguments each.

2.4.3.3 *World Age*

World age is a critical component of Julia's design for performance. It arose out of a problem encountered with the one-shot JIT compilation strategy: what happens if the set of methods changes?

```
> x = 3.14
> f(x) = (
    eval(:(x = 0)
        ),
    x * 2,
    Main.x)

> f(42) # (0, 84,
      0)
> x     # 0
```

Figure 8: Scope of eval in Julia

```
> (defn g [] 2)
> (defn f [x]
    (eval `(defn g []
         ~x))
    (* x (g)))
> (f 42) ; 1764
> (g)    ; 42
> (f 42) ; 1764
```

Figure 9: Eval in Clojure

```
> g() = 2
> f(x) = (
    eval(:(g()=$x));
    x * g())
> f(42) # 84
> g()   # 42
> f(42) # 1764
```

Figure 10: Eval in Julia

Using `eval` Julia programs can modify the global state at any time including both simple variables but functions too. Modifications are limited to the global scope—as shown in 8, local variables are not changed by `eval`—but any global reference may be altered using `eval` at any time. As seen in the example, the local reference to `x` was unaffected but explicitly referring to the outer global `Main.x` shows the new value immediately.

In most dynamic languages global function lookup works the same way as these variable assignements do. For example, in 9 I see that if I use `eval` to define a new implementation of `g` mid-`f` then `f` "picks up" that definition of `g` immediately. This semantics gives Julia serious problems, however, for the one-shot JITting model means that `f` would have a compiled implementation that is now referring to the wrong `g`; Julia would have to either make all method invocations dynamic or implement on-stack-replacement to back out the compilation in order to support it.

World age is Julia's solution to this problem. If faced with a hard problem one can either face it head on (and implement deoptimization/on-stack replacement, in this example) or define it out of existence. Julia took the latter approach: world age concretizes what definitions running code has access to therein providing a consistent semantics for both compiled and dynamically dispatched method invocations.

The action of world age on our example is shown in figure 10. The function `f` is defined exactly as it is in Clojure and yet its results are different; instead of referring to the newly-added method `g` (that returns `42`), it uses the version defined when `f` was first called returning 2. Thus, the first result of calling `f` is 84. Only after execution returns to the top level does the new definition of `g` become visible to `f`, at which point the result matches Clojure's.

Besides returning to the top level (either explicitly or through use of `eval`, which executes its argument at top level) programmers can use the `invokelatest` function to use whatever the newest definitions are. These tools provide escape hatches for cases in which programmers do want to access newer definitions, such as when calling user-generated code, for example.

A number of patterns commonly appear in Julia packages to work with the restrictions imposed by world age when combined with `eval` . I cover a few of them here.

BOILERPLATING. The most common use of `eval` is to automatically generate code for boilerplate functions. These generated functions are typically created at the top-level so that they can be used by the rest of the program. Consider the

`DualNumbers.jl` package, which provides a common dual number representation for automatic differentiation. A dual number, which is a pair of the normal value and an "epsilon", which represents the derivative of the value, should support the same operations as any number does and mostly defers to the standard operations. For example, the `real` function, which gets the real component of a number when applied to a dual number should recurse into both the actual and epsilon value. `Eval` can generate all of the needed implementations at package load time (`@eval` is a macro that passes its argument to `eval` as an AST).

```
for op in (:real, :imag, :conj, :float, :complex)
   @eval Base.$op(z::Dual) = Dual($op(value(z)), $op(epsilon(z)))
end
```

A common sub-pattern is to generate proxies for interfaces defined by an external system. For this purpose, the `CxxWrap.jl` library uses `eval` at the top level to generate (with the aid of a helper method that generates the ASTs) proxies for arbitrary C++ libraries.

```
eval(build_function_expression(func, funcidx, julia_mod))
```

DEFENSIVE CALLBACKS. The most widely used pattern for `invokelatest` deals with function values of unknown age. For example, when invoking a callback provided by a client, a library may protect itself against the case where the provided function was defined after the library was loaded. There are two forms of this pattern. The simplest uses `invokelatest` for all callbacks, such as the library `Symata.jl`:

```
for hook in preexecute_hooks
  invokelatest(hook)
end
```

Every hook in `preexecute_hooks` is protected against world-age errors (at the cost of slower function calls). To avoid this slowdown, the second common pattern catches world-age exceptions and falls back to `invokelatest` such as in from the `Genie.jl` web server:

```
fr::String = try
  f()::String
catch
  Base.invokelatest(f)::String
end
```

This may cause surprises, however. If a sufficiently old method exists, the call may succeed but invoke the wrong method.[4] This pattern may also catch unwanted exceptions and execute `f` twice, including its side-effects.

DOMAIN-SPECIFIC GENERATION As a language targeting scientific computing, Julia has a large number of packages that do various symbolic domain reasoning. Examples include symbolic math libraries, such as `Symata` and `GAP`, which have the functionality to generate executable code for symbolic expressions. `Symata` provides the following method to convert an internal expression (a `Mxpr`) into a callable function. Here, `Symata` uses a translation function `mxpr_to_expr` to convert the `Symata` `mxpr` into a Julia `Expr`, then wraps it in a function definition (written using explicit AST forms), before passing it to `eval` .

```
function Compile(a::Mxpr{:List}, body)
  aux = MtoECompile()
  jexpr = Expr(:function,
               Expr(:tuple, [mxpr_to_expr(x, aux) for x in margs(a)]...),
               mxpr_to_expr(body, aux))
  Core.eval(Main, jexpr)
end
```

BOTTLENECK Generated code is commonly used in Julia as a way to mediate between a high-level DSL and a numerical library. Compilation from the DSL to executable code can dramatically improve efficiency while still retaining a high-level representation. However, functions generated thusly cannot be called from the code

4 In Julia, higher-order functions are passed by name as generic functions, so a callback will be subject to multiple dispatch.

that generated them, since they are too new. Furthermore, this code is expected to be high-performance, so using `invokelatest` for every call is not acceptable. The bottleneck pattern overcomes these issues. The idea is to split the program into two parts: one that generates code, and another that runs it. The two parts are bridged with a single `invokelatest` call (the "bottleneck"), allowing the second part to call the generated code efficiently. The pattern is used in the `DiffEqBase` library, part of the DifferentialEquations family of libraries that provides numerical differential equation solvers.

```
if hasfield(typeof(_prob),:f) && hasfield(typeof(_prob.f),:f) &&
     typeof(_prob.f.f) <: EvalFunc
  Base.invokelatest(__solve,_prob,args...; kwargs...)
else
  __solve(_prob,args...;kwargs...)
end
```

Here, if `_prob` has a field `f`, which has another field `f`, and the type of said inner-inner `f` is an `EvalFunc` (an internally-defined wrapper around any function that was generated with `eval`), then it will invoke the `__solve` function using `invokelatest`, thus allowing `__solve` to call said method. Otherwise, it will do the invocation normally.

SUPERFLUOUS EVAL This is a rare anti-pattern, probably indicating a misunderstanding of world age by some Julia programmers. For example, `Alpine.jl` package has the following call to `eval` :

```
if isa(m.disc_var_pick, Function)
  eval(m.disc_var_pick)(m)
```

Here, `eval(m.disc_var_pick)` does nothing useful but imposes a performance overhead. Because `m.disc_var_pick` is already a function value, calling `eval` on it is similar to using `eval(42)` instead of `42` directly; this neither bypasses the world age nor even interprets an AST.

NAME-BASED DISPATCH Another anti-pattern uses `eval` to convert function names to functions. For example, `ClassImbalance.jl` package chooses a function to call, using its uninterpreted name:

```
func = (labeltype == :majority) ? :argmax : :argmin
indx = eval(func)(counts)
```

It would be more efficient to operate with function values directly, i.e. `func = ... : argmin` and then call it with `func(counts)`. Similarly, when a symbol being looked up is generated dynamically, as it is in the following example from `TextAnalysis.jl`, the use of `eval` could be avoided.

```
newscheme = uppercase(newscheme)
if !in(newscheme, available_schemes) ...
newscheme = eval(Symbol(newscheme))()
```

This pattern could be replaced with a call `getfield(TextAnalysis, Symbol(newscheme))`, where `getfield` is a special built-in function that finds a value in the environment by its name. Using `getfield` would be more efficient than `eval`.

The goal of world age was to nail down what methods are visible to any given part of the program for giving a consistent semantics to compilation. However, its utility is not limited to compilation: I can also use it to solve the key problem for a gradual type system for a language like Julia with open multiple disptach. Concequently, I will return to world age in more detail later when I discuss the design of the type system.

2.4.4 *Discussion*

The design of Julia makes a number of compromises, and I discuss some of the implications here.

```
abstract type AbsPt end
struct Pt <: AbsPt
    x::Int
    y::Int
end

abstract type AbsColPt <: AbsPt
     end
struct ColPt <: AbsColPt
    x::Int
    y::Int
    c::String
end

copy(p::Pt, dx, dy) =
   Pt(p.x+dx, p.y+dy)
copy(p::ColPt, dx, dy) =
   ColPt(p.x+dx, p.y+dy, p.c)

move(p::AbsPt, dx, dy) =
   copy(p, dx, dy)
```

OBJECT ORIENTED PROGRAMMING. Julia's design does not support the class-based object oriented programming style familiar from Java. Julia lacks the encapsulation that is the default in languages going back all the way to Smalltalk: all fields of a struct are public and can be accessed freely. Moreover, there is no way to extend a point class `Pt` with a color field as one would in Java; in Julia the user must plan ahead for extension and provide a class `AbsPt`. Each "class" in that programming style is a pair of an abstract and a concrete class. One can define methods that work on abstract classes such as the `move` method which takes any point and new coordinates. The `copy` methods are specific to each concrete "class" as they must create instances. The unfortunate side effect of the fact that abstract classes have neither fields nor methods is that there is no documentation to remind the programmer that a `copy` method is needed for `ColPt`. This has to be discovered by inspection of the code.

FUNCTIONAL PROGRAMMING. Julia supports several functional programming idioms—higher order functions, immutable-by-default values—but has no arrow types. Instead, the language ascribes incomparable nominal types to functions. Thus, many traditional typed idioms are impractical, and it is impossible to dispatch on function types. However, nominal types do allow dispatch on methods passed as arguments, enabling a different set of patterns. For example, the implementation of `reduce` delegates to a special-purpose function `reduce_empty` which, given a function and list type, determines the value corresponding to the empty list. If reducing with `+`, the natural empty reduction value is 0, for the correct 0. Capturing this, `reduce_empty` has the following definition: `reduce_empty(::typeof(+), T) = zero(T)`. In this case, `reduce_empty` dispatches the nominal `+` function type, then returns the zero element for `T`.

GRADUAL TYPING. The goal of gradual type systems is to allow dynamically typed programs to be extended with type annotations after the fact [72, 78]. Julia's type system superficially appears to fit the bill; programs can start untyped, and, step by step, end up fully decorated with type annotations. But there is a fundamental difference. In a gradually typed language, a call to a function `f(t::T)`, such as `f(x)`, will be statically checked to ensure that the variable `x`'s declared type matches the argument's type `T`. In Julia, on the other hand, a call to `f(x)` will not be checked statically; if `x` does not have type `T`, then Julia throws a runtime error.

Another difference is that, in Julia, a variable, parameter, or field annotated with type `T` will *always* hold a value of type `T`. Gradual type systems only guarantee that values will act like type `T`, wrapping untyped values with contracts to ensure they they are indistinguishable [79]. If a gradually-typed program manipulates a value erroneously, that error will be flagged and blame will be assigned to the part of the program that failed to respect the declared types. Similarly, Julia departs from optional type systems, like Hack [30] or Typescript [60]. These optional type systems provide no guarantee whatsoever about what values a variable of type `T` actually holds.

Julia is closest in spirit to Thorn [12]. The two languages share a nominal subtype system with tag checks on field assignment and method calls. In both systems, a variable of type `T` will only ever have values of type `T`. However, Julia differs substantially from Thorn, as it lacks a static type system and adds multiple dispatch.

2.5 IMPLEMENTING JULIA

Julia is engineered to generate efficient native code at run-time. The Julia compiler is an optimizing just-in-time compiler structured in three phases: source code is first parsed into abstract syntax trees; those trees are then lowered into an intermediate representation that is used for Julia level optimizations; once those optimizations are complete, the code is translated into LLVM IR and machine code is generated by LLVM [47]. Fig. 11 is a high level overview of the compiler pipeline.

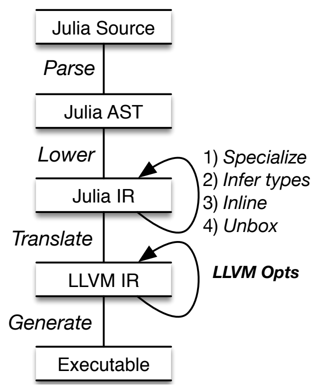


Figure 11: Julia JIT compiler

With the exception of the standard library which is pre-compiled, all Julia code executed by a running program is compiled on demand. The compiler is relatively simple: it is a method-based JIT without compilation tiers; once methods are compiled they are not changed as Julia does not support deoptimization with on-stack replacement.

Memory is managed by a stop-the-world, non-moving, mark-and-sweep garbage collector. The mark phase can be executed in parallel. The collector has a single old generation for objects that survive a number of cycles. It uses a shadow stack to record pointers in order to be precise.

Since v0.5, Julia natively supports multi-threading but the feature is still labeled as "experimental". Parallel loops use the `Threads.@threads` macro which annotates **`for`** loops that are to run in a multi-threaded region. Other part of the multi-threaded API are still in flux. An alternative to Julia native threading is the ParallelAccelerator system of [6] which generates OpenMP code on the fly for parallel kernels. The system crucially depends on type stability—code that is not type stable will execute single threaded. Fig. 12 gives an overview of the implementation of Julia v0.6.2. The standard library, Core, Base and a few other modules, accounts for most of the use of Julia in Julia's implementation. The middle-end is written in C and Julia; C++ is used for LLVM IR code generation. Finally, Scheme and Lisp are used for the front end. External dependencies such as LLVM, which is used as back end, do not participate to this figure.

Figure 12: Source files

Language	files	code
Julia	296	115,252
C	79	44,930
C++	21	18,491
Scheme	17	8,270
C/C++ Header	44	6,205
make	7	684
Bourne Shell	2	85
Assembly	4	74
	470	193,991

2.5.1 *Method specialization*

Julia's compilation strategy is built on runtime type information. Every time a method is called with a new tuple of argument types, it is specialized to these types. Optimizing methods at invocation time, rather than ahead of time, provides the JIT with key pieces of information: the memory layout of all arguments is known, allowing for unboxing and direct field access. Specialization, in turn, allows for devirtualization. Devirtualization replaces method dispatch with direct calls to a specialized method. This reduces dispatch overhead and enables inlining. As the compilation process is rather slow, results are cached, thus methods are only compiled the first time they are called with a new type. This process converges as long as functions are only called with a limited number of types. If a function gets called with many different argument types, then invocations will repeatedly incur the cost of specialization. Julia cannot avoid this pathology, as programs that generate a large number of call signatures are easy to write. To alleviate this problem, Julia allows tuple types to contain a `Vararg` component, which is treated as having type `Any`. Likewise, each function value has its own type, but Julia only specializes on function types if the argument is called in the method body. Other heuristics are used for type `Type`. Julia has one recourse against type unstable code, programmers can use the `@nospecialize` annotation to prevent specialization on a specific argument.

2.5.2 *Type inference*

Type information enables many of Julia's key optimizations. The compiler performs a data-flow analysis to discover types after specializing. Julia uses a set constraint-based analysis with constraints arising from return values, method dereferences, and argument types. Type requirements need to be satisfied at function call sites and field assignments. The system propagates constraints forward to satisfy requirements, inferring the types for intermediate values along the way.

Given the concrete types of all function arguments, intraprocedural type inference propagates types forward into the method body. An example is shown in Fig. 13. When f is called with a pair of integers, type inference finds that a+b returns an integer; therefore c is likewise an integer. From this, it follows that d is a float and so is the return type of the method. Note that this explanation relies on knowing the return type of +. Since addition could be overloaded, it is necessary to be able to infer the return types of arbitrary methods. Return types may vary depending on argument type, and previous inference results may not cover the current case. Therefore, when a new function is called, analysis of the caller must be suspended and continue on the callee to figure out the return type of the call.

```
function f(a,b)
    c = a+b
    d = c/2.0
    return d
end
```

```
function f(a::Int,b::Int)
    c = a+b::Int
    d = c/2.0::Float64
    return d
end => Float64
```

Figure 13: A simple example of type inference

Interprocedural analysis is simple for non-recursive methods as seen in Fig. 14: analysis proceeds with the called method and the return type is computed. For recursive methods cycle elimination is performed. Once a cycle is identified, it is executed until it reaches convergence. The cycle is then contracted into a single monolithic function from the perspective of analysis. More challenging are methods whose argument or return types can grow indefinitely depending on its arguments. To avoid this, Julia limits the size of the inferred types to an arbitrary bound. In this manner, the set of possible types is finite and therefore termination of the analysis is guaranteed.

```
function a()
    return b(3)+1
end
function b(num)
    return num+2
end
```

```
function a()
    return b(3)+1::Int
end => Int
function b(num::Int)
    return num+2::Int
end => Int
```

Figure 14: Simple interprocedural type inference

2.5.3 *Method inlining*

Inlining replaces a function call by the body of the called function. In Julia, it can be realized in a very efficient way because of its synergy with specialization and type inference. Indeed, if the body of a method is type stable, then the internal calls can be inlined. Conversely, inlining can help type inference because it gives additional context. For instance, inlined code can avoid branches that can be eliminated as dead code, which allows in turn to propagate more precise type information. Yet, the memory cost incurred by inlining can be sometimes prohibitive; moreover it requires additional compilation time. As a consequence, inlining is bounded by a number of pragmatic heuristics.

2.5.4 *Object unboxing*

Since Julia is dynamic, a variable may hold values of many types. As a consequence, in the general case, values are allocated on the heap with a tag that specifies their type. Unboxing allows to manipulate values directly. This optimization is helped by a combination of design choices. First, since concrete types are final, a concrete type specifies both the size of a value and its layout. This would not be the case in Java or TypeScript due to subtyping. In addition, Julia does not have a null value; if it did, there would be need for an extra tag for primitive values. As a consequence, values such as integers and floats can always be stored unboxed. Repeated boxing and unboxing can be expensive, and unboxing can also be impossible to realize although the type information is present, in particular for recursive data structures. As with inlining, heuristics are thus used to determine when to perform this optimization.

2.6 julia in practice

In order to understand how programmers use the language, I analyzed a corpus of 50 packages hosted on GitHub. I chose packages—libraries, in Julia parlance—over runnable end-user programs out of necessity: no central repository exists of Julia programs. Packages were included based on GitHub stars. Selected packages also had to pass their own test suites. Additionally, I analyzed Julia's standard library.

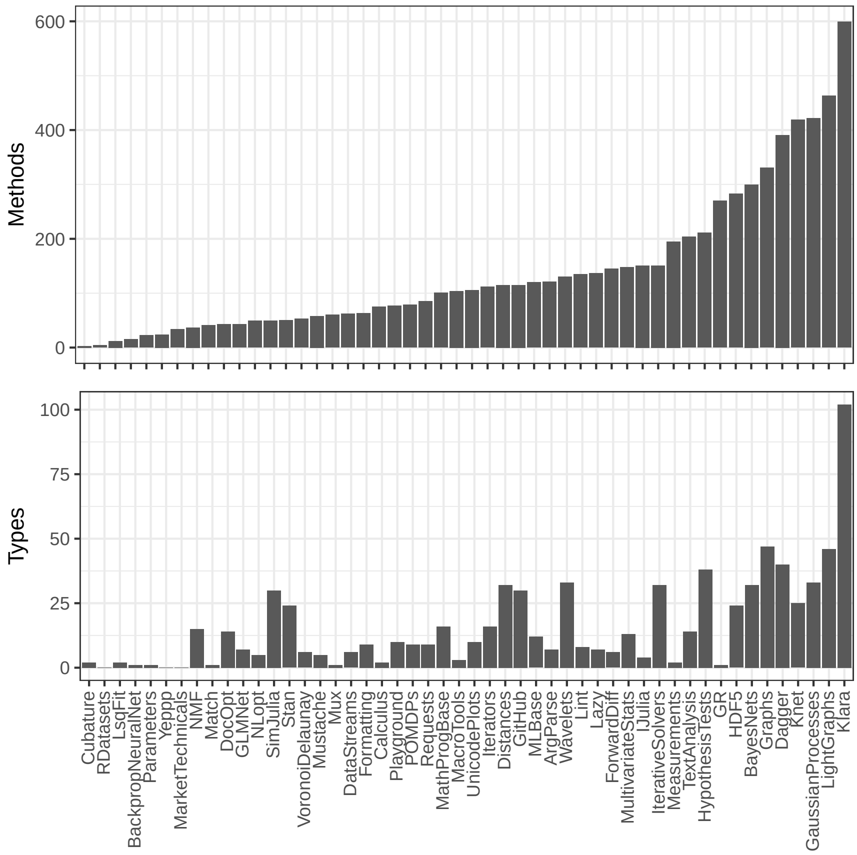
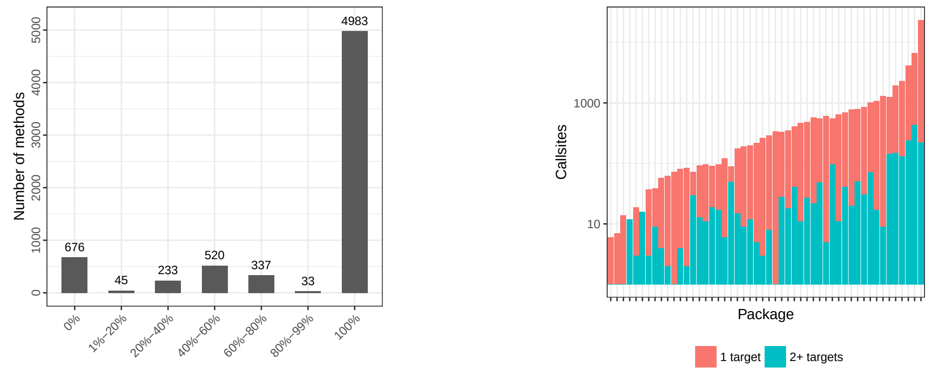


Figure 15: Number of methods and types by package



(a) Methods by percentage of typed arguments

(b) Targets per call site per package

Figure 16: High-level multimethod usage metrics

2.6.1 *Typeful programming*

Julia is a language where types are optional. Yet, knowing them is profitable since it enables major optimizations. Users are thus encouraged to program in a typeful style where code is, as much as possible, type stable. To what extent is this rule followed?

2.6.1.1 *Type annotations*

Fig. 15 gives the number of methods and types defined in each package after it was loaded into Julia, to ensure that generated methods were counted. I performed structural analysis of parsed ASTs, allowing us to measure only methods and types written by human developers. In total, the corpus includes 792 type definitions and 7,018 methods. The median number of types and methods per package is 9 and 104, respectively. Klara, a library for Markov chain Monte Carlo inference, is the largest package by both number of types and methods with 102 and 599, respectively. Three packages, MarketTechnicals, RDatasets, and Yeppp, define zero types; while Cubature defines just 3 methods, the fewest in the corpus. Clearly, Julia users define many types and functions. However, the level of dynamism remains a question. Fig. 16a shows the distribution of type annotations on arguments of method definitions. 0%

means all arguments are untyped (`Any`), while 100% means that all arguments are non-`Any`. An impressive 4,983 (or 62%) of methods are fully type-annotated. Despite having the opportunity to write untyped methods, developers define mostly typed methods.

2.6.1.2 *Type stability*

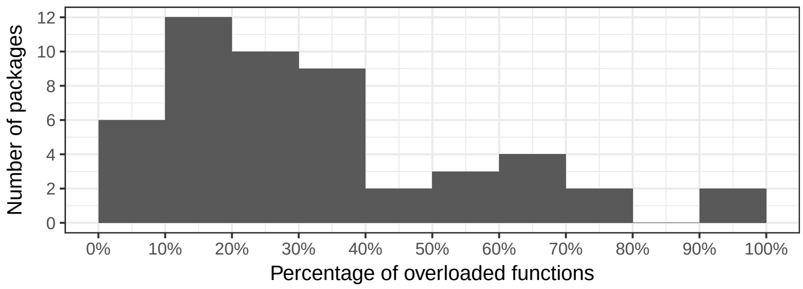
Type stability is key to devirtualizing and inlining methods. I measure type instability at run-time by dynamic analysis of the test suites of our corpus. Each called method was recorded along with the tuple of types of its arguments and the call site. I filtered calls to anonymous and compiler-generated functions to focus on functions defined by humans. Fig. 16b compares, for each package, the number of call sites where all the calls target only one specialized method to those that call two and more. Calls are recorded regardless of whether they were devirtualized. The y-axis is shown in log scale. On average, 92% of call sites target a single specialized method. Code is thus in general type stable, which agrees with the assumption that programmers attempt to write type stable code.

2.6.2 *Multiple dispatch*

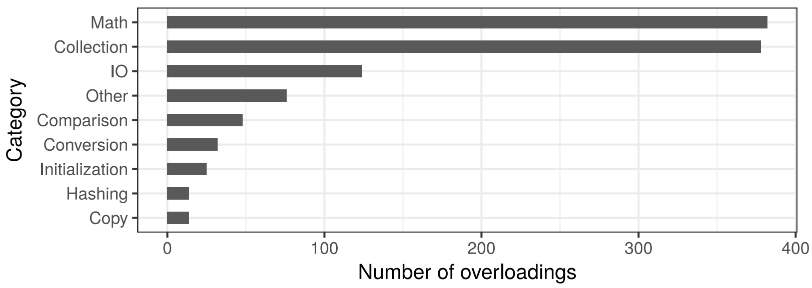
Multiple dispatch is the most prominent features of Julia's design. Its synergy with specialization is crucial to understand the performance of the language and its ability to devirtualize and inline efficiently. How is multiple dispatch used from a programmer's perspective? Moreover, a promise of multiple dispatch is that it can be used to extend existing behavior with new implementations. How much do Julia libraries extend existing functionality, and what functionality do they extend?

