Approximating Type Stability in the Julia JIT (Work in Progress)

Artem Pelenitsyn
pelenitsyn.a@northeastern.edu
Northeastern University
USA

Abstract

Julia is a dynamic language for scientific computing. For a dynamic language, Julia is surprisingly typeful. Types are used not only to structure data but also to guide dynamic dispatch – the main design tool in the language. No matter the dynamism, Julia is performant: flexibility is compiled away at the run time using a simple but smart type-specialization based optimization technique called type stability. Based on a model of a JIT mimicking Julia from previous works, we present the first algorithm to approximate type stability of Julia code. Implementation and evaluation of the algorithm is still a work in progress.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

Keywords: method dispatch, type inference, compilation,dynamic languages, Julia language

ACM Reference Format:
Artem Pelenitsyn. 2023. Approximating Type Stability in the Julia JIT (Work in Progress). In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3623507.3623556

1 Introduction

The scientific computing community has long been of two minds about the language technology best suited for the task. On the one hand, the exploratory nature of programming with raw datasets and mathematical models with many parameters calls for modern convenience features like dynamic typing and garbage collection. On the other, crunching numbers may require to squeeze every CPU cycle available. Accordingly, the language market in this area is split into productivity languages (Python, MATLAB, R) and high-performance oriented languages (C, C++, Fortran).

The split nature of the language landscape in the area of scientific computing calls for a question: can we have the best of both worlds? Julia attempts to answer in the positive.

On the surface, the Julia language has a lot of pythonistic feeling to it. Consider the following function that sums an array of elements but clamps the summands above the given threshold.

```
function sum(v, t)
  res = v[1]
  for i = 2:length(v)
    elm  = v[i] < t ? v[i] : t
    res = res + elm
  end
  res
end
```

Notice the absence of type annotations either on the function arguments or anywhere locally. Of course, it is possible to call `sum` with a string and a number and get a run-time failure. On the other hand, and no less of a surprise, when called with the right arguments the code gets JIT-compiled and performs on par with equivalent C code.

Julia's competitive performance is documented both in the language manual and in academic papers [2]. One of the key techniques enabling essential optimizations is based on the code property called *type stability* [5]. While well-understood on the IR level, the property has been elusive for end users: it is mentioned in the manual and on many forum threads but still does not have a clear source-level model for it. In this short paper we propose the first algorithm to approximate this highly dynamic property statically. In particular, we

- give an informal description of type stability as it comes up in the practice of Julia programming (Sec. 2; largely following [5]);
- discuss challenges to model type stability statically and the relation of the task to the full-fledged type inference (Sec. 3);
- give an algorithm to approximate type stability statically (Sec. 4);
- remark on implementation (Sec. 5) and evaluation of the algorithm (Sec. 6) — both work in progress.

VMIL '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0401-7/23/10…$15.00
https://doi.org/10.1145/3623507.3623556

2 A Type Stability Primer

2.1 Multiple Dispatch in Julia

The Julia language is designed around multiple dispatch [3]. Programs consist of *functions* that are implemented by multiple *methods*; there is nothing more to a Julia function than just a name. Each method is identified by a distinct type signature. At run time, the Julia implementation dispatches a function call to the *most specific* method by comparing the types of the arguments to the types of the parameters of all methods of that function. For example, the call to the `+` function in the example from the introduction can dispatch to one out of over two hundreds methods for `+` in the standard library alone (packages can add more methods to a function).

Julia supports a rich type language for defining method signatures. Base types consist of either bits types (i.e. types with a direct binary representation, like integers) or record types (called structs in Julia). Both bits types and record types, referred to as *concrete types*, can have supertypes, but all supertypes are *abstract types*. Abstract types are arranged into a subtyping hierarchy rooted at the built-in `Any` type. Every value in the program has a unique type tag that can be accessed via the `typeof` function. The type language allows for further composition of these base types using unions, tuples, and bounded existential constructors; the result of composition can be abstract or concrete. Zappa Nardelli et al. [6] gives a detailed discussion of the type language and of subtyping.

Any function call in a program, such as `res+elm` in the example above, requires choosing one of the methods of the target function. *Method dispatch* is a multi-step process. First, the implementation obtains the concrete types of arguments. Second, it retrieves applicable methods by checking for subtyping between argument types and type annotations of the methods. Next, it sorts these methods into subtype order. Finally, the call is dispatched to the most specific method—a method such that no other applicable method is its strict subtype. If no such method exists, an error is produced.

Function calls are pervasive in Julia, and their efficiency is crucial for performance. However, the multi-step dispatch mechanism make the process slow. To attain acceptable performance, the compiler attempts to remove as many dispatch operations as it can. This optimization leverages run-time type information whenever a method is compiled, i.e., when it is called for the first time with a novel set of argument types. These types are used by the compiler to infer types in the method body. Then, this type information frequently allows the compiler to devirtualize and inline the function calls within a method [1], thus improving performance. However, this optimization is not always possible: if type inference cannot produce a sufficiently specific type, then the call cannot be devirtualized.

2.2 Type Stability: A Key To Performance?

To illustrate a profound effect that type inference precision can have on performance, consider the `sum` function from the introduction benchmarked in three scenarios differing only in input types. Assuming a random array of integers called `vint` and a random array of floating-point numbers called `vflt` (both consisting of 10K elements) compare the median running time of the following three calls to `sum`:

- `sum(vint, 0)` — 1.397 microseconds
- `sum(vflt, 0.5)` — 11.489 microseconds
- `sum(vint, 0.5)` — 64.623 microseconds

When a performance regression occurs, it is common for Julia developers to study the intermediate representation produced by the compiler. To facilitate this, the language provides a utilty, `code_warntype`, that shows Julia's intermediate representation of the code along with the inferred types for a given function invocation. Types that are imprecise, i.e. abstract, show up in red: they indicate that concrete type of a run-time value may vary from run to run. In the first two benchmarks, the Julia compiler is able to deduce a concrete return type of the method (`Int` and `Float` correspondingly), but the type of the last one reported as `Union{Int, Float}`, which is an abstract type. Such type imprecision can impact performance in two ways. First, the `res` variable has to be boxed, adding a level of indirection to any operation performed therein. Second, it is harder for the compiler to devirtualize and inline consecutive calls, thus requiring dynamic dispatch.

Julia's compilation model is designed to accommodate source programs with flexible types. Yet, to make such programs efficient, the compiler creates an *instance* of each source method for each distinct tuple of argument types. Thus, even if the programmer does not provide any type annotations, like in the `sum` example, the compiler will create method instances for *concrete* input types seen during an execution. For example, the three benchmarks shown above will make the compiler create three distinct method instances. Because method instances have more precise argument types, the compiler can leverage them to produce more efficient code and infer more precise return types.

In Julia parlance, a method is called *type stable* if, given a concrete input type, it is possible to infer a concrete output type. The `sum` function is not type stable because for the input type `Tuple{Vector{Int}, Float64}`[1] the return type can be either `Int` or `Float64`.

Pelenitsyn et al. [5] formalize type stability as it relates to program *execution* by building a formal model of a type-specializing just-in-time compiler that does its main job at run time. In this work, we set to approximate the property of type stability for arbitrary Julia code statically, without running the