2.6.2.1 *Overloading*

Fig. 17a examines how multiple dispatch is used to extend existing functionality. I use the term *external overloading* to mean that a package adds a method to a function defined in a library. Packages are binned based on the percentage of functions that they overload versus define. Packages with only external overloading are at 100%, while packages that do not use external overloading would be in the 0% bin. Many packages are defined without extensive use of external overloading. For 28 out of 50



(a) Packages by % of overloaded functions



(b) Function overloads by category

Figure 17: Function overloadings

packages, fewer than 30% of the functions they define are overloads. However, the distribution of overloading has a long tail, with a few libraries relying on overloads heavily. The Measurements package has the highest proportion of overloads, with 147 overloads out of a total of 161 methods (91%). This is justified by the purpose of Measurements: it propagates errors throughout other operations, which is done by extending existing functions.

To address the question of what are overloaded, I manually categorized the top 20th quantile of overloaded functions (128 out of 641) into 9 groups. Fig. 17b depicts how many times functions from each group is overloaded. Multiple dispatch is used heavily to overload mathematical operators, like addition or trigonometric functions. Libraries overload existing operators to work with their own types, providing natural interfaces and interoperability with existing code. Examples include Calculus, which overloads arithmetic to allow symbolic expressions; and ForwardDiff, which can compute numerical derivatives of existing code using dual numbers that act just like normal values. Collection functions also are widely overloaded. Many libraries have collection-like objects, and by overloading these methods they can use their collections where Julia expects any abstract collection. However, Julia's interfaces are only defined by documentation, as a result of its dynamic design. The `AbstractArray` interface can be extended by any struct, and it is only suggested in the documentation that implementations should overload the appropriate methods. Use cases for math and collection extension are easy to come by, so their prevalence is expected. However, the lack of overloads in other categories illustrates some surprising points. For example, the large number of IO, math, and collection overloads (which implement variations on `tostring`) suggest a preponderance of new types. However, few overloads to compare, convert, or copy are provided.

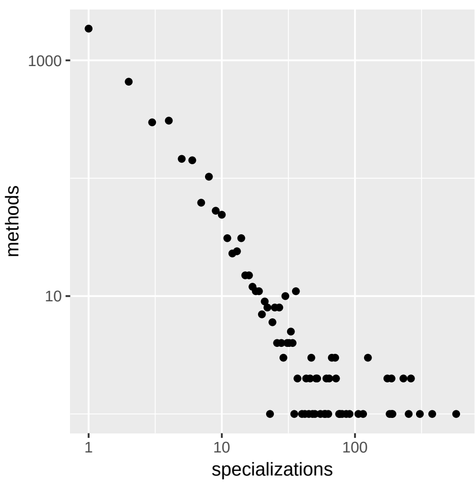


Figure 18: Number of specializations per method

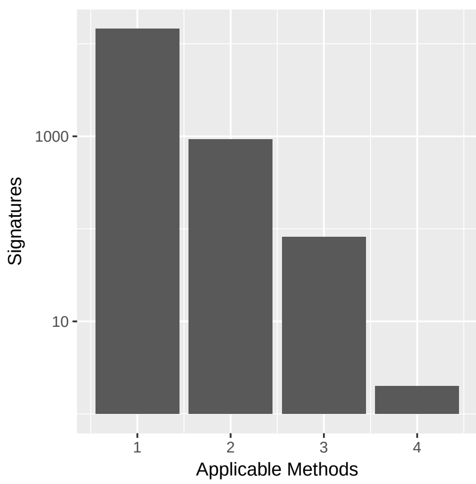


Figure 19: Applicable methods per call signature

2.6.3 *Specializations*

Figure 18 gives the number of specializations per method recorded dynamically on our corpus. The data uses strict eliminations, so that the results from different packages can be summed without duplicate functions. The distribution has a heavy tail, which shows that programmers actually write methods that can be very poly-

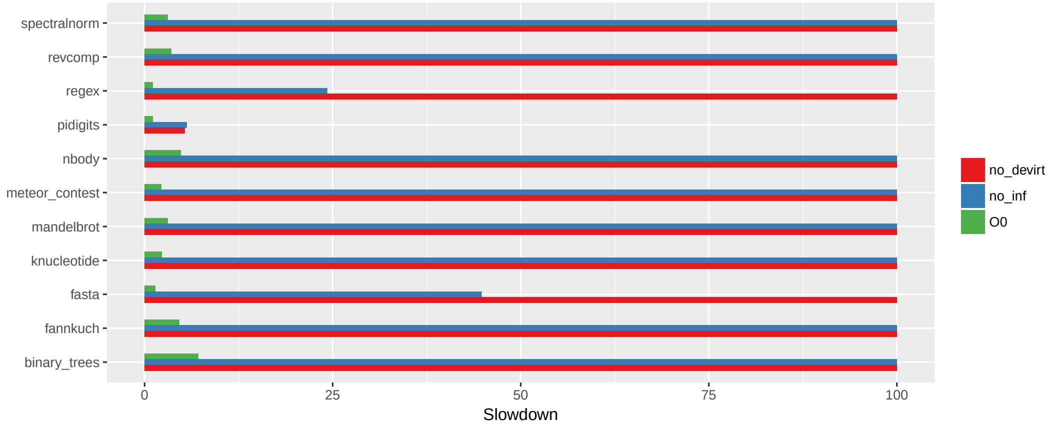


Figure 20: Optimization and performance

morphic. Note that polymorphism is not in contradiction with type stability, since a method called with different tuples of argument types across different call sites can be type stable for each of its call sites. Conversely, 46% of the methods have only been specialized once after running the tests. Many methods are thus used monomorphically: this hints that a number of methods may have a type specification that prevent polymorphism, which means that programmers tend to think of the concrete types they want their methods applied to, rather than only an abstract type specification.

Figure 19 corroborates this hypothesis. It represents the number of applicable methods per call signature. A method is applicable if the tuple of types corresponding to the requirements for its arguments is a supertype of that of the actual call. This data is collected on dynamic traces for functions with at least two methods. 93% of the signatures can only dispatch to one method, which strongly suggests that methods tend to be written for disjoint type signatures. As a consequence it shows that the specificity rules, used to determine which method to call, boil down to subtyping in the vast majority of cases.

2.6.4 *Impact on performance*

Fig. 20 illustrates the impact on performance of LLVM optimizations, type inference and devirtualization. By default Julia uses LLVM at optimization level `O2` . Switch-ing off all LLVM optimizations generates code between 1.1x and 7.1x slower. Turning off type inference means that method are specialized correctly but all internal operations will be performed on values of type `Any` . Functions that have only a single method may still be devirtualized and dispatched to. The graph is capped at 100x slowdown. The actual slowdowns range between 5.6x and 2151x. Lastly, turning off devirtualization implies that no inlining will be performed and all function calls are dispatched dynamically. The slowdowns range between 5.3x and 1905x.

Obviously, Julia was designed to be optimized with type information. These results suggest that performance of fully dynamic code is rather bad. It is likely that if users were to write more dynamic code, some of the techniques that have proved successful for other dynamic languages could be ported to Julia. But clearly, the current implementation crucially relies on code being type stable and on devirtualization and inlining. The impact of the LLVM optimizations is small in comparison.

3 SUBTYPING IN JULIA

Julia's key relation over types is subtyping. Every method invocation is resolved using subtyping both to determine which methods could apply as well as to figure out which is the most specific. Consequently, subtyping is critical both for the semantics of Julia itself and for programmers reasoning about Julia.

Unfortunately, while I have previously formalized the *algorithm* that Julia uses [84] the system has proven theoretically intractable. As will be shown later in this chapter Julia's subtyping relation is very complex and proving meaningful properties about it has proven elusive. Subtyping in Julia is supposed to be based on nominal subtyping—that if the set of values that type A describes is a subset of the set of values of type B then A is a subtype of B and vice versa. Proving even this simple fact for a practical subset of Julia is challenging.

As I were wrestling with this complexity a question came up: if subtyping is this complex then is it even decidable? Julia has bounded quantified types and decidable subtyping in such a setting would be the exception rather than the rule [40, 44]. As it turns out, subtyping in Julia is also provably undecidable; the remainder of this section will be devoted to proving that fact and discussing how I then accommodate this subtyping relation.

3.1 RELATED WORK

Subtyping is key for a language with multiple dispatch. Subtyping is used to decide which methods might be called at a given site or whether a given invocation is safe or not. Static typing for a lanugage with multiple dispatch must then rely extensively on the subtyping relation. Moreover, the decidability of subtyping then determines whether the type system as a whole is decidable.

Parametric polymorphism is the usual pain point; it is easy to create type languages for which subtyping is very hard or impossible to decide with parametric polymorphism. Languages with multiple dispatch differ on whether parametric polymorphism is supported or not. Most previous efforts focused on non-polymorphic types, such as Cecil [17], Typed Clojure [14], and MultiJava [25].

Subtyping is used to check that classes implement all of the required methods of their supertypes. The subtype relations themselves are defined over covariant tuples and discrete unions. Approaches that combine multiple dispatch with parametric polymorphism are more involved.

Mini-Cecil [50, 52] is one example of a language with both universal polymorphism and parametric multiple dispatch. In Mini-Cecil, universal types have only top-level quantifiers. Fortress [4], in addition, supports arrow types, and internally uses both universal and existential types with top-level quantifiers. Mini-Cecil and Fortress both use a constraint generation strategy to resolve subtyping; they support union and intersection types but do not provide distributivity "in order to simplify constraint solving" [52]. For Mini-Cecil typechecking is decidable. Fortress argued decidability based on [15], though no proof is provided.

In type systems with bounded existential types, as well as type systems with nominal subtyping and variance, decidability of subtyping has been a major concern [46, 80]. Pierce demonstrated that even a small subset of System $F_{\leqslant}$ is undecidable [65], demonstrating how easy it is to accidentally be undecidable even with very simple polymorphic type systems.

The decidability of subtyping in practical languages has been extensively studied. For example, Java was shown to be undecidable by Grigore [40] and Scala's current type system is undeicdable [44]. It is relatively easy to accidentally introduce an undecidability into subtyping.

Usually undecidability is not a practical problem for a language. If undecidability arises from features that are uncommonly used, as seen in Java [77], then few programmers are likely to run into programs that fail to compile. Moreover, when undecidability is readily accessible it then became part of the practice of software engineering for the language in question, as seen in C++ [7]. Exploitation of undecidability is also possible: theoretical results about Java have produced encodings of increasingly complex language grammars [35] into the type system. Undecidabilities are managable in practice so long as either the execution model is easy to understand or the critical features or patterns are not typically used.

3.2 FORMALIZATION OF SUBTYPING

Julia's type language is deceptively simple. The language has a number of basic type forms including:

- tag types such as `Int` or `Rational` which are simple inhabitants of an explicit subtyping lattice and can be parameterized such as `Vector{Int}`,
- tuple types `Tuple{Int, Int}`,
- untagged union types `Union{Int, String}`,
- and bounded existential types `Vector{T} where Nothing <: T <: Int`

The first three structures are relatively straightforward; the addition of bounded existential types makes the subtyping relationship much more complicated.

I will not be describing Julia's subtyping relation in depth in this work; instead I will be relying on the formalization of Belyakova et. al. [8]. This formalism captures the overwhelming majority of subtyping for Julia's type language (primarily excluding variadic length tuples). In order to describe the problem with subtyping in Julia I will be using an excerpt of that formalism, shown in figure 21.

I will only describe the part of the formalism required to understand the proof of undecidability. Judgments are of the form $E \vdash \tau <: \tau' \vdash E'$, which should be read as "τ is a subtype of τ' against the environment E producing E'."

Subtyping happens against an environment E that carries the in-scope variables and defines their variance; a variable in E looks like ${}^{S}T_{l}^{u}$, where S is the side (left or right) it first appeared on, l is the lower bound, and u is the upper bound. E may also contain barriers (used to indicate where I switch from a covariant to an invariant context). Left-side variables have forall semantics, while right-side variables have existential semantics.

The proof of undecidability relies on seven subtyping rules that work as follows:

- Refl: Reflexivity of subtyping (every τ is a subtype of itself) is axiomatic within the formalism and makes no demands of or modifications to the environment E.
- Union: I do not reproduce the full generality of left-hand union subtyping as it is not required for the proof of undecidability. Instead, I simply capture that the empty union is the bottom type (e.g. is a subtype of all other types τ). This follows from the original rule Union_Left.
- Tuple: Julia tuples have standard covariant subtyping; each element of the left-hand tuple a_i is checked against the matching element in the right-hand tuple a_i'. The first element is checked against the original context E, begetting the

$$\textsc{Refl}\quad \frac{}{E \vdash \tau <: \tau \vdash E} \qquad \textsc{Union}\quad \frac{}{E \vdash \texttt{Union\{\}} <: \tau \vdash E}$$

$$\textsc{Tuple}\quad \frac{E \vdash a_1 <: a'_1 \vdash E_1 \qquad \ldots \qquad E_{n-1} \vdash a_n <: a'_n \vdash E_n}{E \vdash \texttt{Tuple}\{a_1, \ldots, a_n\} <: \texttt{Tuple}\{a'_1, \ldots, a'_n\} \vdash E_n}$$

$$\textsc{App\_Inv}\quad \frac{\begin{array}{c} n \leqslant m \\ E_0 = \mathit{add}(E, \text{barrier}) \qquad \forall\, 0 < i \leqslant n,\ E_{i-1} \vdash a_i <: a'_i \vdash E'_i \wedge E'_i \vdash a'_i <: a_i \vdash E_i \end{array}}{E \vdash \texttt{name}\{a_1, \ldots, a_m\} <: \texttt{name}\{a'_1, \ldots, a'_n\} \vdash \mathit{del}(\text{barrier}, E_i)}$$

$$\textsc{L\_Intro}\quad \frac{\mathit{add}(E, {}^{L}T^{\tau_2}_{\tau_1}) \vdash \tau <: \tau' \vdash E'}{E \vdash (\tau \texttt{ where } \tau_1 <: T <: \tau_2) <: \tau' \vdash \mathit{del}(T, E')}$$

$$\textsc{R\_Intro}\quad \frac{\mathit{add}(E, {}^{R}T^{\tau_2}_{\tau_1}) \vdash \tau <: \tau' \vdash E' \qquad \mathit{consistent}(T, E')}{E \vdash \tau <: (\tau' \texttt{ where } \tau_1 <: T <: \tau_2) \vdash \mathit{del}(T, E')}$$

$$\textsc{L\_Right}\quad \frac{\mathit{search}(T, E) = {}^{L}T^{u}_{l} \qquad E \vdash l <: \tau \vdash E'}{E \vdash \tau <: T \vdash E'} \qquad \textsc{L\_Left}\quad \frac{\mathit{search}(T, E) = {}^{L}T^{u}_{l} \qquad E \vdash u <: \tau \vdash E'}{E \vdash T <: \tau \vdash E'}$$

$$\textsc{R\_Right}\quad \frac{\begin{array}{c} \mathit{search}(T, E) = {}^{R}T^{u}_{l} \\ (\mathit{is\_var}(\tau) \wedge \mathit{search}(\tau, E) = {}^{L}S^{u'}_{l'}) \implies \neg \mathit{outside}(T, S, E) \qquad E \vdash \tau <: u \vdash E' \end{array}}{E \vdash \tau <: T \vdash \mathit{upd}({}^{R}T^{u}_{\texttt{Union}\{l,\tau\}}, E')}$$

$$\textsc{R\_Left}\quad \frac{\mathit{search}(T, E) = {}^{R}T^{u}_{l} \qquad E \vdash l <: \tau \vdash E'}{E \vdash T <: \tau \vdash \mathit{upd}({}^{R}T^{t}_{l}, E')}$$

$$\textsc{R\_L}\quad \frac{\begin{array}{c} \mathit{search}(T_1, E) = {}^{R}{T_1}^{u_1}_{l_1} \qquad \mathit{search}(T_2, E) = {}^{L}{T_2}^{u_2}_{l_2} \\ \mathit{outside}(T_1, T_2, E) \implies E \vdash u_2 <: l_2 \vdash E' \qquad E \vdash u_1 <: l_2 \vdash E'' \end{array}}{E \vdash T_1 <: T_2 \vdash \mathit{upd}({}^{R}{T_1}^{u_1}_{\texttt{Union}\{T_1, l_1\}}, E')}$$

Figure 21: Julia subtyping (extract)

context E_1, and so on, until all pairs of elements have ben checked. The final context E_n is then returned.

- App_Inv: In contrast, constructor applications (such as `Vector{T}`) are invariant on their arguments where tuples were covariant. The system implements this by appending a barrier element to the context E, then checking that for each pair a_i and a_i' that first $E_{i-1} \vdash a_i <: a_i' \vdash E_i'$ and then that $E_i' \vdash a_i' <: a_i \vdash E_i$. Checking both directions of subtyping ensures that the final environment E_i requires that a_i' and a_i are equal.

- L_Intro: Left-hand addition of variables is straightforward. The system checks that the body of the introduction form τ is a subtype of the right-hand side τ' against the environment extended with T, $add(E, {}^{L}T_{\tau_1}^{\tau_2})$. Once the subtyping relation has been established into the environment E' this new variable is then removed before returning using *del*.

- R_Intro: Right-hand addition works analogously, albeit with the introduction of the *consistent* metafunction. *consistent* simply ensures that the lower bound of T (which may have changed in the interim) is a subtype of the upper bound in E'.

- L_Right: The R/L_Right/Left rules handle cases where a "bare" type variable appears in the subtyping relationship. The L/R refers to where the variable was originally introduced (on the left or right hand side, respectively), while the Right/Left refers to where the variable appeared.

  In the case of L_Right, this was a variable that first appeared on the left-hand-side that is now appearing on the right hand side. Left-hand variables have forall semantics, so I need to ensure that all possible instantiations will be consistent with the bound implied by this subtyping relationship. As a result, to show that $E \vdash \tau <: T \vdash E'$ I need to show that the lower bound of T in E, l, is a subtype of τ using $E \vdash l <: \tau \vdash E'$.

- L_Left: The left-left case is then analogous to the left-right case discussed above. Instead of checking that the lower bound is consistent with the left-hand type expression, I instead check

- R_Right: Right-hand type variables are existentially, rather than universally, quantified. As a result, I modify right-hand variables to ensure that they are consistent with the relation that I re trying to prove.

Trivially, I need to ensure that the existing upper bound is consistent with this new lower bound, which is checked with $E \vdash \tau <: u \vdash E'$. I also need to ensure that if the left-hand-side is a left-introduced variable (which is checked using *is_var*(τ) and *search*$(\tau, E) = {}^{L}S_{l'}^{u'}$)) that it shares the same variance as I do. This prevents right-hand variables that are introduced inside of a invariant context from being instantiated with lower bound begotten from covariantly-quantified variables outside of that context.

For example, consider the statement `(Vector{T} where T<:Real) <: (Vector{U where U<:Real})`. Intuitively, this is a false proposition: the right-hand vector is one of heterogenous real numbers, whereas the left-hand vector is of a homogenous single (albeit unspecfied) type of real number. This is equivalent to ${}^{L}T^{\texttt{Real}}$ *barrier* ${}^{R}U^{\texttt{Real}} \vdash$ `Vector{T}` $<:$ `Vector{U}`. If I allowed R_RIGHT to apply in this case then the judgment holds if ${}^{L}T^{\texttt{Real}}$ *barrier* ${}^{R}U^{\texttt{Real}} \vdash$ `Real` $<:$ `Real` (by applying L_LEFT) which is trivially true. Thus, I cannot allow R_RIGHT to hold in invariant contexts since I cannot instantiate an invariant variable to be equal to a covariant one. Instead, the rule R_L needs to be used.

- R_LEFT: In the right-left case, here, I have a right-hand variable T on the left-hand side of a subtyping judgment. I need to ensure that the bounds on T remain consistent (by checking that $E \vdash l <: \tau \vdash E'$). If all of these are satisifed, then our resulting environment is *upd*$({}^{R}T_{l}^{\tau}, E')$, or E' updated with T upper bounded by the new τ.

- R_L: If I are checking subtyping between two variables that are both on the "wrong" side (e.g. a right-introduced variable on the left and vice versa) I need to perform a more complex procedure. *outside*(T_1, T_2, E) is true if T_1 precedes T_2 in E and is separated by a barrier; that is, if T_2 is invariant and T_1 is an earlier covariant definition. If this is the case I need to constrain T_1 and T_2 to be equal by checking that the upper bound of T_2 is a subtype of the lower bound of T_1. In any case, I also need to ensure that u_1 is a subtype of l_2.

  A brief example is in the same test I used for R_RIGHT where I are testing if a quantified vector is a subtype of a vector of quantified values. In this case, I need to check that the upper bound of U, `Real`, is a subtype of the (implicit) lower bound of T, `Union{}`. This is false and causes the result of subtyping in that example to be correct.

τ^+	$::=$		*Positive types*	τ^-	$::=$		*Negative types*
	$\mid$	Top	Top type		$\mid$	α	Type variable
	$\mid$	$\neg\tau^-$	Negative negation		$\mid$	$\neg\tau^+$	Positive negation
	$\mid$	$\forall\alpha \leqslant \tau^-.\tau^+$	Positive quantification		$\mid$	$\forall\alpha \leqslant \mathit{Top}.\tau^-$	Negative quantification

$$\textsc{PTop}\ \frac{}{\Gamma^- \vdash \tau^- \leqslant \mathit{Top}} \qquad \textsc{PVar}\ \frac{\Gamma^- \vdash \Gamma^-(\alpha) \leqslant \tau^+}{\Gamma^- \vdash \alpha \leqslant \tau^+} \qquad \textsc{PAll}\ \frac{\Gamma^-, \alpha \leqslant \phi^- \vdash \sigma^- \leqslant \tau^+}{\Gamma^- \vdash \forall\alpha \leqslant \mathit{Top}.\sigma^- \leqslant \forall\alpha \leqslant \phi^-.\tau^+} \qquad \textsc{PNeg}\ \frac{\Gamma^- \vdash \tau^- \leqslant \sigma^+}{\Gamma^- \vdash \neg\sigma^+ \leqslant \neg\tau^-}$$

Figure 22: Subtyping for System $F^P_\leqslant$

3.3 PROOF OF UNDECIDABILITY

Our proof of undecidability proceeds by reduction of subtyping in one of the intermediate deterministic fragments of System $F_\leqslant$ as described by Pierce [66] System $F^P_\leqslant$, to Julia subtyping. I do this by translating System $F^P_\leqslant$ judgements to Julia subtyping judgements and vice versa. At a high level, the translation works by flipping upper-bounded universal quantification in $F^P_\leqslant$ to lower-bounded existential quantification in Julia in the opposite order. I will begin by describing System $F^P_\leqslant$ and then our reduction from subtyping in System $F^P_\leqslant$ to subtyping in Julia.

3.3.1 *System* $F^P_\leqslant$

System $F^P_\leqslant$ is a restricted version of Pierce's System $F_\leqslant$ without arrow types and with types that carry explicit information about whether they appear in negative or positive (left or right) position. I provide the grammar and subtyping rules for System $F^P_\leqslant$ in figure 22. Following Pierce, $\Gamma^-(\alpha) = \tau$ holds if $\alpha \leqslant \tau \in \Gamma$.

Pierce showed that subtyping in System $F^P_\leqslant$ is turing complete by reducing reduction in a rowing machine to subytping in a further reduced version called System $F^D_\leqslant$. I focus here on System $F^P_\leqslant$ as each rule is simpler compared to System $F^D_\leqslant$ and begets a more consice translation to and from Julia subtyping terms.

$$
\begin{array}{rcll}
[\![\Gamma^-, \alpha \leqslant \phi^-]\!] & ::= & [\![\Gamma^-]\!]\, {}^L\alpha_l^u\, {}^R\beta_\alpha^\alpha & \textit{Environments} \\
[\![\cdot]\!] & ::= & \cdot & \\
 & & & \\
[\![\mathit{Top}]\!] & ::= & \texttt{Union\{\}} & \textit{Positive types} \\
[\![\neg\tau^-]\!] & ::= & \texttt{Neg}\{[\![\tau^-]\!]\} & \\
[\![\forall\alpha \leqslant \tau^-.\tau^+]\!] & ::= & \texttt{Tuple\{All}\{\alpha\}, [\![\tau^+]\!]\}\ \texttt{where}\ [\![\tau^-]\!] <: \alpha <: \texttt{Any} & \\
 & & & \\
[\![\alpha]\!] & ::= & \alpha & \textit{Negative types} \\
[\![\neg\tau^+]\!] & ::= & \texttt{Neg}\{\alpha\}\ \texttt{where}\ [\![\tau^+]\!] <: \alpha <: \texttt{Any} & \\
[\![\forall\alpha \leqslant \mathit{Top}.\tau^-]\!] & ::= & \texttt{Tuple\{All}\{\alpha\}, [\![\tau^-]\!]\}\ \texttt{where Union\{\}} <: \alpha <: \texttt{Any} &
\end{array}
$$

Figure 23: Translation from System $F^P_\leqslant$ to Julia

3.3.2 *Reduction from System $F^P_\leqslant$ to Julia subtyping*

To show turing completeness of subtyping in Julia I show that that there exists a translation, denoted $[\![\tau]\!]$, such that for any System $F^P_\leqslant$ environment Γ^- and types τ, σ there exists a resultant Julia environment E where $\Gamma^- \vdash \tau \leqslant \sigma \iff [\![\Gamma^-]\!] \vdash [\![\sigma]\!] <: [\![\tau]\!] \vdash E$.

The translation needs to convert environments Γ^- into Julia subtyping environments E as well as both positive and negative types τ^+ and τ^-. Our translation rules are depicted in 23. I use the nominal type constructors `Neg{}` and `All{}` simply to transition into distinguishable invariant contexts; their definitions are simply `struct NegT end` and `struct AllT end`.

I assume that variable names in System $F^P_\leqslant$ and Julia are equivalent for the purposes of clarity and to avoid confusion; I will be referring to variables using the System $F^P_\leqslant$ terminology α and β. Note that I implicitly map System $F^P_\leqslant$ variables to a pair of Julia variables; I treat the System $F^P_\leqslant$ environment Γ^- as mapping both of these Julia variables to their originating System $F^P_\leqslant$ bound.

The translation of the environment is simple: for a given System $F^P_\leqslant$ variable α it creates the matching Julia type variables α and β and bounds the right-hand variable β to be equal to α.

Positive and negative type translation is simple: I take a universally quantified term with an upper bound and flip it into an existentially quantified term with a lower bound. Additionally, I introduce a tuple that contains an invariant reference cell containing the newly-introduced variable as its first element and the translated version of the type being quantified over as the second element. Translation for negated types follows from the original System $F_{\leqslant}$ definition of negation wherein $\neg\tau \equiv \forall\alpha \leqslant \tau.\alpha$; I simply translate the negation as if it were explicitly written out with this definition.

FORWARD I begin by showing that $\Gamma^- \vdash \tau^- \leqslant \sigma^+ \implies [\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\tau^-]\!] \vdash E$. To do so, I proceed by induction on the System $F_{\leqslant}$ rule used to derive the premise.

- PTop; want to show that $[\![\Gamma^-]\!] \vdash [\![\mathit{Top}]\!] <: [\![\tau^-]\!] \vdash E'$ for some E'. This follows trivially from rule Top as $[\![\mathit{Top}]\!]$ is `Union{}`, the bottom type.
- PVar; want to show that $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: [\![\alpha]\!] \vdash E'$. I know that $\Gamma^- \vdash \Gamma(\sigma) \leqslant \tau^+$ so by the IH there is some E' such that $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: [\![\Gamma^-(\alpha)]\!] \vdash E'$. Let $\sigma = \Gamma^-(\alpha)$; note that this implies $\alpha \leqslant \sigma \in \Gamma^-$, so $[\![\Gamma^-(\alpha)]\!] = [\![\sigma]\!]$. Therefore, by L_Right, $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: \alpha \vdash E'$ since $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: [\![\sigma]\!] \vdash E'$ because by definition of environment translation ${}^{L}\alpha^{\texttt{Any}}_{[\![\sigma]\!]} \in [\![\Gamma^-]\!]$. Finally, $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: [\![\alpha]\!] \vdash E'$ as $[\![\alpha]\!] = \alpha$.
- PAll: I want to show $[\![\Gamma^-]\!] \vdash [\![\forall\alpha \leqslant \phi^-.\tau^+]\!] <: [\![\forall\beta \leqslant \mathit{Top}.\sigma^-]\!] \vdash [\![\Gamma^-]\!]$. Equivalently, applying the translation, I want to show that $[\![\Gamma^-]\!] \vdash \texttt{Tuple\{All\{}\alpha\texttt{\}},[\![\tau^+]\!]\texttt{\} where } [\![\phi^-]\!] <: \alpha <: \texttt{Any} <: \texttt{Tuple\{All\{}\beta\texttt{\}},[\![\sigma^-]\!]\texttt{\} where Union\{\}} <: \beta <: \texttt{Any} \vdash [\![\Gamma^-]\!]$. To show this, I must apply L_Intro and R_Intro, making our goal $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{\}}} \vdash \texttt{Tuple\{All\{}\alpha\texttt{\}},[\![\tau^+]\!]\texttt{\}} <: \texttt{Tuple\{All\{}\beta\texttt{\}},[\![\sigma^-]\!]\texttt{\}} \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\alpha}_{\texttt{Union\{}\alpha\texttt{\}}}$. I are left with two subgoals after applying Tuple.
  - $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{\}}} \vdash \texttt{All\{}\alpha\texttt{\}} <: \texttt{All\{}\beta\texttt{\}} \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\alpha}_{\texttt{Union\{}\alpha\texttt{\}}}$; which I resolve by applying App_Inv, again begetting two subgoals.
    * $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{\}}} \vdash \alpha <: \beta \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{}\alpha\texttt{\}}}$, the forward direction of subtyping. I get $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{\}}} \vdash \alpha <: \texttt{Any} \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{}\alpha\texttt{\}}}$ by applying R_Right, then $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{\}}} \vdash \texttt{Any} <: \texttt{Any} \vdash [\![\Gamma^-]\!]$ by L_Left.
    * The converse case, $[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union\{}\alpha\texttt{\}}} \vdash \beta <: \alpha \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\alpha}_{\texttt{Union\{}\alpha\texttt{\}}}$; I first apply R_Left to get

$[\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{\alpha\}} \vdash \texttt{Union\{\}} <: \alpha \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{\alpha\}}$, which holds trivially.

– By the IH, $[\![\Gamma^-, \alpha \leqslant \phi^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\alpha}_{\texttt{Union}\{\alpha\}} \vdash [\![\tau^+]\!] <: [\![\sigma^-]\!] \vdash [\![\Gamma^-]\!], {}^{L}\alpha^{\texttt{Any}}_{[\![\phi^-]\!]}, {}^{R}\beta^{\alpha}_{\texttt{Union}\{\alpha\}}$.

- PNeg: I want to show $[\![\Gamma^-]\!] \vdash [\![\neg\tau^-]\!] <: [\![\neg\sigma^+]\!] \vdash [\![\Gamma^-]\!]$, or, equivalently under translation, that $[\![\Gamma^-]\!] \vdash \texttt{Neg}\{[\![\tau^-]\!]\} <: \texttt{Neg}\{\alpha\}$ where $[\![\sigma^+]\!] <: \beta <: \texttt{Any} \vdash [\![\Gamma^-]\!]$. I show this by applying rule R_Intro begetting $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{[\![\sigma^+]\!]} \vdash \texttt{Neg}\{[\![\tau^-]\!]\} <: \texttt{Neg}\{\beta\} \vdash E$. Note that while E contains β it will be removed by R_Intro in the output environment. Rule App_Inv then applies, giving us two goals, one for each direction.
  - In the forward direction I want to show that $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{[\![\sigma^+]\!]} \vdash [\![\tau^-]\!] <: \beta \vdash E'$ for some E'. Only rule R_Right can apply, the application of which requires us to show $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{[\![\sigma^+]\!]} \vdash [\![\tau^-]\!] <: \texttt{Any} \vdash [\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{[\![\sigma^+]\!]}$ which holds trivially by Top. Therefore E' must be $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{[\![\sigma^+]\!],[\![\tau^-]\!]\}}$.
  - In the backwards direction I want to show that $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{[\![\sigma^+]\!],[\![\tau^-]\!]\}} \vdash \beta <: [\![\tau^-]\!] \vdash E$. I must do so by applying rule R_Left, which then requires us to show that $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{[\![\sigma^+]\!],[\![\tau^-]\!]\}} \vdash \texttt{Union}\{[\![\sigma^+]\!], [\![\tau^-]\!]\} <: [\![\tau^-]\!] \vdash E$. Applying rule Union_Left gives us two ensuing cases:
    * First, I need to show $[\![\Gamma^-]\!], {}^{R}\beta^{\texttt{Any}}_{\texttt{Union}\{[\![\sigma^+]\!],[\![\tau^-]\!]\}} \vdash [\![\sigma^+]\!] <: [\![\tau^-]\!] \vdash E''$. I apply the weakening lemma to simplify this to $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\tau^-]\!] \vdash E'''$, which holds by the IH.
    * Next, I need to show that $E''' \vdash [\![\tau^-]\!] <: [\![\tau^-]\!] \vdash E'''$. I trivially apply reflectivity to show the rule and conclude that $E = E'''$.

REVERSE The reverse direction consists of showing that $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\tau^-]\!] \vdash E \implies \Gamma^- \vdash \tau^- \leqslant \sigma^+$. I proceed by rule induction on the Julia subtyping rule used to derive $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\tau^-]\!] \vdash E$ while performing case analysis of the structure of the translated type. I begin by case analyzing on the left-hand side τ^- to see if it is a variable or not, then on the right-hand-side σ^+. eing translated which then uniquely identifies the rule being applied.

- $\tau^- = \alpha$. I know that $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\alpha]\!] \vdash [\![\Gamma^-]\!]$ or equivalently $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: \alpha \vdash [\![\Gamma^-]\!]$. If α is a left-hand variable, then I know that ${}^{L}\alpha^{\texttt{Any}}_{[\![\sigma^-]\!]}$ is in the environment as L_RIGHT must have been used to resolve it and by definition of environment translation $\alpha \leqslant \sigma^- \in \Gamma$. I then know that $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\sigma^-]\!] \vdash [\![\Gamma^-]\!]$ and by the IH that $\Gamma^- \vdash \sigma^- \leqslant \sigma^+$ or that $\Gamma^- \vdash \Gamma^-(\alpha) \leqslant \sigma^+$. If α is a right-hand variable then I know that ${}^{L}\beta^{\texttt{Any}}_{[\![\sigma^-]\!]}\,{}^{R}\alpha^{\beta}_{\beta} \in [\![\Gamma^-]\!]$ by definition of environment translation and that $\beta \leqslant \sigma^- \in \Gamma^-$. I must have used rule R_RIGHT to conclude this, so I then know that $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: \beta \vdash E$ which then implies $[\![\Gamma^-]\!] \vdash [\![\sigma^+]\!] <: [\![\sigma^-]\!] \vdash E$ from which $\Gamma^- \vdash \sigma^- \leqslant \sigma^+$ follows. Then, $\Gamma^- \vdash \Gamma^-(\beta) \leqslant \sigma^+$ and thus $\Gamma^- \vdash \Gamma^-(\alpha) \leqslant \sigma^+$.

- $\sigma^+ = \mathit{Top}$; therefore $\Gamma^- \vdash \tau^- \leqslant \sigma^+$ holds iff $\Gamma^- \vdash \tau^- \leqslant \mathit{Top}$ which holds by PTOP.

- $\sigma^+ = \neg\sigma^-$. First, I are given that $[\![\Gamma^-]\!] \vdash \texttt{Neg}\{[\![\sigma^-]\!]\} <: [\![\tau^-]\!] \vdash [\![\Gamma^-]\!]$. It follows that $[\![\tau^-]\!]$ must be of the form $\texttt{Neg}\{[\![\tau^+]\!]\}$ as this judgment must have been concluded by first applying L_INTRO followed by APP_INV; this can only occur when the constructor names match and therefore there must exist some τ^+ such that $\tau^- = \neg\tau^+$. Thus, I want to show that $\Gamma^- \vdash \neg\tau^+ \leqslant \neg\sigma^-$ given $[\![\Gamma^-]\!] \vdash \texttt{Neg}\{[\![\sigma^-]\!]\} <: \texttt{Neg}\{\beta\}\ \texttt{where}\ [\![\tau^+]\!] <: \beta <: \texttt{Any} \vdash [\![\Gamma^-]\!]$. To conclude the latter R_INTRO must have been used, so I have $[\![\Gamma^-]\!], {}^{R}\beta_{[\![\tau^+]\!]}\texttt{Any} \vdash \texttt{Neg}\{[\![\sigma^-]\!]\} <: \texttt{Neg}\{\beta\} \vdash E$. Rule APP_INV must have then been applied to find that there exists some environment E' such that $[\![\Gamma^-]\!], {}^{R}\beta_{[\![\tau^+]\!]}\texttt{Any} \vdash [\![\sigma^-]\!] <: \beta \vdash E'$ (1) and $E' \vdash \beta <: [\![\sigma^-]\!] \vdash E$ (2). Rule R_RIGHT must have been used to resolve (1) so therefore $E' = [\![\Gamma^-]\!], {}^{R}\beta_{\texttt{Union}\{[\![\tau^+]\!],[\![\sigma^-]\!]\}}\texttt{Any}$ by applying ANY to show the bound. Thus, to show $[\![\Gamma^-]\!], {}^{R}\beta_{\texttt{Union}\{[\![\tau^+]\!],[\![\sigma^-]\!]\}}\texttt{Any} \vdash \beta <: [\![\sigma^-]\!] \vdash E$, we must have derived $[\![\Gamma^-]\!], {}^{R}\beta_{\texttt{Union}\{[\![\tau^+]\!],[\![\sigma^-]\!]\}}\texttt{Any} \vdash \texttt{Union}\{[\![\tau^+]\!], [\![\sigma^-]\!]\} <: [\![\sigma^-]\!] \vdash E$ to apply R_LEFT. Rule UNION_LEFT must have been applied with $[\![\Gamma^-]\!], {}^{R}\beta_{\texttt{Union}\{[\![\tau^+]\!],[\![\sigma^-]\!]\}}\texttt{Any} \vdash [\![\tau^+]\!] <: [\![\sigma^-]\!] \vdash E''$ as a precondition. Since β is fresh, $[\![\Gamma^-]\!] \vdash [\![\tau^+]\!] <: [\![\sigma^-]\!] \vdash [\![\Gamma^-]\!]$ then follows, and therefore by application of the IH $\Gamma^- \vdash \sigma^- \leqslant \tau^+$. Thus, I can apply PVAR to conclude that $\Gamma^- \vdash \neg\tau^+ \leqslant \sigma^-$ or $\Gamma^- \vdash \tau^- \leqslant \sigma^+$.

- $\sigma^+ = \forall\alpha \leqslant \sigma^-.\tau^+$. By similar reasoning to the prior case I can derive that $\tau^- = \forall\beta.\phi$. Thus, we want to show that $\Gamma^- \vdash \neg(\forall\beta \leqslant \mathit{Top}.\phi) \leqslant \forall\alpha \leqslant \sigma^-.\tau^+$ given that $[\![\Gamma^-]\!] \vdash [\![\forall\alpha \leqslant \sigma^-.\tau^+]\!] <: [\![\forall\beta \leqslant \mathit{Top}.\phi]\!] \vdash E$. I expand the translations to get $[\![\Gamma^-]\!] \vdash \texttt{Tuple}\{\texttt{All}\{\alpha\}, [\![\tau^+]\!]\}\ \texttt{where}\ [\![\sigma^-]\!] <: \alpha <: \texttt{Any} <: \texttt{Tuple}\{\texttt{All}\{\beta\}, [\![\phi]\!]\}\ \texttt{where}\ \texttt{Union}\{\} <: \beta <: \texttt{Any} \vdash E$. I had to have applied L_INTRO and R_INTRO to get $G \vdash \texttt{Tuple}\{\texttt{All}\{\alpha\}, [\![\tau^+]\!]\} <:$

$\texttt{Tuple}\{\texttt{All}\{\beta\}, [\![\phi]\!]\} \vdash E$ where $G = [\![\Gamma^-]\!], {}^L\alpha^{\texttt{Any}}_{[\![\sigma^-]\!]}, {}^R\beta^{\texttt{Any}}_{[\![\phi]\!]}$. To have concluded this, I need to have had some intermediate context E' such that I can use TUPLE to conclude both $G \vdash \texttt{All}\{\alpha\} <: \texttt{All}\{\beta\} \vdash E'$ (1) and $E' \vdash [\![\tau^+]\!] <: [\![\phi]\!] \vdash E$ (2). Furthermore, to have concluded (1), I had to have applied APP_INV with intermediate context E'' such that $G \vdash \alpha <: \beta \vdash E''$ (1.1) and $E'' \vdash \beta <: \alpha \vdash E'$. The same sequence as applies in the forward case must be used to conclude these judgements, so $E' = [\![\Gamma^-]\!]^L\alpha_{[\![\sigma^-]\!]^{\texttt{Any}}}{}^R\beta^{\alpha}_{\alpha}$. Note that $[\![\Gamma^-, \alpha \leqslant \sigma^-]\!] = E'$. Therefore, from (2), it follows that $[\![\Gamma^-, \alpha \leqslant \sigma^-]\!] \vdash [\![\tau^+]\!] <: [\![\phi]\!] \vdash E$. Thus, by the IH, I conclude that $\Gamma^-, \alpha \leqslant \sigma^- \vdash \phi \leqslant \tau^+$ and then by application of PALL that $\Gamma^- \vdash \forall\beta.\phi \leqslant \forall\alpha \leqslant \sigma^-.\tau^+$.

Therefore, System $F^P_{\leqslant}$ subtyping holds if and only if the translated version in Julia holds. By Pierce's result, then, I can conclude that subtyping in Julia is undecidable.

3.4 UNDECIDABILITY IN PRACTICE

The undecidability result is not a purely theoretical one; the described translation is able to match the undesirable behavior of System $F^P_{\leqslant}$ in Julia. For example, Ghelli's looping gadget can be trivially translated into Julia as shown in figure 24.

```
function Neg(T)
  return (Ref{X} where X>:T)
end
function Kappa(T)
  return (Tuple{Ref{Y},Neg(Y)} where Y>:T)
end
const Theta = (Tuple{Ref{Z},Neg(Kappa(Z))} where Z)
Kappa(Theta) <: Theta # stackoverflow error
```

Figure 24: Ghelli's looping gadget, Julia version.

Similarly, I can mechanically translate any System $F^P_{\leqslant}$ judgement (including typing context Γ) into Julia and check it using Julia's typechecker with predictable results, seen in figure 25. The translation takes a Julia representation of a System $F^P_{\leqslant}$ environment (a dictionary mapping names to System $F^P_{\leqslant}$ upper bound types) and a pair of System $F^P_{\leqslant}$ types to compare and translates it into an equivalent Julia subtype check.

Translation itself follows the construct used in the proof exactly, though getting it started takes more effort in the form of a small gadget.

The gadget ensures that the System $F^{P}_{\leqslant}$ variables are the same on both sides of the Julia subtyping judgment. I accomplish this by having both encoded Julia types `A` and `B` begin on the left side of the judgment in the tuple `efA, RefV where Vl>:BTupleR` around which the in-scope variables are added. The right hand side is then constructed as `efT, RefV where Vr>:TTupleR where T`. Julia then first concludes `RefA <: RefT` therein forcing `A` and `T` to be equal. Julia then checks if `lr(RefV where Vl>:B) <: (RefV where Vr>:T)` or, substituting for `T`, if `lr(RefV where Vl>:B) <: (RefV where Vr>:A)`. Running in the forward direction, Julia then ensures that `Vl <: Vr` using R_Right, then checks the opposite direction `Vr <: Vl`. Applying R_Left then L_Right then gives us `A <: B` as the ultimate judgment that needs to be concluded.

While somewhat contrived, these examples demonstrate that it is possible to get undesirable behavior out of Julia's type system and that deep theoretical analysis is probably intractable with the current conception.

The practical impact of subtyping's undecidability is minor. The key feature being used, lower bounds on type variables, is extremely rare in practice. The only identifiable uses of it occur in one place in the entire standard library, nowhere in broader user code, and rarely as a result of type inference. As a result, the main impacts are on the theory of the system.

From the perspective of a type system the result is frustrating: it means that static dispatch resolution is essentially incomplete if it is to terminate. After all, if methods are chosen with subtyping and subtyping is undecidable, then I cannot universally decide which methods will be invoked in a finite amount of time. My objective is instead to try to find a subset of the type language that is *sufficient* to work practically while skirting around the unpleasant generalities.

Julia programmers are already reasoning extensively about types and Julia programs, obviously, already make extensive use of types. It follows, then, that a reasonable implementation of subtyping already exists as part of the Julia compiler. Moreover, Julia's own implementation of subtyping is the one against which programmers judge whether their types are too complex or not; if a relation sends the implementation into an infinite loop most programmers will consider that a bug. Therefore, I will use the notion of subtyping as implemented in Julia.

Adopting this concept leads to a problem. Julia implements subtyping as a very large, relatively opaque, code base that is not tractable to theoretical examination.

```
abstract type FSubType end
struct FSubVar <: FSubType
  name::Symbol
end
struct FSubTop <: FSubType end
struct FSubUni <: FSubType
  binding::Symbol
  ub::FSubType
  body::FSubType
end


enc(v::FSubVar, eenv::Dict{Symbol,TypeVar}) = eenv[v.nam
enc(v::FSubTop, eenv::Dict{Symbol, TypeVar}) = Union{}
function enc(v::FSubUni, eenv::Dict{Symbol, TypeVar})
  nvn = TypeVar(gensym(v.binding), enc(v.ub, eenv), Any)
  eenv[v.binding] = nvn
  return UnionAll(nvn, Tuple{Ref{nvn}, enc(v.body, eenv)})
end
enc(v::FSubType) = enc(v, Dict{Symbol, TypeVar}())

enc(env::Dict{Symbol,FSubType}) = Dict(k => begin nvn = TypeVar(gensym(k)); nvn.
    lb = enc(v); nvn end for (k,v) in env)

esub(a::FSubType,b::FSubType) = enc(b) <: enc(a)
function esub(a::FSubType, b::FSubType, env::Dict{Symbol,FSubType})
  tenv = enc(env)
  A = enc(b, tenv)
  B = enc(a, tenv)
  lhs = foldl((t,v) -> UnionAll(v,t), values(tenv); init=Tuple{Ref{A}, Ref{V}
      where V>:B})
  rhs = (Tuple{Ref{T}, Ref{V} where V>:T} where T)
  return lhs <: rhs
end
```

Figure 25: Implementation of translation from System $F^{P}_{\leqslant}$ subtyping judgments to Julia subtyping.

The simplified formalization after all has proven largely too difficult to reason about. Modeling the implementation is practically infeasible and any such effort would be unlikely to produce pleasant formal results.

As a result, instead of a concrete formal model of subtyping I will instead be relying on a "black box" concept of what subtyping is. In place of a concrete modelled algorithm for subtyping my type system instead relies on subtyping adhering to a set of properties; the type system will then work with any satisfying instantiation of this relation. This design accommodates both future extensibility (as the type language and underlying subtyping relation are not linked to any specific definition) and the inherent need to choose trade-offs when implementing subtyping in Julia.

For the purposes of typing *itelf*, luckily, relatively few properties are needed of subtyping. As will be described later, one property is required: if $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_3$ then $\tau_1 <: \tau_3$—that subtyping is transitive.

4 GRADUAL TYPING FOR JULIA

I have discussed the complexities in typing Julia at length; what remains is to actually type Julia; now, what remains is to describe how to type Julia itself and sketch how some abstractions in Julia might be captured.

I extend Julia to create Typed Julia. Typed Julia builds on normal Julia by adding two constructs:

- Typed methods that are statically guaranteed to be correct with respect to their type arguments.
- Protocols that require that a suitable implementation of a method exists for some set of types.

These typed features coexist with untyped Julia code, allowing programs to be incrementally specified and typed.

The core of Typed Julia is the ability to write typed methods. Typed methods are syntactically identical to untyped methods, differing only in that they are annotated with the `@typed` annotation. Only methods that are annotated as `@typed` will be statically checked.

As one kind of abstraction, Typed Julia also supports protocols. A protocol, declared using the annotation `@protocol`, simply asserts the existence of methods that can handle every instantiation of some signature. The type system then verifies this assertion using a mechanism similar to checking completeness of pattern matching.

The guarantee of Typed Julia is that "typed code will not go wrong," with the exception of method ambiguities. This safety property is quite robust: it can be broken neither by untyped code nor by use of `eval` (up to a point); only when new methods have been dynamically added and execution has returned to the top level can the safety property be broken. Therefore, Typed Julia provides an unusually strong soundness guarantee for a gradually typed language. Consider a small example of (untyped) Julia code:

```
abstract type AbstractArray end
struct List <: AbstractArray
    array::Array{Int, 1}
end
struct Range <: AbstractArray
    start::Int
    end::Int
end
length(v::List) = length(v.array)
size(u::Range) = u.end - u.start
```

Here I define two kinds of array-like-thing: a list (backed by an array) and a range (backed by a start and an end index). The `List` supports a `length` method that returns the length of the underlying array, while `Range` has `size` that determines the number of elements in the range.

Suppose, now, that I wanted to implement a method `array_like` that allocated a basic array that is the same size as the input, whatever that input array is. One, flawed, implementation might be to say

```
array_like(a::AbstractArray) = Array(length(a))
```

This implementation has a simple bug: it will fail if it is ever given a `Range`. However, if I only ever tested `array_like` with `List`s I would only find out about this after someone tried it with a `Range` and complained.

Bugs where a program assumes that a special-case function is universally applicable are extremely common in Julia. One real example is the use of `for i in 1:length(A)` to loop through every index in an array [1]. Not all `AbstractArray`s are integer indexable in the first place and not all of those arrays start at 1. However, most libraries and users test against basic `Array` which are both integer indexable and start at 1 so do not notice the problem.

This problem of `length` is one of both definition and usage. On the definition side there was nothing to tell us that users might expect every implementation to have a `length`. On the usage side, nothing was obviously wrong when I called a function that only exists on *some* abstract arrays. To fix this problem I need to check both definition and usage sites.

1 1:length(A) considered harmful — or how to make Julia code "generic safe"

Let us start by examining how Typed Julia handles usage sites through gradual typing. In Typed Julia, only methods with the `@typed` annotation need to pass the type checker. I need only to check the methods that I am currently implementing. To start with, then, let's add the `@typed` annotation to our new `array_like` method:

```
@typed array_like(a::AbstractArray) = Array(length(a))
```

Now I run the Typed Julia checker on our program giving us the error

```
No implementations of "length" found for type (::AbstractArray). Suitable
    implementations:
        length(::List)
```

The type checker has identified that not all `AbstractArray`s implement `length`; while one exists for `Range`, no such method exists that can handle every `AbstractArray`. I can use typed Julia's other feature, protocols, to ensure that such a method exists. To statically require the existence of the `length` method, I add the declaration

```
@protocol length(a::AbstractArray)::Int
```

which requires that there is a `size` function for every `AbstractArray` that returns an integer. When I add this to our example, the protocol definition now statically produces the error

```
Protocol length not satisfied; missing implementation(s) for:
    (::Range)::Int
```

identifying that there does not exist a method `length(::Range)::Int`. If I fix this by adding

```
length(r::Range) = size(r)
```

to our definitions, then the protocol definition error vanishes.

The typechecker was able to determine that there was an implementation of `size` available for every `AbstractArray` and could infer that their return types adhered to the protocol specification. Now that we have a protocol for `size` our typed implementation of `array_like` passes successfully.

I am then left with the following Typed Julia program, showing the two key additions.

```
abstract type AbstractArray end
struct List <: AbstractArray
    array::Array{Int, 1}
end
struct Range <: AbstractArray
    start::Int
    stop::Int
end
length(v::List) = length(v.array)
size(u::Range) = u.stop - u.start
length(r::Range) = size(r)
@protocol length(a::AbstractArray)::Int

@typed array_like(a::AbstractArray) = Array(length(a))
```

This example has demonstrated both how Julia code can be gradually type checked (with a typed function calling untyped methods) and how I can canonize abstractions within the protocol system. Now, let us examine how protocols work in more detail and some of the design decisions that went into them.

PROTOCOLS Considering `AbstractArray` again, the `size` protocol is both real and not alone. Julia uses `AbstractArray` for all "array-like" things and has a suitably set of protocols that all implementations must support. Every scalar `AbstractArray` implementation (that I will refer to as `A`) is required by the documentation to implement two methods, as shown in table 1.

size(A)	Returns a tuple containing the dimensions of A
getindex(A, :: Int)	Linear scalar indexing

Table 1: `AbstractArray` interface

No single implementation of all of these methods exists for all `AbstractArray`s; each subtype of `AbstractArray` is expected to implement its own version. The need for this is very straightforward: we cannot possibly write an implementation of `getindex` that works the same way for both an sequential array implementation and a linked list, for example.

Protocols of this sort in Julia are very common in libraries. For example, MathOptInterface, an abstraction layer over numerous numerical optimization solvers, exposes more than a hundred different protocols. One example is `optimize!(dest::AbstractOptimizer, src::ModelLike)`, which uses the optimizer `dest` to optimize the model `src`. There must be an implementation of `optimize!` for *every* `AbstractOptimizer` and `ModelLike`. Implementations of `optimize!` tend to specialize on `dest` while being generic over `src` while using protocols exposed by `ModelLike` to interact with the problem being optimized.

Many protocols are implemented outside of the original package. As evidenced in Fig. 17a packages frequently implement functions from other methods for their own types. Fig. 17b shows that math and collection types are the most commonly implemented. Protocol users are also common: Fig. 19 suggests shows that many call sites dispatch to more than one implementation, demonstrating that reliance on a protocol-describable abstraction is common in Julia programs.

In spite of this adoption and wide import, protocols in Julia are ill-defined. Most exist solely in the form of English-language documentation or even just implicitly in the code itself; no machine-legible form is available. While the types and the implementations should exist in the programs themselves, there is no trace of the abstract notion of the protocol.

Introducing mechanically-checked protocols begets a key question: how do we declare and discover them? Two methods present themselves: if I wanted to support as much existing code as possible, I could try to find protocols from the bottom up, identifying protocols from usages. In contrast, I could also require protocols to be explicitly declared. As shown before, I chose to use explicitly-declared protocols at the expense of being able to easily typed existing code but the alternative of bottom-up protocols deserves closer attention.

Bottom-up checking was the original approach I took to identifying protocols [23]. If I assume that all protocol *implementations* in a program are correct (and tractable to the incomplete static analyzer) then this is an ideal solution: no new type annotations are needed and no new specifications are required; protocols effectively arise implicitly from their use in the code.

```
abstract type Number end
struct Int <: Number end

f(::Int) = 2
g(y::Number) = f(y)
```

Consider the above example. I have two types (`Number` and `Int`) and two methods; `f` can handle only `Int`s, while `g` is supposed to handle any kind of `Number`. Since there is an implementation of `f` for every kind of number, it follows that this implementation is so far type-safe.

```
abstract type Number end
struct Int <: Number end
struct Float32 <: Number end

f(::Int) = 2
g(y::Number) = f(y)
```

If I then add a new implementation of `Number`, `Float32`, then the invocation now becomes not-type-safe. Superficially, this feels fine: there is no implementation of `f` to use from `g`, so I error there. However, consider the evolutionary process I took to get here: we first expected there to be a `f` for every kind of `Number` and there was, so all was good. The modification I made to break the program was to add a new type. I did not touch either `f` or `g`, so breaking `f` for a new definition potentially far away from it feels unfair. In effect, I treat the definition as canonical and uses cases as faulty, even when the desired semantics is the other way around.

Two further issues arise from the bottom-up approach:

- Use-site checking does not establish if an unused implementation is correct. Suppose for a second that I never called `f` with an abstract `Number` in a typed position; I would never know that I had violated the `f` protocol when I added `Float32` and would only realize that post facto. Moreover, there is no explicit declaration that the protocol exists or must be adhered to; it would be easy for a programmer new to the project to miss, misuse, or fail to implement some protocol.

- The design lends itself to "spooky action at a distance." As I saw earlier, when I broke the `f` protocol with `Float32` the error occurred when we called `f` from within `g`, implying that the problem lies there. Errors should point at the actual cause of the failure, rather than merely where the program might go wrong as a result. I should ideally point at the new definition `Float32` as the cause of the failure.

This problem becomes particularly stark with an eye towards the reality that most existing Julia protocols are violated *somewhere*. Even `getindex` has some nonconformant implementations: `LogicalIndex`, used to represent an array of indices masked by a boolean value, has no implementation of `getindex`. As a result, many "obvious" function calls suddenly become use-site errors in spite of the true fault lying with the implementation.

EVAL Using `eval` Julia programmers can insert new definitions of methods and types at any point in the program. Let us consider a small example.

```
abstract type A end
struct B <: A end
f(::A) = 2
g(a::A) = f(a)

function main()::Int
    g(B())
end
main()
```

It should be possible to type-check this program; the return type of both `f` and `g` should be `Int`, trivially.

What should happen, though, if I rewrote the program as

```
abstract type A end
struct B <: A end
f(::A) = 2
g(a::A) = f(a)

function main()::Int
    eval(:(f(::B) = "hello world"))
    g(B())
end
main()
```

There is nothing *semantically* wrong with this program—but what should the return type of `g(B())` now be? The type checker cannot reasonably analyze all `eval`ed expressions, so I would most likely determine that `main` is type-safe. However, if the invocation of `g` ends up calling our new most specific method `f(::B)` then the returned value will be a `String`, not an `Int`.

A type checker written against this "naive" dynamics would not be particularly useful; while it could guarantee that a method *exists* (since while Julia does allow definitions to be deleted it is so uncommon as to be easily prohibited dynamically) the type checker needs to know about *every* potential method that could be called in order to determine what their return types might be. A type checker with an open-world assumption can only say that the return type of any method call is `Any`, which is not especially useful. Therefore, a "closed-world" assumption is practically mandatory.

Luckily for us, Julia has also ran into this issue internally. I talked earlier about how Julia tries to one-shot compile every method call into a statically typed version. This process would be broken just as badly by dynamically added methods as the type checker would be. Consequently, Julia "fixes" the set of visible methods to those that were defined the last time the code reached the top level (or was invoked with the `invokelatest` function). Thus, with the right selection of methods I can make a closed-world assumption that pans out in reality.

Julia's semantics are illustrated with a small modification of the above example. If we add a second call to `main` like the following:

```
abstract type A end
struct B <: A end
f(::A) = 2
g(a::A) = f(a)

function main()::Int
    eval(:(f(::B) = "hello world"))
    g(B())
end
main()
main() # breaks the return type
```

then execution returns to the top level and allows `main` to see the newly-added definition. The second invocation of `main` now returns the string `"hello world"` and violates the return type annotation. Dynamic guards on return types are therefore needed once the set of visible methods has evolved past what I originally checked against, but no earlier.

Type checking for Julia therefore is something of a "hybrid" between a traditional gradual type system (which needs to insert casts to ensure type safety) and that for a statically typed language (which does not). As long as the program's "world" matches the one that was originally checked against no casts are needed. However, if the world "moves on" from that state through the insertion of new definitions via `eval` then the return type becomes unsound and needs dynamic checking.

If I contextualize the type system for Julia within the taxonomic framework that I described with the KafKa language [24] then it is a hybrid of a fully-static type checker, the concrete, and the transient semantics. If the program is running in the original world age that was used for type checking, then no checks whatsoever are needed. In effect, Typed Julia programs in the original world age have a simple soundness guarantee, as they would in a fully-typed language. If the program has moved on from that world age then Typed Julia must check return types as the transient semantics does, but does so using the concrete notion of type membership where tags are checked rather than the superficial identity of the value.

AMBIGUITIES The final challenge that I need to consider is the question of ambiguities. As a simple example, consider two versions of the + function:

```
+(::Int, ::Number) = ...
+(::Number, ::Int) = ...
```

Now, suppose that I call `+(1, 2)`. `1` is an `Int` and `2` is a `Number`, so I could apply the first, but `1` is also a `Number` while `2` is a `Int` so the second could apply as well. Additionally, the two definitions are not related by subtyping, so neither can be called more specific than the other. Thus, no singular most specific implementation exists, and Julia errors at `+(1, 2)`. Should I statically call out either these definitions of + as ones that could be ambiguous or should I treat the invocation as an error if an ambiguity is possible?

If I look at the prior work many typed languages with multiple dispatch identified ambiguities. Eliminating the potential for ambiguities was a major goal and challenge for both Fortress and Cecil [4, 64, 50]. Julia's experience suggests, however, that ambiguity checking may actually be undesirable, particularly at definition sites.

Julia, at one point, included a definition-time ambiguity detection heuristic. However, unlike Fortress and Cecil, Julia decided to *remove* their definition-time ambiguity heuristic [2]. From the perspective of static typing, this decision seems unusual; why suffer runtime errors when they could have been caught statically?

One of Julia's selling points has been the "unreasonable effectiveness" (in their words) of multiple dispatch [3]. When the Julia developers talk about this "effectiveness," they are referring to how multiple dispatch solves the expression problem in a compositional way wherein different libraries can provide the same resuable abstraction for their own structures and thereby be composed.

Let's revisit the + function again. Suppose I implemented a library that had polynomial types. I would like to be able to add a polynomial to a polynomial. Additionally, I want to be able to add simple numbers to a polynomial to get another polynomial with a larger constant. Moreover, because our polynomial acts like a `Number` I subtype `Number`. An (extremely simplified) implementation of this in Julia could look as follows.

```
struct Polynomial <: Number
    values::Tuple
end

Base.+(x::Polynomial, y::Polynomial) = Polynomial(x.values .+ y.values)
Base.+(x::Polynomial, y::Number) = Polynomial((x.values[1:end-1]..., x.values[
    end] + y))
```

I can then use a `Polynomial` as `Polynomial((1,2,3)) + 3` which gives us `Polynomial(1,2,6)`. So far, so good.

The problem is that the "unreasonable effectiveness" of Julia can burn us. Practical Julia programs frequently compose different libraries with one another, therein creating ambiguities. Suppose that there's then an automatic differentiation library that defines a dual number and an addition method.

```
struct Dual <: Number
    value
    epsilon
end
Base.+(x::Number, y::Dual) = Dual(y.value+x, y.epsilon)
```

2 https://github.com/JuliaLang/julia/pull/16125
3 https://www.juliaopt.org/meetings/santiago2019/slides/stefan_karpinski.pdf

If I then imported both `Polynomial` and `Dual` into a third project and then try to invoke `Polynomial(1,2,3) + Dual(0, 0)`, I get an ambiguity. This function call is trivially ambiguous: I cannot decide whether to call the implementation for `Polynomial` or `Dual`. Therefore a sound static ambiguity checker should reject these definitions. So far, so good.

The problem with this answer is entirely practical. It is common for libraries to add special implementations of shared functionality for their own specific use cases and for those implementations to be potentially ambiguous with methods from other packages [4]. Resolving these ambiguities requires adding a suitable more-specific method for *every combination of types*. Therefore, disambiguation methods quadratic in the number of libraries need to be added, requiring both:

- a hilarious number of additional implementations,
- and perfect awareness of all other extensions of the same function in all other libraries.

These requirements are impractical and led to the removal of ambiguity checking as a default in Julia (though it still exists for use in test cases) [5]. Julia's experience was that most ambiguity detection were false, and did not eventuate in actual executions.

Arguably, this realization that ambiguity checking at definition time is impractical is a consequence of Julia realizing multiple dispatch at scale. Ambiguity checking makes sense in the context of a single library or project where a single team of developers controls the entire system. In the aforementioned cases of both Cecil and Fortress the largest programs written in the language by an overwhelming margin were their respective compilers. In these programs ambiguities were clear bugs and the responsibility of only a single team. In contrast, in Julia's much larger ecosystem of loosely-interacting developers the very compositionality of multiple dispatch makes the potential for ambiguities more common and makes it harder to resolve ambiguities organizationally.

The final reason for not performing ambiguity checking in Julia is that actual ambiguous calls are *rather uncommon in practice*: most of the time only one library is

4 https://github.com/JuliaStats/DataArrays.jl/issues/51, https://github.com/JuliaStats/DataArrays.jl/issues/77
5 https://github.com/JuliaLang/julia/issues/6190

being used at a time or composition is relatively simple, therein avoiding the ambiguity. When an ambiguity error is encountered it is usually a sign of bad library design or can be resolved easily by the user adding a suitable disambiguation method themselves—therein addressing the "awareness of all other implementations" issue previously mentioned.

As a result of this experience I do not include static ambiguity checking as a goal for static typing in Julia; method calls in statically-typed Julia may still fail dynamically with an "ambiguous method call" error.

I will now present the type system for Julia in two major parts:

- First, a theoretical system that types the JULIETTE [9] calculus and introduces the key operations and metafunctions.
- Second, an implementation that allows programmers to utilize the proposed system and allows it to be applied to various use cases.

4.1 RELATED WORK

Many of the challenges inherent to typing Julia are not unique. Where does Typed Julia fall in the broader landscape of gradual typing and how does it relate to the usual properties of a gradual type system? Additionally, the combination of static typing and dynamic metaprogramming is not new, nor is typing for multiple dispatch. Let us consider how the prior work in this space bit off these problems.

4.1.1 *Gradual Typing*

Gradual type systems aim to allow the incremental addition of types to untyped code. Siek et al [73], for example, claims five criteria for gradual typing:

1. Equivalency to normal static typing for fully typed terms;
2. Equivalency to dynamic typing for fully untyped terms;
3. Type soundness;
4. Statically-typed code will not be blamed for type errors;
5. Any set of type annotations can be removed from a partially typed program without changing program behaviour (the gradual guarantee).

Criteria 1 and 2 are uncontroversial. Criteria 3, 4, and 5, however, are somewhat trickier.

An example of the disagreement comes from Siek's homepage, wherein he states that

> I've been fortunate to see some of my ideas get used in the software industry:
>
> - Microsoft created a gradually-typed dialect of JavaScript, called TypeScript.
> - Facebook has added gradual typing to PHP. [...]

Neither TypeScript nor Hack (Facebook's PHP type system) satisfy criteria 3 or 4. Both implement so-called *optional* type systems, wherein they erase type annotations after static checking. As a result, they are *unsound* (untyped code may violate typed assumptions at any time) and *have no concept of blame*. As a result, I would conclude that TypeScript and Hack are not gradually typed, contradicting Siek's own description. This internal disagreement about what it means to be a gradual type system is reflective of a broader lack of consensus around the term.

Type soundness alone is a point of much research. Despite being foundational to the concept of static typing, when untyped code is introduced At least 5 distinct concepts have been proposed, each providing different theoretical and practical tradeoffs. Options for how to define soundness range from "nothing" (as in the optional approach) to "type inhabitants must carry type tags that are a subtype of the statically declared type" (the concrete approach, used in languages like C#).

This then plays into the discussion about blame. The concept of blame is to redirect errors created by untyped code that manifest in typed code back into the untyped source. For example, if an untyped function returns an ill-typed value to a typed caller the untyped function should be blamed. However, blame inherently depends on what soundness guarantee the system provides, as that dictates where and when errors will occur. For example, blame makes little sense in a concrete setting (as only statically checked behaviors are allowed to pass type boundaries and typed mutable references are checked on write), but is vital for other semantics.

Finally, I have the gradual guarantee. The idea behind the gradual guarantee is that it captures the migration process, wherein an untyped program is incrementally typed while remaining observationally identical. It expresses this process in the opposite direction: with a type system that satisfies the gradual guarantee I can always remove types while retaining semantic equivalence. Thus, if I have an untyped program that only needs annotations to be typed then a gradual guarantee-compliant type system would let us add the annotations in any order and anywhere I so chose.

The problem is that in practice few untyped programs are actually typable without modification. The TypeScript documentation, for example, explicitly discusses common modifications needed to Javascript programs for them to be typable; Takikawa et al's [76] benchmark suite required numerous code modifications beyond simple annotation insertion in order to satisfy the type checker. Few programmers write perfectly typable untyped code without the aid of a type checker. The programs that the gradual guarantee applies to are, in reality, those that have *already* been typed—not those that have yet to be typed.

Julia makes answering this question easy. Julia's existing type checks work by comparing the runtime type tag associated with values to the type annotations applied to methods, and the guarantees about dispatch follow from such. Thus, the concrete semantics is a very straightforward choice for Julia. In turn, this choice means that blame is immaterial and that (without considerable runtime modifications, as in [61]) I can not satisfy the gradual guarantee. However, as Julia argument type annotations are already used for method dispatch and cannot be removed without modifying the program's semantics, no Julia type system could realistically satisfy the gradual guarantee. Therefore, while the proposed type system for Julia is not gradual by some definitions of the term, it allows for mixing typed and untyped code to interoperate while closely adhering to existing programmer expectations about type annotations.

The concrete semantics alone are not sufficient for Julia, however, due to the challenges posed by multiple dispatch as mentioned earlier. The intersection of multiple dispatch and gradual typing is a relatively unexplored domain. The primary realized example is for the Dylan language [59]. Gradual typing in Dylan is fundamentally different than what I describe here, though; Mehnert's approach is built around a nonlocal type inference algorithm that tries to build out a set of constraints begotten by a realized program and solve them. On one hand, this dramatically simplifies several problems, such as the need for protocols or dealing with underspecified argument types. At the same time, however, it provides many fewer developer-facing benefits of typing (such as improved documentation and robustness to changes) compared to the local system that I present here.

On the vein of nonlocal inference another topic that deserves mention is soft typing. Soft type systems aim to automatically infer types for untyped programs using type inference [31]; the dream was that one could provide a completely untyped program and it would be inferred to a fully typed one for performance. Soft typing has similar problems to the gradual guarantee, however: real untyped programs are rarely written in every detail so that they would type check if you just worked hard enough. Soft typing, in particular, has issues with untypable operations: soft type system rely on nonlocal unificiation-based inference algorithms that may take some time to conclude that something has gone wrong. Whether the unification process provides an error that makes it clear where that "wrong" was depends on the structure of the program. Practically, then, when writing code for soft typing one must take as much care as they would if they were writing code for a traditional gradual type system with static type annotations but while suffering much worse error messages.

4.1.2 *Eval*

Most programming languages control *where* definitions are visible, as part of their scoping mechanisms. Controlling *when* function definitions become visible is less common. Languages with an interactive development environment had to deal with the addition of new definitions for functions from the start [58]. Originally, these languages were interpreted. In that setting, allowing new functions to become visible immediately was both easy to implement and did not incur any performance overhead.

Just-in-time compilation changed the performance landscape, allowing dynamic languages to have competitive performance. However, this meant that to generate efficient code, compilers had to commit to particular versions of functions. If any function is redefined, all code that depends on that function must be recompiled; furthermore, any function currently executing has to be deoptimized using mechanisms such as on-stack-replacement [42]. The drawback of deoptimization is that it makes the compiler more complex and hinders some optimizations. For example, a special `assume` instruction is introduced as a barrier to optimizations by [32], who formalized the speculation and deoptimization happening in a model compiler.

Java allows for dynamic loading of new classes and provides sophisticated controls for where those classes are visible. This is done by the class-loading framework that is part of the virtual machine [48]. Much research happened in that context to allow the Java compiler to optimize code in the presence of dynamic loading.

Detlefs [29] describe a technique, which they call preexistence, that can devirtualize a method call when the receiver object predates the introduction of a new class. Further research looked at performing dependency analysis to identify which methods are affected by the newly added definitions, to be then recompiled on demand [62]. Glew [37] describes a type-safe means of inlining and devirtualization: when newly loaded code is reachable from previously optimized code, these optimizations must be rechecked.

Controlling *when* definitions take effect is important in dynamic software updating, where running systems are updated with new code [26]. Hicks [75] introduce a calculus for reasoning about representation-consistent dynamic software updating in C-like languages. One of the key elements for their result is the presence of an `update` instruction that specifies when an update is allowed to happen. This has similarities to the world-age mechanism described here.

Substantial amounts of effort have been put into building calculi that support `eval` and similar constructs. For example, Politz [67] described the ECMAScript 5.1 semantics for `eval`, among other features. Glew [36] formalized dynamic class loading in the framework of Featherweight Java, and Matthews [57] developed a calculus for `eval` in Scheme. These works formalize the semantics of dynamically modifiable code in their respective languages, but, unlike Julia, the languages formalized do not have features explicitly designed to support efficient implementation.

Julia's use of the world-age mechanism, the method tables that I mentioned earlier, allows Julia to "lock down" what methods might be visible at any point in time. In this manner, Julia dramatically simplified the implementation of their JIT compiler.

4.1.3 *Static Typing of Multiple Dispatch*

Static type systems aim to identify and rule out classes of dynamic error. In a multiple dispatch context, this entails identifying code wherein one of the two aforementioned errors, no applicable methods and ambiguous method call, could occur. Practically, static type systems also enable code completions and facilitate automatic refactoring.

Static typing for multiple dispatch is an old idea, with an early comprehensive concept put forward by [2], which describes a type system able to eliminate both no method found and ambiguous method errors. Agrawal focuses on ambiguous method errors, for as in comparison, it is easier to identify cases were no method exists versus when multiple ambiguous methods apply. They describe an algorithm designed to statically identify cases where an ambiguous invocation may occur and how likely these cases are under different language semantics. Notably, however, they focus primarily on systems in the vein of CLOS, which add declaration order as a means of additional disambiguation beyond subtyping; as a result, they are able to frequently reject ambiguities in cases where Julia would be ambiguous.

The Cecil language [16] is the statically typed language with the best analogy to Julia. Cecil features the same external (not associated with any one object) methods and a similar polymorphic type language to Julia's. As a result, its type system can serve as a point of reference for the design of a type system for Julia.

Cecil's type system went through several iterations from the earliest versions described in passing in [16], further expounded upon to a relatively comprehensive cover of the language in [18], and finally extended to support constraint based polymorphism [51]. The project aimed to be evolved into the Diesel language (which simplified the object model and implemented a module system), but no publications were forthcoming.

The most relevant work for type checking in Julia is [18], which describes the core of Cecil's type system. Typechecking in Cecil is broken into two components: *implementation* and *client*.

Implementation-side checking in Cecil is the core of the approach. The issue arises from how Cecil addresses one of Julia's key correctness issues: signatures. In Cecil, inter-library behavioral specifications can be written as a list of methods that are required to work for all possible type instantiations. For example, I could specify addition as

```
type num;
type int subtypes num;
type fraction subtypes num;
signature +(num, num): num;
signature +(int, int): int;
signature +(fraction, fraction): fraction;
```

Here, I say that it must be possible to add any two numbers, regardless of their types, producing an arbitrary number. Similarly, I also specify that adding two ints must produce an int and the same for fractions. Cecil statically guarantees that if I added, say, `type irrational subtypes num` that it must be possible to add an `int` and a `irrational` to get a subtype of `num`.

Ensuring correctness against a set of signature declarations in Cecil requires that implementations satisfy three properties: conformance, completeness, and consistency. Conformance requires that the argument and result types for each method implementing the signature must be compatible with the types specified by the signature; for example, our implementation of `+` for `irrational` cannot return a `string`. Completeness enforces that potential signature instantiations must implement the signature; I must implement `+` for `irrational` because `irrational` is a `num`. Between them, conformance and completeness rule out message not understood errors. Finally, consistency requires that no ambiguities may exist among different implementations of the same signature; I cannot implement `+(fraction, num)` and `+(num, fraction)` without `+(fraction, fraction)`, for otherwise the latter case would be ambiguous.

Cecil statically requires that all implementations satisfy these three properties. As a result, client-side checking in Cecil is very simple: as long as the static type system can guarantee that either some concrete implementation exists or that some signature is employed, then the call can be considered safe.

Neither Cecil nor Fortress considered the question of "what happens with dynamically-generated code?" Both systems were, effectively, research projects that had few external users and were designed from the start to support static typing. Consequently, `eval` and similar dynamic metaprogramming was not a major concern.

4.2 A CORE CALCULUS FOR JULIA

I formalize my type system for Julia using the JULIETTE calculus. JULIETTE was originally used in our paper formalizing world age in Julia [9]. The calculus focuses on capturing how method invocation in Julia works with an eye towards what it means to add new methods and when they can be called. I will first describe the basic JULIETTE calculus, define the static semantics for typed JULIETTE, then consider the dynamic semantics of typed JULIETTE alongside the two key correctness properties.

JULIETTE uses *method tables* to represent sets of methods available for dispatch. The *global table* is the method table that records all definitions and always reflects the "true age" of the world; the global table is part of JULIETTE program state. *Local tables* are method tables used to resolve method dispatch during execution and may lag behind the global table when new functions are introduced. Local tables are then baked into program syntax to make them explicit during execution. As in Julia, JULIETTE separates method tables (which represent code) from data: as mentioned in Chapter 2, the world-age semantics only applies to code. As global variables interact with `eval` in the standard way, I omit them from the calculus.

The treatment of methods is similar in both JULIETTE and Julia up to (lexically) local method definitions. In both systems, a generic function is defined by the set of methods with the same name. In Julia, local methods are syntactic sugar for global methods with fresh names. For simplicity, I do not model this aspect of Julia: JULIETTE methods are always added to the global method table. All function calls are resolved using the set of methods found in the current local table. A function value `m` denotes the name of a function and is not itself a method definition. Then, since JULIETTE omits global variables, its global environment is entirely captured by the global method table.

Although in Julia `eval` incorporates two features—top-level evaluation and quotation[6]—only top-level evaluation is relevant to world age, and this is what I model in JULIETTE. Instead of an `eval` construct, the calculus has operations for evaluating expressions in different method-table contexts. In particular, JULIETTE offers a *global evaluation construct* $(\!|\, e \,|\!)$ (pronounced "banana brackets") that accesses the most recent set of methods. This is equivalent to `eval`'s behavior, which evaluates in the latest world age. Since JULIETTE does not have global variables, $(\!|\, e \,|\!)$ reads from the local environment directly instead of using quotation.

Every function call $\mathtt{m}(\overline{\mathtt{v}})$ in JULIETTE gets resolved in the closest enclosing local method table M by using an *evaluation-in-a-table* construct $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{M}$. Any top-level function call first takes a snippet of the current global table and then evaluates the call in that *frozen* snippet. That is, $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)$ steps to $(\!|\, \mathtt{m}(\overline{\mathtt{v}}) \,|\!)_{M}$ where M is the current global table. Thus, once a snippet of the global table becomes local table, all function calls that ensue from the body of of $\mathtt{m}(\overline{\mathtt{v}})$ will be resolved using this table, reflecting the fact that a currently executing top-level function call does not see updates to the global table.

JULIETTE is parameterized over values, types, type annotations, a subtyping relation, and primitive operations. Only minimal assumptions are needed for these primitives.

6 Represented with the `$` operator in Julia, as in `eval(:(g() = $x))` in Fig. 10.

e	$::=$		*Expression*
	$\mid$	$\mathtt{v}$	value
	$\mid$	x	variable
	$\mid$	$e_1 \,;\, e_2$	sequencing
	$\mid$	$\delta_l(\overline{e})$	primop call
	$\mid$	$e(\overline{e})$	function call
	$\mid$	$\mathtt{md}$	method definition
	$\mid$	$\llparenthesis\, e \,\rrparenthesis$	global evaluation
	$\mid$	$\llparenthesis\, e \,\rrparenthesis_M$	evaluation in a table
$\mathtt{p}$	$::=$	$\llparenthesis\, e \,\rrparenthesis$	*Program*
$\mathtt{md}$	$::=$	$\triangleleft\, \mathtt{m}(\overline{x :: t}) = e \,\triangleright$	*Method definition*

$\mathtt{v}$	$:=$	$\ldots$	*Value*
	$\mid$	$\mathtt{unit}$	unit value
	$\mid$	$\mathtt{m}$	generic function
σ	$:=$	$\ldots$	*Type tag*
	$\mid$	$\mathbb{1}$	unit type
	$\mid$	$\mathtt{m}$	type tag of function $\mathtt{m}$
t	$:=$	$\ldots$	*Type annotation*
	$\mid$	$\top$	top type

Figure 26: Surface syntax

4.2.1 *Syntax*

The surface syntax of JULIETTE is given in Fig. 26. It includes method definitions $\mathtt{md}$, function calls $e(\overline{e})$, sequencing $e_1 \,;\, e_2$, global evaluation $\llparenthesis\, e \,\rrparenthesis$, evaluation in a table $\llparenthesis\, e \,\rrparenthesis_M$, variables x, values $\mathtt{v}$, primitive calls $\delta_l(\overline{e})$, type tags σ, and type annotations t. Values $\mathtt{v}$ include `unit` (unit value, called `nothing` in Julia) and $\mathtt{m}$ (generic function value [7]). Primitive operators δ_l represent built-in functions such as `Core.Intrinsics.mul_int`. Type tags σ include $\mathbb{1}$ (unit type, called `Nothing` in Julia) and $\mathtt{m}$ (tag of function value $\mathtt{m}$). Type annotations t include $\top \in t$ ($\top$ is the top type, called `Any` in Julia) and $\sigma \subseteq t$ (all type tags serve as valid type annotations).

4.2.2 *Semantics*

The internal syntax of JULIETTE is given in the top of Fig. 27. It includes evaluation result $\mathtt{r}$ (either value or error), method table M, and two evaluation contexts, $\mathtt{X}$ and

7 Distinguished function values are a simplification begotten by the calculus; Julia allows functions to be called on any receiver, not just special method ones as in JULIETTE.

$\mathtt{C}$, which are used to define small-step operational semantics of JULIETTE. Evaluation contexts $\mathtt{X}$ are responsible for simple sequencing, such as the order of argument evaluation; these contexts never contain global/table evaluation expressions $(\!|\cdot|\!)$ and $(\!|\cdot|\!)_{\mathrm{M}}$. World evaluation contexts $\mathtt{C}$, on the other hand, capture the full grammar of expressions.

Program state is a pair $\langle \mathrm{M}, \mathtt{C}\,[e]\rangle$ of a global method table M and an expression $\mathtt{C}\,[e]$.

Global method table

$$\langle \boxed{\mathrm{M}}, \boxed{\mathtt{C}}\,[e]\rangle \rightarrow \langle \boxed{\mathrm{M}'}, \boxed{\mathtt{C}}\,[e']\rangle$$

World evaluation context

I define the semantics of the calculus using two judgments: a normal small-step evaluation denoted by $\langle \mathrm{M}, \mathtt{C}\,[e]\rangle \rightarrow \langle \mathrm{M}', \mathtt{C}\,[e']\rangle$, and a step to an error $\mathrm{M} \vdash \mathtt{C}\,[e] \rightarrow \mathtt{error}$. The $\mathbf{typeof}(\mathtt{v}) \in \sigma$ operator returns the tag of a value. I require that $\mathbf{typeof}(\mathtt{unit}) = \mathbb{1}$ and $\mathbf{typeof}(\mathtt{m}) = {}_{\mathtt{m}}$. I write $\mathbf{typeof}(\overline{\mathtt{v}})$ as a shorthand for $\overline{\mathbf{typeof}(\mathtt{v})}$. Function $\Delta(l, \overline{\mathtt{v}}) \in \mathtt{r}$ computes primop calls, and function $\Psi(l, \overline{\sigma}) \in \sigma$ indicates the tag of l's return value when called with arguments of types $\overline{\sigma}$. These functions have to agree, i.e. $\forall \overline{\mathtt{v}}, \overline{\sigma}.(\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \wedge \Delta(l, \overline{\mathtt{v}}) = \mathtt{v}' \implies \mathbf{typeof}(\mathtt{v}') = \Psi(l, \overline{\sigma}))$. The subtyping relation $t_1 <: t_2$ is used for multiple dispatch. I require that subtyping is transitive so if $t_1 <: t_2$ and $t_2 <: t_3$ then $t_1 <: t_3$; transitivity is needed by the type system for subsumption.

NORMAL EVALUATION These rules capture successful program executions.

Rule E-SEQ is completely standard: it throws away the evaluated part of a sequencing expression. Rules E-VALGLOBAL and E-VALLOCAL pass value $\mathtt{v}$ to the outer context. This is similar to Julia where `eval` returns the result of evaluating the argument to its caller. Rule E-MD is responsible for updating the global table: a method definition `md` will extend the current global table M into $\mathrm{M} \bullet \mathtt{md}$, and itself evaluate to `m`, which is a function value. Note that E-MD only extends the method table and leaves existing definitions in place. If the table contains multiple definitions of a method with the same signature, it is then the dispatcher's responsibility to select the right method; this mechanism is described below in more detail.

The two call forms E-CALLGLOBAL and E-CALLLOCAL form the core of the calculus. The rule E-CALLGLOBAL describes the case where a method is called directly from a global evaluation expression. In Julia, this means either a top-level call, an `invokelatest` call, or a call within `eval` such as `eval(:(g(...)))`. The "direct" part is encoded

$\mathtt{r}$	::=		*Result*
	|	$\mathtt{v}$	value
	|	$\mathtt{error}$	error
M	::=		*Method table*
	|	$\varnothing$	empty table
	|	$M \bullet \mathtt{md}$	table extension

$\mathtt{X}$	::=		*Simple evaluation context*
	|	$\square$	hole
	|	$\mathtt{X}\,;\,e$	sequence
	|	$\delta_l(\overline{\mathtt{v}}\ \mathtt{X}\ \overline{e})$	primop call (argument)
	|	$\mathtt{X}(\overline{e})$	function call (callee)
	|	$\mathtt{v}(\overline{\mathtt{v}}\ \mathtt{X}\ \overline{e})$	function call (argument)
$\mathtt{C}$	::=		*World evaluation context*
	|	$\mathtt{X}$	simple context
	|	$\mathtt{X}\,[(\!|\, \mathtt{C}\,|\!)]$	global evaluation
	|	$\mathtt{X}\,[(\!|\, \mathtt{C}\,|\!)_M]$	evaluation in a table M

E-Seq
$$\frac{}{\langle M, \mathtt{C}\,[\mathtt{v}\,;\,e]\rangle \;\to\; \langle M, \mathtt{C}\,[e]\rangle}$$

E-Primop
$$\frac{\Delta(l, \overline{\mathtt{v}}) = \mathtt{v}'}{\langle M, \mathtt{C}\,[\delta_l(\overline{\mathtt{v}})]\rangle \;\to\; \langle M, \mathtt{C}\,[\mathtt{v}']\rangle}$$

E-MD
$$\frac{\mathtt{md} \equiv \triangleleft\, \mathtt{m}(\overline{x :: t}) = e \,\triangleright}{\langle M, \mathtt{C}\,[\mathtt{md}]\rangle \;\to\; \langle M \bullet \mathtt{md}, \mathtt{C}\,[\mathtt{m}]\rangle}$$

E-CallGlobal
$$\frac{}{\langle M, \mathtt{C}\,[(\!|\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,|\!)]\rangle \;\to\; \langle M, \mathtt{C}\,[(\!|\, \mathtt{X}\,[(\!|\, \mathtt{m}(\overline{\mathtt{v}})\,|\!)_M]\,|\!)]\rangle}$$

Copy global table to local

E-CallLocal

Use local method table
$$\frac{\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathbf{getmd}(M', \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: t}) = e \,\triangleright}{\langle M, \mathtt{C}\,[(\!|\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,|\!)_{M'}]\rangle \;\to\; \langle M, \mathtt{C}\,[(\!|\, \mathtt{X}\,[e[\overline{x} \mapsto \overline{\mathtt{v}}]]\,|\!)_{M'}]\rangle}$$

Ignore global table

E-ValGlobal
$$\frac{}{\langle M, \mathtt{C}\,[(\!|\, \mathtt{v}\,|\!)]\rangle \;\to\; \langle M, \mathtt{C}\,[\mathtt{v}]\rangle}$$

E-ValLocal
$$\frac{}{\langle M, \mathtt{C}\,[(\!|\, \mathtt{v}\,|\!)_{M'}]\rangle \;\to\; \langle M, \mathtt{C}\,[\mathtt{v}]\rangle}$$

E-VarErr
$$\frac{}{M \vdash \mathtt{C}\,[x] \;\to\; \mathtt{error}}$$

E-PrimopErr
$$\frac{\Delta(l, \overline{\mathtt{v}}) = \mathtt{error}}{M \vdash \mathtt{C}\,[\delta_l(\overline{\mathtt{v}})] \;\to\; \mathtt{error}}$$

E-CalleeErr
$$\frac{\mathtt{v}_c \neq \mathtt{m}}{M \vdash \mathtt{C}\,[\mathtt{v}_c(\overline{\mathtt{v}})] \;\to\; \mathtt{error}}$$

E-CallErr
$$\frac{\mathbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \mathbf{getmd}(M', \mathtt{m}, \overline{\sigma}) = \mathtt{error}}{M \vdash \mathtt{C}\,[(\!|\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]\,|\!)_{M'}] \;\to\; \mathtt{error}}$$

$$
\begin{aligned}
\mathbf{getmd}(M, \mathtt{m}, \overline{\sigma}) &= \min(\mathrm{applicable}(\mathbf{latest}(M), \mathtt{m}, \overline{\sigma})) \\[1em]
\mathbf{latest}(M) &= \mathbf{latest}(\emptyset, M) \\
\mathbf{latest}(mds, \varnothing) &= mds \\
\mathbf{latest}(mds, M \bullet \mathtt{md}) &= \mathbf{latest}(mds \cup \mathtt{md}, M) \text{ if } \neg\, \mathrm{contains}(mds, \mathtt{md}) \\
\mathbf{latest}(mds, M \bullet \mathtt{md}) &= \mathbf{latest}(mds, M) \qquad \text{if } \mathrm{contains}(mds, \mathtt{md}) \\[1em]
\mathrm{applicable}(mds, \mathtt{m}, \overline{\sigma}) &= \{\triangleleft\, \mathtt{m}(\overline{x :: t}) = e \,\triangleright \in mds \mid \overline{\sigma} <: \overline{t}\} \\[1em]
\min(mds) &= \triangleleft\, \mathtt{m}(\overline{x :: t}) = e \,\triangleright \in mds \text{ such that } \forall \triangleleft\, \mathtt{m}(\overline{\_ :: t'}) = \_ \,\triangleright \in mds\ .\ \overline{t} <: \overline{t}' \\
\min(mds) &= \mathtt{error} \text{ otherwise} \\
\mathrm{contains}(mds, \mathtt{md}) &= \exists\, \mathtt{md}' \in mds \text{ such that} \\
&\qquad (\mathtt{md} \equiv \triangleleft\, \mathtt{m}(\overline{\_ :: t}) = \_ \,\triangleright) \wedge (\mathtt{md}' \equiv \triangleleft\, \mathtt{m}(\overline{\_ :: t'}) = \_ \,\triangleright) \wedge \overline{t} <: \overline{t}' \wedge \overline{t}' <: \overline{t}
\end{aligned}
$$

Figure 27: Internal Syntax and Semantics

with the use of a simple evaluation context $\mathtt{X}$. In this global-call case, I need to save the current method table into the evaluation context for a subsequent use by E-CallLocal. To do this, I annotate the call $\mathtt{m}(\overline{\mathtt{v}})$ with a copy of the current global method table M, producing $\llparenthesis\, \mathtt{m}(\overline{\mathtt{v}}) \,\rrparenthesis_{M}$.

To perform a local call—or, equivalently, a call after the invocation has been wrapped in an annotation specifying the current global table—E-CallLocal is used. This rule resolves the call according to the tag-based multiple-dispatch semantics in the "deepest" method table M' (the use of $\mathtt{X}$ makes sure there are no method tables between M' and the call). Once an appropriate method has been found, it proceeds as a normal invocation rule would, replacing the method invocation with the substituted-for-arguments method body. Note that the body of the method is still wrapped in the $\llparenthesis\ \rrparenthesis_{M'}$ context. This ensures that nested calls will be resolved in the same table (unless they are more deeply wrapped in a global evaluation $\llparenthesis\ \rrparenthesis$).

An auxiliary meta-function **getmd**$(M, \mathtt{m}, \overline{\sigma})$, which is used to resolve multiple dispatch, is defined in the bottom of Fig. 27. This function returns the most specific method applicable to arguments with type tags $\overline{\sigma}$, or errs if such a method does not exist. If the method table contains multiple equivalent methods, older ones are ignored. For example, for the program

$$\llparenthesis \triangleleft\, \mathtt{g}() = 2 \,\triangleright;\ \triangleleft\, \mathtt{g}() = 42 \,\triangleright;\ \mathtt{g}() \,\rrparenthesis,$$

function call $\mathtt{g}()$ is going to be resolved in the table $(\varnothing \bullet \triangleleft\, \mathtt{g}() = 2 \,\triangleright) \bullet \triangleleft\, \mathtt{g}() = 42 \,\triangleright$, which contains two equivalent methods (I call methods equivalent if they have the same name and their argument type annotations are equivalent with respect to subtyping). In this case, the function **getmd** will return method $\triangleleft\, \mathtt{g}() = 42 \,\triangleright$ because it is the newest method out of the two.

Note that functions can be mutually recursive because of the dynamic nature of function call resolution.

ERROR EVALUATION These rules capture all possible error states of Juliette. Rule E-VarErr covers the case of a free variable, an `UndefVarError` in Julia. E-PrimopErr accounts for errors in primitive operations such as `DivideError`. E-CalleeErr fires when a non-function value is called. Finally, E-CallErr accounts for multiple-dispatch resolution errors, e.g. when the set of applicable methods is empty (no method found), and when there is no best method (ambiguous method).

4.3 STATIC TYPE SYSTEM

The type system for JULIETTE augments the method definition form with a return type $\triangleleft\, \mathtt{m}(\overline{x :: t}) :: \mu = e \triangleright$; these types are either expressed explicitly (in typed code) or are inferred statically (in untyped code, based on the declared argument types). I do not modify the dynamic semantics of JULIETTE for this system. Untyped methods have a return type $\mu = \star$, while typed methods have a return type $\mu = t$.

Typing for JULIETTE works over partially-typed programs of the form $M; P\ e$ consisting of a method table M, a protocol table P, and an executing (untyped) expression e. T-PROG defines the program well-formedness relation, ensuring that all protocol definitions $\overline{\mathtt{pd}}$ in the program are satisfied followed by checking that all method definitions are well-formed against the other definitions and the protocol table. Typed methods are well-formed if their bodies typecheck; untyped methods are always well-formed. Protocol well-formedness is delegated to the **checkproto** metafunction whose definition I will explore later.

The basic equation typing relation types and translates an expression with the relation

$$\overbrace{\Gamma}^{\text{Variable typing context}} \vdash_{M;P} \underbrace{e}_{\text{Expression to check}} \Rightarrow \underbrace{e'}_{\text{Resulting expression}} : \overbrace{t}^{\text{Resulting type}}$$

Expressions are type checked against a typing context Γ that contains the variables in scope (introduced in JULIETTE as arguments to the function containing the current expression). The expression being checked e is then translated into a resulting expression e' while producing a result type t.

The type system addresses each of the five errors as follows:

- E-VARERROR by ensuring that all variables lexically exist.
- E-PRIMOPERROR by checking that the primitive operation is defined for the realized argument types.
- E-CALLEEERR by ensuring that only method-typed variables are valid in invocee position.
- E-CALLERR, with the exception of ambiguities, by ensuring that a suitable implementation always exists be it by there being an implementation for a supertype or there being a suitable protocol.

$$
\begin{array}{llll}
M & ::= & & \textit{Method table} \\
 & \mid & \varnothing & \text{empty table} \\
 & \mid & M \bullet \lhd\, \mathtt{m}(\overline{x :: t}) :: \mu \mathrel{=} e \,\rhd & \text{extension} \\[1em]
P & ::= & & \textit{Protocol table} \\
 & \mid & \varnothing & \text{empty table} \\
 & \mid & P \bullet \lhd\, \mathtt{m}(\overline{:: t}) :: t_r \,\rhd & \text{extension} \\[1em]
\Gamma & ::= & & \textit{Variable typing} \\
 & \mid & \varnothing & \text{empty table} \\
 & \mid & x :: t \,\bullet\, \Gamma & \text{variable typing}
\end{array}
\qquad
\begin{array}{llll}
e & ::= & & \ldots \\
 & \mid & e(\overline{e}) ::_M t & \text{checked function call} \\[1em]
\mathtt{X} & ::= & & \ldots \\
 & \mid & \mathtt{X}(\overline{e}) ::_M t & \text{checked function call (callee)} \\
 & \mid & \mathtt{v}(\overline{\mathtt{v}}\ \mathtt{X}\ \overline{e}) ::_M t & \text{checked function call (argument)} \\[1em]
\mathtt{md} & ::= & \lhd\, \mathtt{m}(\overline{x :: t}) :: \mu \mathrel{=} e \,\rhd & \textit{Method definition}
\end{array}
$$

$$
\textsc{T-Value}\;\frac{\mathbf{typeof}(\mathtt{v}) = \sigma}{\Gamma \vdash_{M;P} \mathtt{v} \Rightarrow \mathtt{v} : \sigma}
\qquad
\textsc{T-Var}\;\frac{x : t \in \Gamma}{\Gamma \vdash_{M;P} x \Rightarrow x : t}
\qquad
\textsc{T-Seq}\;\frac{\Gamma \vdash_{M;P} e_1 \Rightarrow e_1' : t_1 \qquad \Gamma \vdash_{M;P} e_2 \Rightarrow e_2' : t_2}{\Gamma \vdash_{M;P} e_1\,;\, e_2 \Rightarrow e_1'\,;\, e_2' : t_2}
$$

$$
\textsc{T-Sub}\;\frac{\Gamma \vdash_{M;P} e \Rightarrow e' : t \qquad t <: t'}{\Gamma \vdash_{M;P} e \Rightarrow e' : t'}
\qquad
\textsc{T-Primop}\;\frac{\overline{\Gamma \vdash_{M;P} e \Rightarrow e' : t} \qquad \Psi(l, \overline{t}) = t'}{\Gamma \vdash_{M;P} \delta_l(\overline{e}) \Rightarrow \delta_l(\overline{e'}) : t'}
$$

$$
\textsc{T-Call}\;\frac{\Gamma \vdash_{M;P} e_r \Rightarrow e_r' : \mathbf{typeof}(\mathtt{m}) \qquad \overline{\Gamma \vdash_{M;P} e_a \Rightarrow e_a' : t_a} \qquad \mathbf{dispatch}(M, P, \mathtt{m}, \overline{t_a}) = t_r}{\Gamma \vdash_{M;P} e_r(\overline{e_a}) \Rightarrow e_r'(\overline{e_a'}) ::_M t_r : t_r}
$$

Embed return type

$$
\textsc{T-Prog}\;\frac{\overline{\mathtt{md}} \vdash \mathtt{pd}_i\ \forall \mathtt{pd}_i \in \overline{\mathtt{pd}} \qquad \overline{\mathtt{md}};\ \overline{\mathtt{pd}} \vdash \mathtt{md}_i \Rightarrow \mathtt{md}_i\ \forall \mathtt{md}_i \in \overline{\mathtt{md}}}{\vdash \overline{\mathtt{md}}; \overline{\mathtt{pd}}\ e \Rightarrow \overline{\mathtt{md}}'\ e}
$$

$$
\textsc{T-MD-Typed}\;\frac{\overline{x :: t_a} \vdash_{M;P} e \Rightarrow e' : t_r}{M; P \vdash \lhd\, \mathtt{m}(\overline{x :: t_a}) :: t_r \mathrel{=} e \,\rhd \Rightarrow \lhd\, \mathtt{m}(\overline{x :: t_a}) :: t_r \mathrel{=} e' \,\rhd}
\qquad
\textsc{T-MD-Untyped}\;\frac{}{M; P \vdash \lhd\, \mathtt{m}(\overline{x :: t}) :: \star \mathrel{=} e \,\rhd \Rightarrow \lhd\, \mathtt{m}(\overline{x :: t}) :: \star \mathrel{=} e \,\rhd}
$$

$$
\textsc{T-PD}\;\frac{\mathbf{checkproto}(\mathtt{m}, \overline{t}, t_r, M)}{M \vdash \lhd\, \mathtt{m}(\overline{:: t}) :: t_r \,\rhd}
$$

Figure 28: Typed translation for JULIETTE

- E-TypeErr by ensuring that, so long as the dynamically used method table is the same as the statically checked one, the values returned from methods always are of the statically-known type.

E-TypeErr deserves a deeper examination. Casts do not appear in normal Juliette code; they are only inserted by the translation to enforce statically-determined return types. In the prior example of `f` and `g`, `g` expected `f` to always return a `Int` but the method table was extended at runtime with a `f` that returned a `String`. Therefore, I *can* suffer cast failures when a new method is added after the method table the program was typechecked against.

The need to check the real returned value from a method determines the structure of T-Call. T-Call typechecks a method invocation by ensuring that it is statically resolvable using the **dispatch** metafunction. The rule then translates checked calls to include both the statically-determined return type and the method table against which that return type was generated. This method table and return type will be used later to ensure type safety of returned values.

4.3.1 *Static Dispatch Resolution*

The key operator used by the static type system is **dispatch**. Method calls are the key component of Julia's semantics and the **dispatch** metafunction is used in T-Call to determine the two key properties for a function call:

- Will there be a method to invoke?
- What will the return type be?

To motivate our treatment of dispatch, suppose that I have evaluated a method call down to a bare call $\mathtt{m}(\overline{\mathtt{v}})$. At this point in evaluation the called method is known and all arguments are now values; I must now figure out what method could actually be called here. The Juliette calculus handles this using rule E-CallLocal, which works primarily through the $\mathbf{getmd}(\mathrm{M}, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: t_a}) :: \_ = e \triangleright$ function where M is our current method table, $\overline{\sigma} = \overline{\mathbf{typeof}(\mathtt{v})}$ is the vector of value types, and the method definition $\triangleleft\, \mathtt{m}(\overline{x :: t_a}) :: \_ = e \triangleright$ is the resulting implementation found for this value vector. If it cannot find a singular most specific implementation it instead produces `error`.

Now, suppose that instead I have an unevaluated call $e(\overline{e'})$. If we suppose that its arguments are typed $\Gamma \vdash_{\mathrm{M}} e' : t'$ then by soundness all ensuing argument values

$\overline{v}$ are instances of $\overline{t'}$. Therefore, to prove that E-CALLLOCAL can eventually apply it suffices to show that for any value vector $\overline{v}$ with tags $\overline{\sigma} = \overline{\textbf{typeof}(v)}$ such that $\overline{\sigma} <: \overline{t'}$ that I can find some unique most specific method definition in M. The statically-inferred return types are then simply the meet of the return types of the identified methods.

I use the abstract metafunction **dispatch** to describe this operation. I say that $\textbf{dispatch}(M, P, \mathtt{m}, \overline{t}) = t_r$ holds if it can guarantee that calling $\mathtt{m}$ in method table M and with protocol table P with arguments $\overline{t}$ will produce a return type that is a subtype of t_r. Alternatively, $\textbf{dispatch}(M, P, \mathtt{m}, \overline{t}) = \mathtt{error}$ if it cannot provide this guarantee.

SUCCESS The **dispatch** metafunction can succeed in two cases:

- Satisfying method: there is some method whose arguments $\overline{t_a}$ satisfy $\overline{t} <: \overline{t_a}$. In this case resolving **dispatch** is trivial: that implementation ensures that the call will go through to something, even if there are more specific implementations that might also be called.

- Protocol: there is no single method whose arguments are a supertype of the given arguments, but for every concrete type vector $\overline{\sigma} <: \overline{t'}$ there is a method with type $\overline{t_a}$ such that $\overline{\sigma} <: \overline{t_a}$.

The first case is trivial: I know a suitable method exists, so I can simply say "at least that one will be invoked." The second case is addressed by the protocol mechanism and is deferred by **dispatch** to those static declarations. As mentioned previously, I could check for protocol safety as part of **dispatch** but determined that the design consequences were undesirable.

Thus, I propose a solution to **dispatch** success in two parts: the **dispatch** metafunction *itself* only checks for satisfying methods, methods whose declared types are supertypes of the given argument typing. We then pair this with an *protocol checking* mechanism, wherein programmers can define protocols that act as an independent source of truth for both protocol implementations (e.g. every `AbstractArray` must have a `size`) as well as for use sites.

FAILURE A JULIETTE (and Julia) function call can fail in two ways:

- No implementations exist: There are no methods whose signatures are a supertype of the runtime argument type vector. Equivalently, $\exists \overline{v}$ such that

$\overline{\textbf{typeof}(\mathtt{v}) <: \mathrm{t}}$ but where for any method $\triangleleft\ \mathtt{m}(\overline{x :: t_a}) :: t_r = e \triangleright$ there is some i such that $t_i \not<: t_{a,i}$.

- There are multiple *most specific* implementations of m for this case.

The first kind of failure, no implementation exists, is straightforward and is the multidispatch equivalent of "message not understood;" effectively, it is the inverse of the "method found" success case. The second kind is more challenging. Julia dispatches method calls to the *most specific*, *satisfying* implementation, which is represented in JULIETTE with the min and applicable metafunctions. As mentioned earlier, I do not statically protect against ambiguities; method ambiguous errors may occur at any invocation at runtime.

DEFINITION This then brings us to the definition of $\textbf{dispatch}(M, P, \mathtt{m}, \bar{t})$:

$$\frac{\triangleleft\ \mathtt{m}(\overline{:: t_a}) :: t_r \triangleright \in P \qquad \bar{t} <: \overline{t_a}}{\textbf{dispatch}(M, P, \mathtt{m}, \bar{t}) = t_r}$$

$$\frac{\begin{array}{c}\exists \triangleleft\ \mathtt{m}(\overline{x :: t_a}) :: \mu = e \triangleright \in M \wedge \bar{t} <: \overline{t_a} \\ \forall \bar{\sigma} : \bar{\sigma} <: \bar{t},\ \textbf{getmd}(M, \mathtt{m}, \bar{\sigma}) = \triangleleft\ \mathtt{m}(\overline{x :: t'_a}) :: \mu = e \triangleright \implies \overline{x :: t'_a} \vdash_M e : t'_r \wedge t'_r <: t_r\end{array}}{\textbf{dispatch}(M, P, \mathtt{m}, \bar{t}) = t_r}$$

Trivially, if the method is one that I statically checked using a protocol then the dispatch will defer to that protocol. Otherwise, as is handled by the second rule, I need to check if this specific invocation is safe.

I break up non-protocol dispatch checking into two clauses. The first clause ensures that a suitable method always exists, while the second ensures that the return type is fully general for all possible implementations. The **dispatch** function may return either a typed or typed method as most specific; the return type need only be valid for the final result and does not have to reflect a static typing of the method body.

The second clause covers the case when I dispatch to a method more specific than this "sufficiently general" one; it states that any method that can be invoked from the current call site must be *inferrable* to have the correct return type.

Inference in my system is treated as a black box. The judgment takes the form

$$\overbrace{\Gamma}^{\text{Inference environment}} \vdash_{\underbrace{M}_{\text{Inference method table}}} \overbrace{e}^{\text{Expression to infer}} : \underbrace{t}_{\text{Inferred type}} \qquad \text{(Type inference)}$$

For convenience I define the notation $\langle M', e\rangle \to_M \langle M'', e'\rangle$ as $\forall C\, \langle M', C[(\!|\, e \,|\!)_M]\rangle \to \langle M'', C[(\!|\, e' \,|\!)_M]\rangle$; interpret this as "the expression e evaluates to e' with local method table M."

The primary required property of inference is weaker than that for a traditional static type system. Instead of a strong soundness guarantee that rules out errors entirely inference merely states that *if* an expression steps then it will be well-typed. Thus, soundness of inference is stated as follows:

Definition 1 (Soundness of inference)**.** *If* $\vdash_M e : t$ *then either*

- $\exists v:\ e = v$ *and* $\mathbf{typeof}(v) <: t$
- $\forall M'\, \exists M'', \langle M', e\rangle \to_M \langle M'', e'\rangle$ *and* $\vdash_M e' : t$
- $\forall M', M' \vdash (\!|\, e \,|\!)_M \to \mathtt{error}$

For the purposes of typing I additionally require that inference is consistent under substitution:

Definition 2 (Substitution for inference)**.** *If* $\overline{x :: t'} \vdash_M e : t$ *and* $\mathbf{typeof}(\overline{v}) <: \overline{t'}$ *then* $\vdash_M e[\overline{v} \mapsto \overline{x}] : t$

The good news, then, is that **dispatch** itself is easy to implement. I just need to look for a method whose arguments are supertypes of the known arguments. If such a method exists, I meet its (potentially-inferred) return type with the return type of all other possible implementations and I are done. Julia already implements a type inference system with which I can infer return types for untyped methods making this task straightforward.

The remaining problem is how does the protocol checking metafunction **checkproto** work?

4.3.2 *Protocols*

Consider the earlier example of `Range` and `List`, both of which were required to have an implementation of `size`. Implementing the check for `size` is easy enough

- simply explore all possible instantiation of `AbstractArray` - but it is not so easy in the general case. For example, I could have a version of `size` that takes multiple arguments like `size(::AbstractArray, ::AbstractArray)` (which requires joint exhaustion of all arguments) or could start using Julia's type language for richer properties such as `f(::T, ::T) where T<:AbstractArray`.

The protocol checker need not be complete; it must only be sound. My formalism relies on an abstract protocol checker **checkproto**($\mathtt{m}, \overline{t}, M$) that ensures that protocol `m` exists in the method table M with type arguments t. The implementation of **checkproto**must adhere to the following specification:

$$\frac{\mathbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: t}) :: \mu {=}\, e \triangleright \quad \wedge \quad \begin{array}{c}\forall \overline{v} : (\mathbf{typeof}(\overline{v}) = \overline{\sigma} \wedge \overline{\sigma} <: \overline{t}), \\ \overline{x :: t} \vdash_M e : t'_r\end{array} \quad \wedge \quad t'_r <: t_r}{\mathbf{checkproto}(\mathtt{m}, \overline{t}, t_r, M)}$$

Within Juliette I cannot add new types, only new methods. As a result, **checkproto**can guarantee existence of suitable methods as the method table continues to evolve; like **dispatch**, it can not guarantee that the return type is still correct, however, so dynamic checks will still be needed if the method table is modified.

Ensuring that some implementation exists for any instantiation of a given type is analogous to completeness checking in pattern matching. For the Juliette calculus I present a simple algorithm based on Maranget [55] that is able to provide sound and complete checking for the Juliette calculus, then discuss its key limitations when applied to generalized Julia.

MARANGET-STYLE CHECKING Maranget's algorithm is described in terms of patterns; I will first describe the pattern language and the briefly go over the function of the algorithm. I will then describe the reduction from Juliette protocol checking to these patterns.

In Maranget's system values solely consist of constructors $c(v_1, \dots, v_n)$; base values are constructors with no arguments (for example the nil constructor $nil()$). Patterns are then either wildcards _, constructor applications $c(v_1, \dots, v_n)$, or disjunctions $p_1 \mid p_2$.

I use Maranget's inexhaustiveness algorithm $\mathcal{I}(P, n)$ to implement completeness checking. In Maranget's setting P is a "matrix" of patterns (that is, each pattern is a row) and n is the number of arguments being matched. The algorithm $\mathcal{I}(P, n)$ returns a pattern vector p of size n that is *not* matched by P; if no such vector exists, it returns $\perp$. The implementation of $\mathcal{I}(P, n)$ used for Julia is identical to that of Maranget; I refer the reader to that treatment for details.

I then reduce completeness checking in JULIETTE to an instance of pattern matching in Maranget's language. I consider each leaf type σ to be a constructor with zero arguments and to inhabit the same type membership heirarchy as exists in JULIETTE. For example, `Int()` is a constructor for the type `Number`. In this manner, I can trivially define a mapping $\mathrm{pat}(\sigma) = \sigma()$ from JULIETTE base types to patterns. Abstract types are handled by conversion into a disjunction of base types.

Similarly, I can take a set of implementations and abstract an equivalent pattern matrix. Suppose that I have method implementations $\mathtt{m}(\overline{t^1}), \ldots, \mathtt{m}(\overline{t^n})$. I can construct a pattern matrix with $P(\mathtt{m}(\overline{t^1}), \ldots, \mathtt{m}(\overline{t^n})) = \begin{bmatrix} \overline{\mathrm{pat}(t_1)} \\ \ldots \\ \overline{\mathrm{pat}(t_n)} \end{bmatrix}$ that matches the converted argument if and only if a suitable method exists in the original JULIETTE method table.

This system is trivially incomplete. In particular, while it can adequately handle tag types, tuples, unions, and *simple* (non-bounded, non-diagonal) parametric types, its ability to check signatures that exploit Julia's bounded polymorphism is very limited.

As a practical example of where this may arise consider the earlier example of $+$ of two `Numbers`. I might write

```
@protocol +(::Number, ::Number)::Number
```

to specify that there must exist an implementation of $+$ for *any two* numbers. That is, if I have both `Int` and `Float64` as subtypes of `Number` I must implement

```
+(::Int, ::Int)::Number
+(::Float64, ::Float64)::Number
+(::Float64, ::Int)::Number
+(::Int, ::Float64)::Number
```

to satisfy this protocol. Obviously, writing exponentially many definitions gets wearisome after only a few types and Julia does not require that $+$ implementations do so.

Instead, Julia uses a mechanism called promotion (which I will not describe in detail here) to make the two types equal, therein requiring only two implementations:

```
+(::Int, ::Int)::Int
+(::Float64, ::Float64)::Float64
```

I could write this requirement as a protocol in Julia's full type language as

```
@protocol +(::T, ::T)::T where T<:Number
```

but then this wanders enthusiastically outside of the type language supported by the *warnings*-derived algorithm.

The Maranget-derived checker is sufficient to check protocols in Juliette programs because Juliette's type system does not have many of the complex features that real Julia has. While the concept of protocols is quite general, this specific implementation is not; a truly generic type system for generalized Julia would need a much more sophisticated protocol checking mechanism than I describe here.

4.4 Typed Dynamic Semantics

Next, I describe the dynamic semantics for typed Juliette.

The main differentiation between the typed and untyped dynamic semantics for Juliette arises from the handling of method invocation. Typed Juliette has three rules for method calls:

- E-Call used for untyped invocation, and is the same as base Juliette.
- E-TypedCallGlobal used to capture the current global evaluation context into a new local evaluation context.
- E-CallInferredLocal used when the current method table M is the same as the one used to statically check correctness and thus for which static inference can be relied upon.
- E-CallCheckedLocal used when the method table M has been extended with some M_e and thus the returned value needs to be dynamically checked.

E-TypedCallGlobal

$$\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{\mathrm{M}'} \mathrm{t}_r]\,\rrparenthesis]\rangle \;\rightarrow\; \langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{\mathrm{M}'} \mathrm{t}_r]\,\rrparenthesis_{\mathrm{M}}]\rangle}$$

E-CallInferredLocal

$$\frac{\textbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \textbf{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \mathrm{t}}) :: \mu = e\, \triangleright}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{\mathrm{M}'} \mathrm{t}_r]\,\rrparenthesis_{\mathrm{M}'}]\rangle \;\rightarrow\; \langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\llbracket\, e[\overline{x} \mapsto \overline{\mathtt{v}}]\,\rrbracket]\,\rrparenthesis_{\mathrm{M}'}]\rangle}$$

E-CallCheckedLocal

$$\frac{\textbf{typeof}(\overline{\mathtt{v}}) = \overline{\sigma} \qquad \textbf{getmd}(\mathrm{M}', \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \mathrm{t}}) :: \mu = e\, \triangleright}{\langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{\mathrm{M}' \bullet \mathrm{M}''} \mathrm{t}_r]\,\rrparenthesis_{\mathrm{M}'}]\rangle \;\rightarrow\; \langle \mathrm{M}, \mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[\llbracket\, e[\overline{x} \mapsto \overline{\mathtt{v}}]\,\rrbracket_{\mathrm{t}_r}]\,\rrparenthesis_{\mathrm{M}'}]\rangle}$$

E-ValChecked

$$\frac{\textbf{typeof}(\mathtt{v}) <: \mathrm{t}}{\langle \mathrm{M}, \mathtt{C}\,[\llbracket\, \mathtt{v}\,\rrbracket_{\mathrm{t}}]\rangle \;\rightarrow\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}$$

E-ValInferred

$$\frac{}{\langle \mathrm{M}, \mathtt{C}\,[\llbracket\, \mathtt{v}\,\rrbracket]\rangle \;\rightarrow\; \langle \mathrm{M}, \mathtt{C}\,[\mathtt{v}]\rangle}$$

E-TypeErr

$$\frac{\textbf{typeof}(\mathtt{v}) \not<: \mathrm{t}}{\mathrm{M} \vdash \mathtt{C}\,[\llbracket\, \mathtt{v}\,\rrbracket_{\mathrm{t}}] \rightarrow \mathtt{error}}$$

Figure 29: Dynamic semantics for typed Juliette.

Typed invocations are solely the domain of the type system; the programmer cannot manually write typed calls.

The two local typed invocations each rely on their own cast form: $[\![e]\!]_t$which performs a dynamic check that e has type t and $[\![e]\!]$that indicates that e has a reliable inferred type. $[\![e]\!]_t$is used when the return type of e cannot be statically guaranteed and needs to be dynamically checked. In $[\![e]\!]_t e$ might get stuck at any point or it might be an ill-typed value, in which case the program steps to an error. $[\![e]\!]$, on the other hand, guarantees that e inferred to t under the relevant method table. As a result, e might get stuck internally but if e is a value then it will always be well-typed.

I use a standard progress and preservation proof methodology to show soundness in JULIETTE. Towards this end, I define a typing relation for the target language, shown in figure 30. The system follows the typed translation rules closely, with the primary exception being the inclusion of TE-CAST that simply asserts that the result of evaluating some expression e is type t, therein providing a interpretation of the cast contexts $[\![e]\!]_t$.

This target type system is similar to the one in our prior work in Belyakova et al [9] but diverges by providing a typing judgment for function invocations TE-CALL, for inferred expressions TE-INFERRED, and for checked expressions TE-CHECKED. Additionally, it prohibits nested local and global evaluation contexts from existing within typed expressions.

From here, I can state soundness in typed JULIETTE as follows.

Theorem 3 (Soundness of typed JULIETTE)**.** *If* $\vdash_{M;P} e : t$ *one of three cases holds:*

1. e *is a value* v *where* $\textbf{typeof}(v) <: t$
2. *For all* M', M_e *there is some* M'' *and* e' *such that* $\langle M', e\rangle \rightarrow_{M \bullet M_e} \langle M'', e'\rangle$ *and* $\vdash_{M;P} e' : t$
3. e *is of the form* $X[m(\overline{v})]$ *and there exists an equivalence class* T *in* M *of at least two methods with the form* $\overline{\triangleleft\, m(\overline{\_ :: t_a}) :: \_ = \_ \triangleright} \subseteq \text{applicable}(\textbf{latest}(M), m, \overline{\sigma})$ *that are not pairwise related by subtyping (for any two argument types* $\overline{t_a}$ *and* $\overline{t'_a}$ *in* T *then* $\overline{t_a} \not<: \overline{t'_a}$ *and* $\overline{t'_a} \not<: \overline{t_a}$*) and that are are more precise than all other implementations (e.g. for any argument type* $\overline{t_a}$ *in* T *then* $\forall \triangleleft\, m(\overline{\_ :: t}) :: \_ = \_ \triangleright \in \text{applicable}(\textbf{latest}(M), m, \overline{\sigma})$ *if* $\overline{t} \notin T$ *then* $\overline{t_a} <: \overline{t}$*).*
4. *There exists some* X, e', t' *such that* $e = X[[\![e']\!]_{t'}]$.

$$\frac{\textbf{typeof}(\mathtt{v}) = \sigma}{\Gamma \vdash_{M;P} \mathtt{v} : \sigma}\ \text{TE-Value} \qquad \frac{x : t \in \Gamma}{\Gamma \vdash_{M;P} x : t}\ \text{TE-Var} \qquad \frac{\Gamma \vdash_{M;P} e_1 : t_1 \qquad \Gamma \vdash_{M;P} e_2 : t_2}{\Gamma \vdash_{M;P} e_1\, ;\, e_2 : t_2}\ \text{TE-Seq}$$

$$\frac{\Gamma \vdash_{M;P} e : t \qquad t <: t'}{\Gamma \vdash_{M;P} e : t'}\ \text{TE-Sub} \qquad \frac{\overline{\Gamma \vdash_{M;P} e : t} \qquad \Psi(l, \overline{t}) = t'}{\Gamma \vdash_{M;P} \delta_l(\overline{e}) : t'}\ \text{TE-Primop}$$

$$\frac{\Gamma \vdash_{M;P} e : \textbf{typeof}(\mathtt{m}) \qquad \overline{\Gamma \vdash_{M;P} e' : t_a} \qquad \textbf{dispatch}(M, P, \mathtt{m}, \overline{t_a}) = t_r}{\Gamma \vdash_{M;P} e(\overline{e'}) ::_{M'} t_r : t_r}\ \text{TE-Call}$$

$$\frac{\vdash_M e : t}{\Gamma \vdash_{M;P} [\![e]\!] : t}\ \text{TE-Inferred} \qquad \frac{}{\Gamma \vdash_{M;P} [\![e]\!]_t : t}\ \text{TE-Checked}$$

Figure 30: Typed Juliette target language typing

5. *There exists some* X, e' *such that* $e = X[[\![e']\!]]$ *where* $\forall v, e' \neq v$.

The first two cases are obvious: if I have executed the program to a value then it should be correctly-typed. Similarly, if I have a typed context within our execution, it should step to a similarly well-typed form.

The third case of soundness is the carve-out for ambiguous method invocations. An ambiguous method invocation occurs when there is a congruence class within the set of applicable methods (e.g. those that could handle the given arguments) that are not comparable by subtyping (that are equally precise as one another) but that are all more precise than any other implementation. This is where the carve-out of the definition of **dispatch** shows up: it merely guarantees that a method *exists*, not that there will always be a unique most specific one. Thus, I cannot guarantee the absence of such an equivalence class based on a successful **dispatch** check.

The fourth case of soundness applies to dynamic typechecks applied to typed function invocations when new methods M_e have been added to the original method table M. In this case I cannot guarantee that all callable methods have a return type that infers to a subtype of the expected type t and thus must dynamically check the return type. Moreover, the body of the unchecked and uninferred method might go wrong at any time.

Finally, the fifth case of soundness applies to the expressions resulting from function invocations that occurred against the statically relevant method table. In this case **dispatch** guarantees that whatever method is called infers to return a subtype of the expected type t. Thus, the embedded expression e might go wrong but if it is a value $\mathtt{v}$ then it will be appropriately typed.

Note, first, that soundness generalizes on any global method table M' and extended local method table M_e. Typed methods themselves will thus not go wrong; ill-typed values returned from newly-`eval`ed methods might cause a cast failure but will not break typed code.

Next, observe that $[\![e]\!]$requires no dynamic checks. Dynamic checks $[\![e]\!]_t$are only inserted when the local method table has been extended with some M_e. As a result, if no new methods are visible in the local method table the type system applies no additional dynamic overhead. Only once methods that were not known to the static checker have been added does the type system begin adding overhead.

PROOF For the use of the proof I define *redexes* for the calculus, derived from the original concept that JULIETTE used. A redex is an expression that is immediately reducible and contains no reducible subexpressions. The redexes in typed JULIETTE are shown in Fig. 31; compared to the original JULIETTE calculus typed JULIETTE adds the cast redex as well as redexes for statically-typed function calls in global and local method tables.

I use three lemmas for the proof of soundness.

Lemma 4. Unique Form of Expressions Any expression e can be uniquely represented in one of the following ways:

- $e = \mathtt{v}$
- $e = \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}})]$
- $e = \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_M t]$
- $e = \mathtt{X}\,[\mathtt{rdx}]$

Proof. By induction on e using auxiliary definitions and lemmas extended from [9]; an additional set of canonical forms representations are added to handle $e = \mathtt{X}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_M t]$ and casts $[\![e]\!]_t / [\![e]\!]$. □

rdx	::=		*Redex Base*	
	$\mid$	x	variable	(error)
	$\mid$	$\mathtt{v}\,;\,e$	sequencing	(normal)
	$\mid$	$[\![\,\mathtt{v}\,]\!]_t$	cast	(normal/error)
	$\mid$	$[\![\,\mathtt{v}\,]\!]$	cast	(normal)
	$\mid$	$\mathtt{v}^{\neq\mathtt{m}}(\overline{\mathtt{v}})$	non-function call	(error)
	$\mid$	$\mathtt{md}$	method definition	(normal)
	$\mid$	$\delta_\iota(\overline{\mathtt{v}})$	primop call	(normal/error)
	$\mid$	$(\!\vert\, \mathtt{v} \,\vert\!)$	value in global context	(normal)
	$\mid$	$(\!\vert\, \mathtt{v} \,\vert\!)_M$	value in table context	(normal)
	$\mid$	$(\!\vert\, X\,[\mathtt{m}(\overline{\mathtt{v}})] \,\vert\!)$	function call in global context	(normal)
	$\mid$	$(\!\vert\, X\,[\mathtt{m}(\overline{\mathtt{v}})] \,\vert\!)_M$	function call in table context	(normal/error)
	$\mid$	$(\!\vert\, X\,[\mathtt{m}(\overline{\mathtt{v}}) ::_M t] \,\vert\!)$	typed function call in global context	(normal)
	$\mid$	$(\!\vert\, X\,[\mathtt{m}(\overline{\mathtt{v}}) ::_M t] \,\vert\!)_M$	typed function call in table context	(normal/error)

Figure 31: Redex Bases

Lemma 5. Context Irrelevance $\langle M_g, \mathtt{C}\,[\mathtt{rdx}]\rangle \to \langle M'_g, \mathtt{C}\,[e]'\rangle \iff \langle M_g, \mathtt{C}'\,[\mathtt{rdx}]\rangle \to \langle M'_g, \mathtt{C}'\,[e]\rangle$

Proof. By analyzing normal-evaluation steps I can see that only $\mathtt{rdx}$ matters for the reduction; therefore, by inspecting the reduction step for either $\mathtt{C}\,[\mathtt{rdx}]$ or $\mathtt{C}'\,[\mathtt{rdx}]$ I can then construction a corresponding step for $\mathtt{C}'$ or $\mathtt{C}$, respectively. □

Lemma 6. Simple-Context Irrelevance $\langle M_g, e\rangle \to_M \langle M'_g, e'\rangle \implies \langle M_g, \mathtt{X}\,[e]\rangle \to_M \langle M'_g, \mathtt{X}\,[e']\rangle$

Proof. By lemma 4 e is either $\mathtt{v}$, $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}})]$, $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{M_i} t]$, or $\mathtt{C_e}\,[\mathtt{rdx}]$. If e is $\mathtt{v}$ then the assumption cannot hold (since $\mathtt{C}\,[\llparenthesis\, \mathtt{v} \,\rrparenthesis_M]$ steps to $\mathtt{C}\,[\mathtt{v}]$). Therefore, I need only consider the $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}})]$, $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{M_i} t]$, or $\mathtt{C_e}\,[\mathtt{rdx}]$ cases.

- When e is $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}})]$ then $\llparenthesis\, e \,\rrparenthesis_M = \llparenthesis\, \mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}})] \,\rrparenthesis_M$ is a redex and $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M]$ steps by E-CallLocal. Similarly, $\llparenthesis\, \mathtt{X}\,[e] \,\rrparenthesis_M = \llparenthesis\, \mathtt{X}\,[\mathtt{X_e}]\,[\mathtt{m}(\overline{\mathtt{v}})] \,\rrparenthesis_M$ is also a redex and steps as $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M]$

- When e is $\mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{M_i} t]$ then $\llparenthesis\, e \,\rrparenthesis_M = \llparenthesis\, \mathtt{X_e}\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{M_i} t] \,\rrparenthesis_M$ is a redex and $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M]$ steps by E-CallLocal. Similarly, $\llparenthesis\, \mathtt{X}\,[e] \,\rrparenthesis_M = \llparenthesis\, \mathtt{X}\,[\mathtt{X_e}]\,[\mathtt{m}(\overline{\mathtt{v}}) ::_{M_i} t] \,\rrparenthesis_M$ is also a redex and steps as $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M]$.

- When e is $\mathtt{C_e}\,[\mathtt{rdx}]$ then $\llparenthesis\, e \,\rrparenthesis_M = \llparenthesis\, \mathtt{C_e}\,[\mathtt{rdx}] \,\rrparenthesis_M$ and $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M] = \mathtt{C}'\,[\mathtt{rdx}]$ where $\mathtt{C}' = \llparenthesis\, \mathtt{C_e} \,\rrparenthesis_M$. Since $\mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[e] \,\rrparenthesis_M] = \mathtt{C}''\,[\mathtt{rdx}]$ for $\mathtt{C}'' = \mathtt{C}\,[\llparenthesis\, \mathtt{C_e} \,\rrparenthesis_X]$, $\mathtt{C}\,[\llparenthesis\, e \,\rrparenthesis_M]$ and $\mathtt{C}\,[\llparenthesis\, \mathtt{X}\,[e] \,\rrparenthesis_M]$ step similarly by lemma 5.

□

The proof of soundness for Juliette is straightforward by rule induction on the rule used to derive $\vdash_{M;P} e : t$:

Proof.

- TE-Value: trivial, case (1) of soundness.
- TE-Var: impossible, as Γ is empty.
- TE-Seq: Apply the IH to the first typing relation $\vdash_M e_1 : t_1$:
  - If e_1 is a value, then apply E-Seq; case 2 of soundness applies.

– If there is some e_1' such that $\langle M', e_1\rangle \rightarrow_M \langle e_1', M''\rangle$ and $\vdash_{M;P} e_1' : t_1$ then let $X = \square; e_2$ and therefore by simple context irrelevance $\langle M', X[e_1]\rangle \rightarrow_M \langle M'', X[e]_1'\rangle$; let $e' = X[e_1'] = e_1'; e_2$ and therefore $\langle M', X[e_1]\rangle \rightarrow_M \langle M'', e'\rangle$ and since $\vdash_{M;P} e_2 : t_2$ it follows that $\vdash_{M;P} e' : t_2$. Case 2 holds.

– If e_1 is an ambiguous method call $X'[m(\overline{v})]$ then e is stuck at the same ambiguous method call via the X construction as in case 1. Case 3 holds.

– If there is some X' and e_1' such that $e_1 = X[[\![e_1']\!]_t]$ then construct $X = X'; e_2$ and therefore $e = X[[\![e_1']\!]_t]$; case 4 holds.

– As in case 4.

.

- TE-SUB: Apply the IH to the typing relation.
  – If e is a value v and $\textbf{typeof}(v) <: t'$ then by transitivity $\textbf{typeof}(v) <: t$ and case (1) holds.
  – If $\langle M', e\rangle \rightarrow_M \langle e', M''\rangle$ and $\vdash_{M;P} e' : t'$ then $\vdash_{M;P} e' : t$ by TE-SUB.
  – Trivial.
  – Trivial.
  – Trivial.

.

- TE-PRIMOP: Apply the IH inductively over the typings of the primop arguments $\overline{\vdash_M e : t}$. If any argument i steps then I construct a context $X = \delta_l(\overline{v}Xe)$ and the expression as a whole steps via simple context irrelevance as in TE-SEQ. Additionally, the context X suffices to show that if either case 2 or 3 apply for any argument i then the respective case applies to the expression as a whole. Otherwise if all arguments are values $\overline{v}$ then by the IH case 1 of soundness applies and $\textbf{typeof}(\overline{v}) <: \overline{t}$. Then, by the definition of the typed primop resolver $\Psi(l, \overline{v}) = v'$ where $\textbf{typeof}(v') <: t'$. Therefore I can apply E-PRIMOP to find that $\langle M', \delta_l(\overline{v})\rangle \rightarrow_M \langle M'', v'\rangle$ and case 2 applies.
- TE-INFERRED: I apply the correctness property of inference 1 to the embedded expression and case analyze.

  - I know that $\exists \mathtt{v} :\ e = \mathtt{v}$ and $\textbf{typeof}(\mathtt{v}) <: \mathsf{t}$. Apply E-CallChecked; case (2) applies.
  - I know that $\forall M' \exists M'', \langle M', e\rangle \rightarrow_M \langle M'', e'\rangle$ and $\vdash_M e' : \mathsf{t}$. I can therefore construct $\mathtt{X} = [\![\square]\!]_{\mathsf{t}}$ and therefore by simple context irrelevance $\langle M', \mathtt{X}[e]\rangle \rightarrow_M \langle M'', \mathtt{X}[e']\rangle$. By rule TE-Inferred $\vdash_{M;P} [\![e']\!] : \mathsf{t}$ and thus case (2) applies.
  - Construct $\mathtt{X}$ as in case 2; the ambiguous method definition remains thus and case 3 applies.
  - Construct $\mathtt{X}$ as in case 2 and case 4 applies.
  - If e is an expression then construct $\mathtt{X}$ as in case 2 and thus case 5 applies. Otherwise, if there is some value $\mathtt{v}$ such that $e = \mathtt{v}$ then since $\vdash_M \mathtt{v} : \mathsf{t}$ it follows that $\langle M', [\![\mathtt{v}]\!]\rangle \rightarrow_M \langle M'', \mathtt{v}\rangle$ since $\textbf{typeof}(\mathtt{v}) <: \mathsf{t}$ by case (1) of the definition of inference and thus case (2) applies.

- TE-Checked: Trivial, case 5 of soundness.
- TE-Call: Apply the IH inductively as before, using the callee form of simple evaluation contexts for the receiver and the argument form for arguments. If any step then I construct the associated simple evaluation contexts and the expression as a whole steps.

  If the receiver and all arguments are values with types $\mathtt{m}$ and $\overline{\sigma}$ respectively, then one of two cases of **dispatch** could have applied: normal or protocol invocation.

  - **Normal invocation**: by the definition of **dispatch** there is some $\triangleleft\, \mathtt{m}(\overline{x :: \mathsf{t}'_a}) :: \mu = e \,\triangleright \in M$ where $\overline{\textbf{typeof}(\mathtt{v})} <: \mathsf{t}'_a$. Therefore, I need to case analyze on if there is a unique most specific applicable method:
    * $\textbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: \mathsf{t}'_a}) :: \mu = e' \,\triangleright$. Case analyze on whether M_e is empty or not.
      - M_e is empty. By the definition of **dispatch**, $\overline{x :: \mathsf{t}_a} \mathrel{|\!\!\sim_M} e' : \mathsf{t}_r$. By substitution for inference, therefore, $\mathrel{|\!\!\sim_M} e'[\overline{x} \mapsto \overline{\mathtt{v}}] : \mathsf{t}_r$. Thus, $\vdash_{M;P} [\![e'[\overline{x} \mapsto \overline{\mathtt{v}}]]\!] : \mathsf{t}_r$. Moreover, since $\langle M', \mathtt{m}(\overline{\mathtt{v}}) ::_M \mathsf{t}_r\rangle \rightarrow_M \langle M'', [\![e'[\overline{x} \mapsto \overline{\mathtt{v}}]]\!]\rangle$ by E-CallInferredLocal case (2) applies.

· M_e is nonempty. Then $\langle M', \mathtt{m}(\overline{v}) ::_M t_r\rangle \rightarrow_{M \bullet M_e} \langle M'', [\![e'[\overline{x \mapsto v}]]\!]_{t_r}\rangle$ by E-CallCheckedLocal. Then, $\vdash_{M;P} [\![e'[\overline{x \mapsto v}]]\!]_{t_r} : t_r$ by E-ValChecked.

* $\mathbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \mathtt{error}$ where $\min(\text{applicable}(\mathbf{latest}(M), \mathtt{m}, \overline{\sigma})) = \mathtt{error}$ due to there being an equivalence class T of most-applicable signatures that are not otherwise related by subtyping. Case (3) of soundness applies as I let $\mathtt{X} = \mathtt{X}'$ and then $\mathtt{X}\,[\mathtt{m}(\overline{v})] = \mathtt{m}(\overline{v})$.

– **Protocol invocation**: by definition of **dispatch** there was some $\triangleleft\, \mathtt{m}(\overline{:: t'_a}) :: t_r \triangleright \in P$ where $\overline{\mathbf{typeof}(v) <: t'_a}$. By well-formedness of the protocol table I then have that $\mathbf{checkproto}(\mathtt{m}, \overline{t}, t_r, M)$ and then that $\forall \overline{v}' : (\mathbf{typeof}(\overline{v}') = \overline{\sigma} \wedge \overline{\sigma <: t'_a}), \mathbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: t}) :: \mu{=}\, e \triangleright \wedge \overline{x :: t} \vdash_M e : t'_r \wedge t'_r <: t_r$. Instantiating $\overline{v}'$ with $\overline{v}$ gives (1) $\mathbf{getmd}(M, \mathtt{m}, \overline{\sigma}) = \triangleleft\, \mathtt{m}(\overline{x :: t}) :: \mu{=}\, e \triangleright$, (2) $\overline{x :: t} \vdash_M e : t'_r$, and (3) $t'_r <: t_r$. Protocol invocation then proceeds as with normal invocation with the $\mathtt{m}$ provided by (1), the inference result from (2), and subsumption using (3).

□

By soundness, then, *within the local method table* M, no return type checks need be inserted; so long as the inference result is correct it follows that all statically-determined return types are then correct. If the local method table is then extended with some M_e I must then start dynamically checking return types.

This soundness theorem ultimately answers my thesis statement. In particular,

- The concrete semantics aligns neatly with Julia's existing concept of typing; the theorem wholly relies on Julia's inherent definition of type while still being able to produce a strong guarantee.

- Soundness does not rely on any property of subtyping besides transitivity, as mentioned earlier.

- Performance depends on whether the dynamic local method table matches the static method table. If the method tables are the same then no overhead is incurred. If they differ then return type tests must be performed.

These properties make the nature of the type system dependent on whether new methods have been added or not. If new methods have not been added then typed code receives guarantees comparable to those of a fully static language: calls to other functions are guaranteed to go through and their return types are guaranteed to be correct. If new methods have been added then the guarantee degrades to a hybrid between the concrete and transient semantics under my taxonomy as part of KafKa [24]. As in the transient semantics, I must check returned values to ensure that they adhere to the statically determined types. However, unlike the transient semantics, I do so by checking their entire identity through the type tag rather than merely their surface-level structure.

Thus, on paper, I have an answer to my original question: yes, it is possible to design a gradual type system for Julia that matches the philosophy of the language. How practical is it though, in its present form?

5 TYPED JULIA IN PRACTICE

With the theory laid out, I can now examine the realization of Typed Julia. In this chapter I will describe the implementation of the type checker and discuss how it works on a few real programs as well as empirically evaluate some of the assumptions that went into its design.

5.1 IMPLEMENTATION

The type checker is implemented in Julia and relies extensively on existing Julia language features. The type checker is a standalone Julia program that analyzes source files and produces warnings and errors.

The type checking pipeline consists of three packages, two of which were developed for this project. A general schematic is shown in Fig. 32.

Programs go through four phases of type checking:

- Syntactic analysis: Julia files are parsed into Julia's expression form.
- Semantic analysis: raw Julia ASTs in the form of S-expressions are parsed into a semantic AST.
- Module scope analysis: Julia files are traversed to identify the relationship between files and to identify what module each file is logically in.

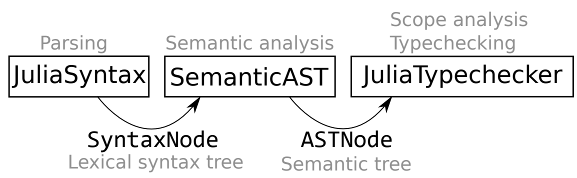


Figure 32: Type checker architecture

Package	Lines of code	Files
SemanticAST	1160	5
JuliaTypechecker	1453	7

Table 2: Typechecker component packages

- Type analysis: annotated methods are identified by the type checker and are analyzed.

Syntactic and semantic analyses are performed by the dedicated libraries `JuliaSyntax` and `SemanticAST` respectively. Both scope analysis and type analysis are performed by the package `JuliaTypechecker`. `SemanticAST` and `JuliaTypechecker` were developed specifically for type checking. The broad size of each new package is indicated in table 2; statistics exclude test files.

5.1.1 *Syntactic analysis*

Parsing and syntactic analysis is performed using the `JuliaSyntax` library, a Julia parser implemented in Julia. `JuliaSyntax` provides several benefits when compared to Julia's own parser, including character-precise attribution information as well as a better-described expression representation. However, `JuliaSyntax`'s expressions still largely follow Julia's which poses a practical challenge to semantic analysis.

Julia expressions (and by extension those produced by `JuliaSyntax`) are based on S-expressions and are close to the syntax of the input file. As a result, the same semantic concept can be represented in many different forms. For instance, function declarations can come with one of five different expression heads which need to be disambiguated based on their children.

5.1.2 *Semantic analysis*

My solution to this problem is the Julia source-level semantic analyzer `SemanticAST`. `SemanticAST` takes the expressions produced by `JuliaSyntax` and parses them into semantically meaningful ASTs. In `SemanticAST` there are only two forms for function declarations, both of which are fully descriptive at the top level.

As an example, consider the following definitions:

Source	S-expression
`f(x) = 3`	`(= (call f x) (block 3))`
`f(x) where T = 3`	`(= (where (call f x) T) (block 3))`
`f(x)::Int = 3`	`(= (:: (call f x) Int) (block 3))`
`f(x)::Int where T = 3`	`(= (:: (call f x) (where Int T)) (block 3))`

All of these definitions describe an identical function. However, the head and structure of the S-expression differs to match the precise source syntax. For example, if I specify a type variable then the function's head is now **`where`** or if I specify both a type variable and a return type then the head is `::` with two arguments: the "normal" function head followed by a **`where`** structure whose body and type variables are then interpreted as the return type and the quantifier for the whole function, respectively.

Interpreting Julia's parsed S-expressions is then traditionally difficult: one must handle a wide range of special cases to align with programmer expectations. My SemanticAST library provides an abstraction layer over these details: all function definition have a single representation.

In this example, all four methods are inline definitions of a function named `f`. Inline definitions are handled as any other assignment would be, so all take on the form `Assignment([lvalue], [expression])`. The lvalue in these cases are different `FunctionAssignment` instances, while the rvalue is the expression `Literal(3)`. `FunctionAssignment` then takes on the form `FunctionAssignment(name::FunctionName, args_stmts::Vector{FnArg}, kwargs_stmts::Vector{KwArg}, sparams::Vector{TyVar}, rett::Union{Expression, Nothing})`, taking a name, the positional, keyword, and type arguments, followed by the return type.

Source	ASTNode for LValue
`f(x) = 3`	`FunctionAssignment(f,FnArg(x),[],[],nothing)`
`f(x) where T = 3`	`FunctionAssignment(f,FnArg(x),[],[TyVar(T)],nothing)`
`f(x)::Int = 3`	`FunctionAssignment(f,FnArg(x),[],[],Var(Int))`
`f(x)::Int where T = 3`	`FunctionAssignment(f,FnArg(x),[],[TyVar(T)],Var(Int))`

In all four cases above the L-value is now a consistent instance of `FunctionAssignment`. Downstream consumers, such as the type system, need then only worry about assigning to a function rather than "what happens if I see a `::` in the lvalue of an assignment."

The implementation of `SemanticAST` is based on Julia's own semantic analysis step: lowering. Lowering is the first phase of compilation in Julia after parsing and converts parsed S-expressions into a pseudo-SSA representation that's then fed into later stages of analysis. Lowering is, however, unsuitable for use in static tooling as the process loses attribution information and performs numerous other transformations on the program being lowered. In constrast, `SemanticAST` maintains equivalence to the source program while abstracting parsing details.

One challenge for semantic AST analysis is macros. The current implementation is hard coded; a few common macros are supported but most fall into a hardcoded exception. Generalized macro handling is a challenge for static semantic analysis as while the macros could be expanded and their output analyzed this may lose source-level meaning. A "method not found" error from within a macro expansion is difficult to understand without examining the macro itself. I will return to this topic when discussing future work.

5.1.3 *Scope analysis*

My next step is module scope analysis. Julia packages and projects usually consist of multiple files that are all related by some sort of "root." Julia packages, for instance, are loaded by executing their eponymous file (for instance, `SemanticAST` will be loaded from the file `SemanticAST.jl`), which is then responsible for using the `include` function to load the remainder of the package. This root file can use the entire Julia language to decide whether to `include` a file or not. Moreover, `include` loads a file into whatever the *current* module is. Knowing the module that a file was `include`d into is essential in order to determine the module whose definitions it should be type checked against.

As a brief example, suppose I had three files, as shown in Fig. 33.

```
module A
    const x = 1
    include("A.jl")
end
module B
    const x = "hello"
    include("B.jl")
end
```

```
foo() = x
```

```
foo() = x
```

`Root.jl` `A.jl` `B.jl`

Figure 33: A tiny package

This package will load the contents of `A.jl` into module `A` and will load the contents of `B.jl` into module `B`. If I call `A.foo()` then I get an integer back; if I call `B.foo()` then I get a string back. I cannot identify which `x` is being used without knowing from where the reference is being `include`d from. Julia files must be analyzed within the context of the entire project.

Scope analysis is then a two-step process: files must first be analyzed to figure out what "tree" that they might be contained in, then their scopes can be determined based on while file(s) they might be included from.

The first step of scope analysis is "tree" identification: determining which files include what other files. The algorithm begins with a set of "roots" (top-level files from which other files are used) and then removes roots when a reference from one preliminary root to another is identified. In pesudocode the algorithm can be described as

```
roots = the set of all files
for file in roots
    for referenced_file in references(file)
        if referenced_file in roots
            remove(roots, referenced_file)
        end
    end
end
```

Once root identification has occured scope identification can start. Scope identification starts at each root and runs recursively into each referenced file, propagating the scope from the referencing file into the referenced. Again, in pseudocode, this works as follows:

```
function analyze_scope(outer_scope, file)
    for module in file
        inner_scope = combine(outer_scope, module)
        for referenced_file in references(module)
            analyze_scope(inner_scope, referenced_file)
        end
    end
end

for root in roots
    analyze_scope(main scope, root)
end
```

The implementation of scope analysis has two major limitations: it miss files that are added by calling `include` with some variable value; similarly, it may collect files that are included from within a method that is never used. However, most `include` usages by packages are at the top level with a hardcoded string. Scope analysis takes around 200 lines of code.

5.1.4 *Type analysis*

Finally, I preform type analysis. Using the scope information provided by the previous step the type checker performs a recursive descent through each function annotated with the special-cased `@typed` macro in the set of files provided.

The implementation of the type checker follows the typing rules as described in figure 28. It recursively descends into expression forms while maintaining a type checking context to determine the type of an expression. The code itself is small at 1,100 lines; A number of architectural decisions deserve discussion, however.

entity-component system The type checker maintains a side data structure to the AST in entity-component form. Entity component systems are a design pattern from video games that support large numbers of lightweight entities that can have many components. Each component contains some additional information. This design is used in video games to decouple various high-level behaviors from one another by making them only have to consider which entities have which components.

The type checker uses aspects of this entity-component architecture to allow it to handle semantic information about the AST. Each AST node is associated by the type checker with an entity; each entity has an AST component that indicates which node it is attached to and what its parent is. The type checker then attaches various additional components to these entities as it type checks the program including what the inferred type of a given sub-expression is, what methods a given call site could be dispatched to, and what the errors at a given location were.

Using entities and components to capture this metadata allows very straightforward use of the information by other parts of the type checker. One example is that scope information is included as a component. Moreover, it allows weak coupling between type checker components as analyses only need to consider the components that are relevant to their task.

METHOD TABLES As in the theory, the implementation performs type checking and inference against some reference method table. My implementation prepares a Julia instance with the package loaded to act as this table. In turn, this Julia instance then is used to

- resolve subtyping queries,
- find applicable methods,
- infer return types for a given invocation.

By using a real running Julia instance the method table used for type checking is closely aligned to the method table used at runtime by most packages. Definitions are dynamically reloaded when files are changed using `Revise.jl`.

Using an actual Julia instance to serve as the "black box" for type checking closely aligns the type checker with Julia's runtime reasoning about types but has drawbacks. In particular, Julia cannot dynamically reload redefinitions of the same type and the instance must be reinitialized from scratch each time a type definition is changed.

INFERENCE Again as in the theory I use type inference to determine the return types for invocations. The implementation uses its Julia instance to perform this inference, leveraging the same inferencer used for runtime type specialization of methods. My type checker queries it with the statically determined arguments for a given method to determine the inferred return type.

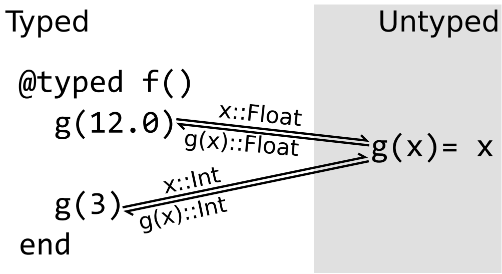


Figure 34: Inference example

A consequence of using Julia's inference system to determine the return type for each invocation site is that different typed calls to the same untyped method may result in different return types. Fig. 34 illustrates a simple example. Here, function `f` calls the untyped identity function `g` with the floating point number `12.0` and the integer `3`. The inferred return type for `g` at the first call site is `Float64` while at the latter it is `Int` even though the same function is being called.

Rerunning inference for each call site has several benefits but also tradeoffs. When used with concrete types the inference returns very precise types and aligns closely with the dynamic behavior of Julia's optimizer. However, the inferencer struggles with abstract types. For example, the inferred return type for `+` when applied to two `Numbers` is `Any`. As I will see, imprecision of abstract inference can pose challenges for typing library code.

5.2 EVALUATION

The practice of a type system is an inherent part of its design; at several times in my theoretical treatment I have made design decisions fundamentally motivated by practical justifications. Moreover, my type system is a basic framework from which more of Julia's features may be covered, but I do not know just how comprehensive my treatment of Julia is.

I wanted to answer three questions about the type checker:

- What is the right choice of method table? How many programs might use `eval` to extend their method table after they are first loaded and thereby require return type checks?

- How common are signatures in actual code and are they completely implemented?
- What mutations are needed for existing Julia programs to be type checkable?

5.2.1 *Picking a method table*

As mentioned in the theoretical examination of the type system it is important to carefully chose the method table M that I type check against. If the table M is the one actually used at runtime then the type system is "free": no dynamic checks are needed. If there are additional methods added after the fact then this guarantee goes out the window (though practically only for functions that *actually* have new methods added as a result of JIT compilation).

As part of my earlier work on world age and method tables in Julia [9] I conducted an emperical evaluation of uses of `eval` in the wild. I will summarize the relevant results here.

I wanted to examine how frequently libraries either modify the method table themselves after initialization or support user programs that do. Self-modifications of the method table after initialization would need to use `eval` somewhere within a method. If a library wishes to support a user program that adds new methods after initialization then it must call those methods using either `eval` or the `invokelatest` function from inside a method. Therefore, I statically analyzed a corpus to examine how many packages use `eval` or `invokelatest` from withing methods.

My corpus consists of all 4,011 registered Julia packages as of August 2020. The results of statically analyzing the code base are shown in Fig. 35a. The analysis shows that 2,846 of the 4,011 packages used neither `eval` nor `invokelatest`, and thus are definitely age agnostic. Of the remaining packages, 1,094 used `eval` only, and so *could* be impacted. 15 packages used `invokelatest` only, and some 56 used both. I can reasonably presume that at least these latter 71 packages are impacted by world age because they bypass it using `invokelatest`.

To understand if packages that only use `eval` dynamically define or use dynamically defined methods, I statically analyzed the location of calls to `eval` and their arguments by parsing files that contain `eval`. For each call, I classified the argument ASTs, recursively traversing them and counting occurrences of relevant nodes. The analysis is conservative: it assumes that an AST that is not statically obvious (such as a variable) could contain anything. Fig. 35b shows how many packages use

(a) `eval` and `invokelatest` use by package

(b) Static use of AST forms in all packages

Figure 35: Dynamic code generation usage metrics

method table relevant AST forms. Only uses of `eval` from within methods—where the method table could be relevant—are shown; top-level uses of `eval`—which cannot be affected by world age—are filtered out. The “all-others” category encompasses all AST forms not relevant to world age. While this aggregate is, taken as a whole, more common than any other single AST form, none of the constituent AST forms is more prevalent than function calls. Therefore, most common arguments to `eval` are function definitions, followed by function calls and loading of modules and other files.

Using the results of the static analysis, I estimate that about 4–9% of the 4,011 packages might be affected by world age. The upper bound (360 packages) is a conservative estimate, which includes 289 packages with potentially method table-related calls to `eval` but without calls to `invokelatest`, and the 71 packages that use `invokelatest`. The lower bound (186 packages) includes 115 `invokelatest`-free packages that call `eval` with both function definitions and function calls, and the 71 packages with `invokelatest`.

This analysis only considers usages of `eval` that are not at the top level— that is, usages that are within function bodies or similar constructs. Using `eval` at the top level is a common practice in Julia packages to generate boilerplate definitions. From the perspective of world age, these usages of `eval` are identical to writing these same definitions explicitly and are visible in the “as-imported” method table used by the type checker. The analysis is conservative, however, in that it considers `eval` used in *any* non-top-level context as being “not-top-level.” Some of these methods may be helpers that are only used from the top level at import-time to define methods that are then visible in the as-imported method table; the analysis will consider these usages of `eval` to be not top-level and it thus overapproximates the number of packages that use `eval` below the top level.

(a) Static use of AST forms used by package

(b) Dynamic use of AST forms used by package

Figure 36: AST form usage within `eval`

To validate my static results, I dynamically analyzed 32 packages out of the 186 identified as possibly affected by world age. These packages were selected by randomly sampling 49 packages and then removing packages that did not run, whose tests failed, or that did not call `eval` or `invokelatest` at least once during testing. Over this corpus, the dynamic analysis was implemented by adding instrumentation to record calls to `eval` and `invokelatest`, recording the ASTs and functions, respectively, as well as the stack traces for each invocation.

The results of the static and dynamic analysis of the 32 packages are given in Fig. 36a and Fig. 36b, respectively. Both analysis methods agreed that the most common method table-relevant use of `eval` was to define functions, followed by making function calls and importing other packages. In general, the dynamic analysis was able to identify more packages that used each AST form, as it can examine every AST ran through `eval`, not only statically declared ones. However, this accuracy is dependent on test coverage.

As a result of this analysis, I conclude that the method table that results from evaluating but not using a given package is the same method table that is used at runtime for between 91% and 96% of packages.

5.2.2 *Protocols*

To evaluate the prevalence of protocols in Julia code I performed a small corpus analysis of Julia packages to identify how many protocols they defined and what patterns occurred within those protocol definitions.
I analyzed the base library as well as the top 10 most-starred Julia packages on Github as well as their first-order dependencies. I selected this corpus as many of these packages provide abstraction layers over other systems (such as JuMP.jl, a numerical optimization abstraction library, or DifferentialEquations.jl, an abstraction library over numerical integrators). Moreover, the popularity of these packages indicates that they are widely used and thus that their protocols are important for the broader Julia ecosystem. The full list of root packages is provided in Fig. 37; their first-order dependencies produce a total of 200 corpus packages.
The lack of consistent machine-checkable protocol specifications means that there is no source of truth to check against; the protocols must be discovered from the source code alone. Moreover, there may be no use sites for a given protocol in a given package if the protocol is only intended for external consumption. As a result, the protocol analysis I used for the corpus exploration is differently constructed than the one that the type system utilizes.

DuckDB	Pluto
Flux	IJulia
DifferentialEquations	Genie
JuMP	MakieCore
Gadfly	Turing

Figure 37: Corpus packages

In order to be able to identify protocols from definitions I adopt a simplified, weaker, version of protocols versus the statically checked one. The protocol checker requires that there exists an implementation for every instantiation of the protocol argument typing that returns a value of the correct type. However, checking this is impossible given definitions alone for no protocol argument or return types are available. Instead, my definition of a protocol for the purposes of corpus analysis is a set of methods that:

- have the same name,
- specialize on every subtype (tag, variables are not considered) of some abstract type A in a consistent position i,
- is not implemented for A in position i or a supertype thereof

For example,

```
abstract type A end
struct B <: A end; f(::B) = 1
struct C <: A end; f(::C) = "hi"
```

satisfies this definition as there is a set of methods with name `f` that take every subtype of `A` despite the return types not being consistent. Note that the analysis does not consider varargs functions.

The last component of the definition excludes methods that are implemented abstractly. If I were to define an implementation of `f` that takes any `A` like

```
f(::A) = 9
```

then that implementation is *not* a protocol; it is only a single implementation that can handle multiple cases. As a result, while concrete specializations might exist (as they would in the example of `f`) I exclude these cases.

Figure 38 shows the number of protocols defined by each package. Sets of definitions that satisfy all three of my properties are complete protocols. A set of definitions that have a implementation in the supertype or are missing some implementations are considered partial protocols. Finally, any two methods with the same name that specialize on subtypes of the same concrete type are considered "type-specialized."

I can clearly see that many protocols are defined by these large packages and their dependencies. One example is `DiferentialEquations.jl` (abbreviated as DiffEq) which defines 200 strict protocols. However, this is dwarfed by the number of partial protocols and type specializations detected by the analysis. There may be many more protocols *intended* but that are not properly implemented.

A simple example of an incomplete protocol implementation can be seen in the protocol defined by the MathOptInterface library for the `AbstractFunction` type. The documentation says that every `AbstractFunction` must implement `constant(::AbstractFunction)` that returns the constant component of the function, and *nearly* all implementations do. The protocol is not fully implemented, however, as a type exists for which there is no matching implementation of `constant`: `MathOptInterface.FileFormats.MOF.Nonlinear`. Thus, the

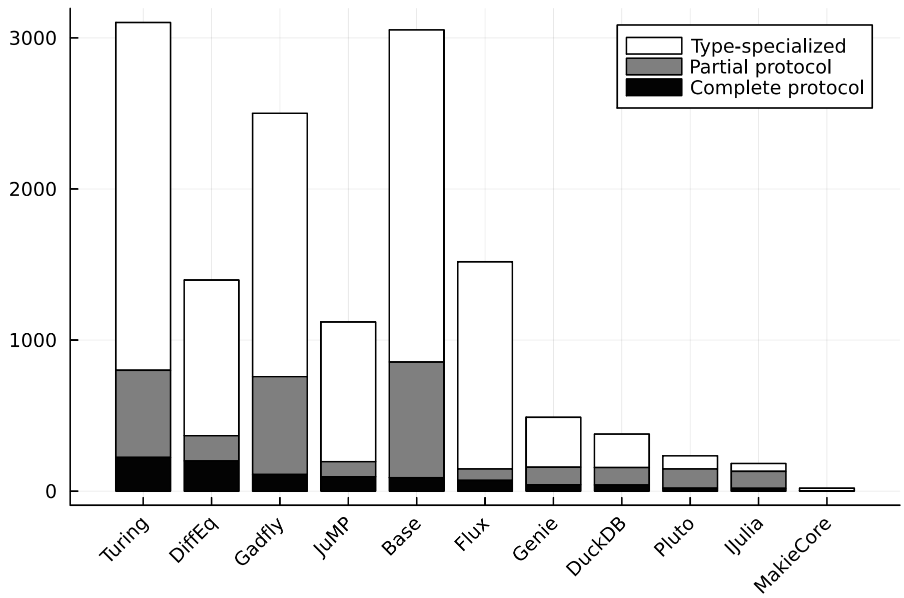


Figure 38: Protocols defined by package

checker categorizes `constant` as a partial protocol since there are concrete subtypes of `AbstractFunction` for which no implementation exists.

Several takeaways can be derived from this observation. In this light the large number of complete protocols is notable; it reinforces the concept that protocols are important for Julia programmers and that they work to maintain them. At the same time, these results emphasize the need for checked protocol declarations due to the number of intended but faulty protocols. Thus, the ability to statically enforce protocol adherence may be useful for Julia package developers and the concept of protocols that I describe here is a common use case for many packages.

5.2.3 *Case Study*

To evaluate how practical the current state of the type system is I considered two case studies:

- A obstacle-avoidance trajectory optimizer for quadcopters based on a numerical optimization algorithm, representing "user" code.
- Julia's `math.jl` that implements several basic math operations (such as logarithms), representing "library" code.

In total the two case studies comprise around 2,500 lines of code.

USER CODE One Julia use case is to write concrete analyses, where specific values and types are known. To evaluate the utility of the type checker on such code, I considered an implementation of the PIPG algorithm [83]. This program determines the flight path of a quadcopter that avoids two obstacles in its path using numerical optimization. I aim to statically type all of the user code; the libraries that the code uses will remain untyped.

The trajectory optimization routine depends on several existing libraries:

- LinearAlgebra, part of Julia's standard library
- StaticArrays, which provides statically-typed sized matrices and vectors
- JuMP, an abstraction library over various numerical solvers
- Plots, to visualize the output trajectories

Program	Functions		Argument annotations		Return types
	Typed	Untypable	Concrete	Abstract	
PIPG	43	2	106	7	3
`math.jl`	21	59	63	83	69
Concrete `math.jl`	77	3	134	12	69

Table 3: Typability of methods in case study programs

Invocations into these untyped libraries are resolved as described earlier using inference.

Type checking the program required both adding argument and return type annotations and removing metaprogramming. I added argument type annotations to all methods based either on the comments for each function or based on their usage in the code; I added return type annotations only to those methods that returned meaningful values.

Metaprogramming is a larger challenge for the practical type checker. The optimizer relies on the JuMP abstraction layer, as mentioned, which introduces a problem description EDSL implemented as macros. A trivial example is

```
@variable(model,x[1:nx,1:N])
```

which binds `x` to be an array of variables of size `nx, N` inside the `model`. I replace introduction forms such as `@variable` with binding forms that introduce variables of the correct types. In the case of `x`, the above macro invocation is replaced with

```
x = nothing::Matrix{JuMP.VariableRef}
```

While the type assertion will fail at runtime, the cast to a matrix of `JuMP.VariableRef`s causes `x` to be correctly statically typed.

The results of modifying and type checking the program are shown in the first column of table 3.

Most functions in this program are typable; only two functions must remain untyped. The majority of fuctions work by mutating their arguments; only three functions actually returned values and thus needed return type annotations.

Of the added type annotations almost all were concrete. The only exceptions were the methods `construct_G` and `construct_H` which initialize new problem definition matrices.

The function `construct_G` is untypable since the line

```
cat([cat(value.(u)[:,t],value.(x)[:,t+1],dims=1) for t=1:N-1]...,dims=1)
```

is rejected as the type checker cannot determine the length of the array being splatted into `cat`. I need to know the length of the arguments in order to statically determine which method might be called. With such a variable-length argument the type checker cannot determine a precise return type.

The second untyped method, `construct_H`, is untypable because of the array index access `B[1]`. The root of the problem is that the type of `B` is the abstract type `Vector{<:SMatrix{M, N, Float64} where {M, N}}`, or a vector of statically-sized matrices of currently indeterminate size whose elements are all `Float64`s. Accessing `B[1]` should clearly return a value that is some instance of `SMatrix{M, N, Float64} where {M, N}`. The Julia type inferencer cannot determine this fact and instead indicates that the return type is `Any`. This imprecision then causes all downstream operations to fail.

The array access `B[1]` going wrong due to imprecise inference is illustrative of the limitations of the type checker. So long as the program is concretely typed (that is, works over Julia tag types) programs can frequently be type checked without needing to modify their implementation even if they use untyped functionality.

Libraries are more challenging than user code, however. Library code is supposed to be generic; it cannot simply say that every argument is concretely typed. I will next examine how readily a small library can be type checked.

LIBRARIES The `math.jl` file was the next target, consisting of Julia's implementation of several basic math functions. As an example, one function provided by `math.jl` is

```
function clamp!(x::AbstractArray, lo, hi)
```

```
    for i in eachindex(x)
        x[i] = clamp(x[i], lo, hi)
    end
    x
end
```

Here, I have an implementation of the `clamp` function generalized to clamp each element of an array to be between `lo` and `hi`. Here I see the main challenge posed when typing library functions: generic or otherwise underspecified arguments.

Abstraction and generic arguments are much more common in libraries compared to user code. The quadcopter knew that the trajectory being optimized was represented as an array of `Float64` values. In contrast, this implementation of `clamp!` knows nothing about what `lo` or `hi` might be at runtime; all it knows is that they are subtypes of `Any` which is not helpful. The only property guaranteed by the original type annotations is that `x` is an `AbstractArray`. Adding more precise types is then difficult without knowing what code uses this function.

A reasonable typing of this `clamp!` function might be

```
clamp!(x::AbstractArray{T}, lo::T, hi::T)::T where T<:Number
```

I specify here that there must be a single concrete subtype of `Number` such that `x` is some sort of array of numbers and that both `lo` and `hi` are also instances of said number.

Making a best-effort attempt to type the methods produces the results seen in the second column of table 3. The input program was already heavily annotated. The majority of added types were return types or specializations of an existing type annotation which are not counted. However, in spite of these annotations, few definitions were statically typeable. The most common reason for a method failing to typecheck was some form of unhandled abstraction.

An example of such abstractions can be seen in the earlier `clamp!` example. Consider, for a moment, what is the type of `i`? It should clearly be the type of the elements produced from `eachindex`, but the documentation says that

> For array types that have opted into fast linear indexing (like Array), this is simply the range 1:length(A). For other array types, return a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and

dictionaries, return an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

As a result, I do not really know what `i` actually is. The only type I can reasonably give `i` is `Any`. This then sets the stage for ensuing chaos with the next problem: `x[i]`.

In Julia, the syntax `x[i]` entails the multimethod call `getindex(x, i)`. In this case, I know that `x` is an `AbstractArrayT`; I only know that `i` is an instance of `Any`. Referring to the Julia documentation for the `AbstractArray` set of protocols, I find that there only needs to be either a method `getindex(A, i::Int)` or a method `nt, Ngetindex(A, I:VarargI)` defined for each `AbstractArray`. Therefore, with our annoyingly-`Any` typed `i` I cannot invoke any of them.

The underlying problem is twofold; one an artifact of the implementation and the other a product of Julia's scale. First, on the implementation end Julia type inference is optimized to work on concrete types and while it *can* work on abstract types, sometimes, it tends towards imprecision. Moreover, Julia's type inference system struggles to deal with type variables, as seen earlier with the `StaticArrays` example, to the point that it simply cannot infer a return type in some cases.

Even if I were to implement a new type inference algorithm that could better handle generically-typed code I run into another problem: representation of abstraction. Protocols, as described earlier, capture the case when there is an implementation of a function that can handle every concrete instantiation of the protocol type. However, as seen in the example of `eachindex`, real Julia code also has much more complicated abstractions then can be represented with protocols alone.

In order to type `eachindex` effectively I need `AbstractArray` to specify its iterator type. I could then treat this as a generic type dependent on the value of `x` and ensure that operations on this type were safe. This approach, effectively a multimethod version of path dependent types [5], would allow this to type check. Similarly, we could introduce a new existential type variable to `AbstractArray` that encodes the type of the iterator. However, both approaches are substantial additions to Julia and would break backwards compatibility.

The practically encountered abstractions are thus much more complex than the simple protocols that my system can check. Moreover, the documentation about what protocols are necessary to implement for an new instance of a type to work *correctly* is frequently lacking. Even in this example—a tiny mathematical function included as part of Julia's own core library—the documented protocols were insufficient for it to type check correctly.

Representation of abstraction is clearly a key problem for type checking Julia. Does `math.jl` pose problems for the type checker besides its use of abstraction, however? To answer this question, I re-annotated `math.jl` with concrete types. For example, the annotations for `clamp!` are now:

```
clamp!(x::Vector{Float64}, lo::Float64, hi::Float64)
```

No additional annotations were added to the generically-annotated version; the only change was that existing generic annotations were replaced with concrete instantiations of each. As seen in the third column of table 3, I can see that the same code when given concrete types generally type checks. Only three methods were then left untyped:

- `literal_pow`, which dispatches on specific concrete values in the type and cannot be practically statically analyzed.
- `hypot`, which splats arguments for a function call and thus the invocation target cannot be resolved.
- `_hypot` (a helper for `hypot`) that uses a first class function and is thus not supported.

The type checker can thus type almost all operations used in `math.jl`, but only when concretely typed.

6 CONCLUSION

In this dissertation I have described a gradual type system for Julia. My approach can provide a strong soundness guarantee for typed code and requires no additional dynamic checks in typed or untyped code so long as no new methods are added with `eval`.

Providing this strong guarantee is a product of both multiple dispatch and Julia's design. Multiple dispatch effectively answers one of the key questions of gradual typing, how to establish type membership, for me. With multiple dispatch a method will only get called with values that are members of its argument types. Moreover, Julia's emphasis on type inference allows me to infer return types for almost any function. These systems then provide assurance about typed arguments and the result of calling untyped methods from within a typed context thereby allowing the elimination of runtime checks.

While Julia's design facilitates this core of a type system, it is missing the primitives needed for typed abstraction. Julia provides no mechanism for developers to declare common functionality between types, related or otherwise. As a result, developers frequently create bugs when types do not implement or improperly implement some expected functionality.

Julia's subtyping was also a considerable challenge. Julia's extensive usage of types is both blessing and curse for static analysis for it also begets a complex subtyping relationship. I showed that subtyping was undecidable, which is not a crippling blow in practice, but suggests that firm theoretical results about subtyping may be hard to come by. My theory treats subtyping (and the related protocol completeness problem) as a "black box," with the implementation using Julia's own subtyping system and a naive completeness checker.

Finally, I described a protocol system that solves part of the abstraction problem. Protocols capture the case where a method exists for every concrete instantiation of some signature. Protocol declarations then statically enforce the existence of suitable methods and can be used by the type checker to let the user call a function they may not otherwise be able to. I additionally showed a basic algorithm for deciding whether a protocol has been implemented or not.

6.1 FUTURE WORK

My type system for Julia is a foundation. As seen in the case studies it is very applicable to programs where concrete types are known but struggles with abstractly typed code. Abstractly typed code stands to benefit the most from static checking, however. The clearest parts of future work are thus in building on top of this foundational type system to support a larger set of the abstractions that have grown up in the Julia community. One simple example is to support a larger swathe of Julia's type language for use in protocol declarations.

PROTOCOLS. The protocol system that I describe is able to capture some of the protocols in Julia code but has several major limitations. In particular, it cannot handle generic types and type variables. The underlying issue is the same as was seen with subtyping: dealing with bounded type variables is hard. A protocol checker for Julia is trivially going to be undecidable; it is easy to see that a protocol checking procedure could be used to decide subtyping (at least under the semantic definition) and thus the same proof applies.

Accepting this undecidability one approach would be to apply one of several more sophisticated pattern matching completeness algorithms. In particular, Lower Your Guards [39] describes a compositional completeness checker that may be able to handle the more general Julia type language. However, several challenges (particularly the number of constructors for some types such as `Any`) make its application to protocol checking not trivial.

Protocols lack generality in one key way, however: they only apply to subtypes of one specific abstract type vector. Frequently users wish to have some common behavior that is shared between otherwise unrelated types. Protocols cannot capture this requirement. Instead, a trait system fits this need better.

TRAITS. The Julia community has been interested in the concept of traits for some time. Traits capture some shared behavior that exists outside of the type hierarchy [70] allowing the programmer to write code that is even more general than the nominal type hierarchy would allow. Julia developers have came up with several ad-hoc ways of defining and using traits (usually by having some method that exists and returns a sentinel value for a type that supports the trait), but none are statically checked.

Designing and implementing a trait system for Julia would further the ability to type abstract code. One problem, for example, is that the seemingly-simple definition of the `+` operator has been of some contention over time leading to there being no consistent implementation for all `Numbers`; in effect, the practice of `+` is that "you know it when you see it." Being able to declare, enforce, and use traits would substantially simplify this problem.

The primary challenge in developing a trait system in Julia is integrating it into multiple dispatch. Users would like to be able to say that an argument is an instance of a trait, which is easy enough. More challenging are cases such as "this array holds only trait implementations" or "this type member is inhabited with subtypes of this abstract trait." A good implementation of traits needs to be efficient, roughly match the expectations of existing programmers, and support a large subset of Julia's type language.

FIRST-CLASS FUNCTIONS. This work does not type check first-class functions for Julia; its support for higher-order functions like `map` is wholly through special-casing and hard coding. Introducing first-class functions (and a suitably specific type) makes the key problem of gradual typing much harder in a multiple dispatch setting. Consider, for a moment, that you have

```
f(x::Function{String, Int})::Int = x("hello") + 2
f(x::Function{String, String})::String = trim(x("world"))
```

and then you call it with `f(x -> x)`. How do you decide which implementation of `f` should be called without having to do deep analysis of the lambda?

The multiple dispatch setting makes many gradual typing approaches impractical. The behavioral semantics, for example, would tell us that we should wrap `x` in a proxy object that enforces the type I expect on it. That is great, but does not tell us *which* of these implementations I should call in the first place. Other gradual type systems have analogous problems as they rely on knowing what type the higher-order functionality *should be* when the dispatch system needs to know what it *is*.

One potential solution would be to only dispatch on typed function. This approach is straightforward and would be expressible within Julia's existing type language. Requiring typed lambdas would, however, prevent untyped code from being able to invoke specific typed implementations.

BIBLIOGRAPHY

[1] Riposte: A trace-driven compiler and parallel vm for vector code in r. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[2] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *OOPSLA*, 1991.

[3] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1993.

[4] Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy Steele. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2011.

[5] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *ACM Sigplan Notices*, 2014.

[6] Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. Parallelizing julia with a non-invasive DSL. In *European Conference on Object-Oriented Programming (ECOOP)*, 2017.

[7] Roberto Bagnara, Abramo Bagnara, and Patricia M Hill. Coding guidelines and undecidability. *arXiv preprint arXiv:2212.13933*, 2022.

[8] Julia Belyakova. Decidable tag-based semantic subtyping for nominal types, tuples, and unions. In *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs FTFJP*, 2019.

[9] Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. World age in julia: optimizing method dispatch in the presence of eval. In *OOPSLA*, 2020.

[10] Jeff Bezanson. *Abstraction in technical computing*. PhD thesis, Massachusetts Institute of Technology, 2015.

[11] Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: Dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[12] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2009.

[13] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, 1986.

[14] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *European Symposium on Programming (ESOP)*, 2016.

[15] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Symposium on Principles of Programming Languages (POPL)*, 1994.

[16] Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP'*, 1992.

[17] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. In *Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA)*, 1994.

[18] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. 1995.

[19] John Chambers. Object-oriented programming, functional programming and r. *Statistical Science*, (2), 2014.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. 2005.

[21] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017.

[22] Maxime Chevalier-Boisvert, Laurie J. Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In *Conference on Compiler Construction (CC)*, 2010.

[23] Benjamin Chung and Paley Li. Towards typing Julia. In *The -2th Workshop on New Object-Oriented Languages (NOOL)*, 2017.

[24] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. Kafka: Gradual typing for objects. In *ECOOP*, 2018.

[25] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[26] Robert P. Cook and Insup Lee. Dymos: A dynamic modification system. In *Proceedings of the Symposium on High-Level Debugging*, 1983.

[27] Luiz De Rose and David Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2), 1999.

[28] Linda DeMichiel and Richard Gabriel. The Common Lisp Object System: An overview. In *European Conference on Object-Oriented Programming (ECOOP)*, 1987.

[29] David Detlefs and Ole Agesen. Inlining of virtual methods. In Rachid Guerraoui, editor, *ECOOP*, 1999.

[30] Facebook. Hack, 2016. `http://hacklang.org`.

[31] Mike Fagan. *Soft typing: an approach to type checking for dynamically typed languages*. PhD thesis, Rice University, 1991.

[32] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.*, 2(POPL), 2018.

[33] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Symposium on Logic in Computer Science (LICS)*, 2002.

[34] Miguel Garcia, Francisco Ortin, and Jose Quiroga. Design and implementation of an efficient hybrid dynamic and static typing language. *Softw. Pract. Exper.*, 46(2), 2016.

[35] Yossi Gil and Ori Roth. Fling-a fluent api generator. In *ECOOP*, 2019.

[36] Neal Glew. Method inlining, dynamic class loading, and type soundness. *Journal of Object Technology*, 4(8), 2005.

[37] Neal Glew, Jens Palsberg, and Christian Grothoff. Type-safe optimisation of plugin architectures. In *International Static Analysis Symposium*, 2005.

[38] Isaac Gouy. The computer language benchmarks game, 2018.

[39] Sebastian Graf, Simon Peyton Jones, and Ryan G Scott. Lower your guards: a compositional pattern-match coverage checker. *Proceedings of the ACM on Programming Languages*, (ICFP), 2020.

[40] Radu Grigore. Java generics are turing complete. In *Symposium on Principles of Programming Languages (POPL)*, 2017.

[41] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. 2012.

[42] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[43] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[44] Zhong Sheng Hu. Decidability and algorithmic analysis of dependent object types (dot). Master's thesis, University of Waterloo, 2019.

[45] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for r. In *Conference on Virtual Execution Environments (VEE)*, 2014.

[46] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, 2007.

[47] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Symposium on Code Generation and Optimization f(CGO)*, 2004.

[48] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, 1998.

[49] Vassily Litvinov. Constraint-based polymorphism in cecil: Towards a practical and static type system. In *Addendum to Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA Addendum)*, 1998.

[50] Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Addendum to the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[51] Vassily Litvinov. Contraint-based polymorphism in cecil: Towards a practical and static type system. In *OOPSLA*, 1998.

[52] Vassily Litvinov. *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington, 2003.

[53] Francesco Logozzo and Herman Venter. Rata: Rapid atomic type analysis by abstract interpretation – application to javascript optimization. In Rajiv Gupta, editor, *Compiler Construction*, 2010.

[54] Miles Lubin and Iain Dunning. Computing in operations research using julia. In *INFORMS Journal on Computing*, 2013.

[55] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[56] MATLAB. *version 9.4*. The MathWorks Inc., Natick, Massachusetts, 2018.

[57] JACOB MATTHEWS and ROBERT BRUCE FINDLER. An operational semantics for scheme. *Journal of Functional Programming*, 18(1):47–86, 2008.

[58] John McCarthy. History of lisp. In *History of programming languages (HOPL)*, 1978.

[59] Hannes Mehnert. Extending dylan's type system for better type inference and error detection. In *Proceedings of the 2010 international conference on Lisp*, pages 1–10, 2010.

[60] Microsoft. TypeScript – language specification, 2016.

[61] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, (OOPSLA), 2017.

[62] Trung Nguyen, Junjie Gu, and Zhiyuan Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In Chua-Huang Huang, Ponnuswamy Sadayappan, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, 1996.

[63] Michael Paleczny, Christopher Vick, and Cliff Click. The java HotSpot server compiler. In *Symposium on Java Virtual Machine Research and Technology (JVM)*, 2001.

[64] Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu. Polymorphic symmetric multiple dispatch with variance. (POPL), 2019.

[65] Benjamin C. Pierce. Bounded quantification is undecidable. In *Symposium on Principles of Programming Languages (POPL)*, 1992.

[66] Benjamin C Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 305–315, 1992.

[67] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Symposium on Dynamic Languages*, DLS, 2012.

[68] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008.

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

[70] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *ECOOP*, 2003.

[71] Olin Shivers. Data-flow analysis and type recovery in scheme. In *Topics in Advanced Language Implementation*, pages 47–88. MIT Press, 1990.

[72] Jeremy Siek. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006. http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf.

[73] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, 2015.

[74] Guy Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (sun HPCS language). In *Encyclopedia of Parallel Computing*, pages 718–735. 2011.

[75] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[76] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *POPL*, 2016.

[77] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in java's type system. In *PLDI*, 2011.

[78] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Symposium on Dynamic languages (DLS)*, 2006.

[79] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*, 2008.

[80] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *Programming Languages and Systems (ESOP)*, 2009.

[81] Andrew Wright and Robert Cartwright. A practical soft type system for scheme. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 250–262, 1994.

[82] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.

[83] Yue Yu, Purnanand Elango, and Behcet Acıkmese. Proportional-integral projected gradient method for model predictive control. *IEEE Control Systems Letters*, 2020.

[84] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: A rational reconstruction. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.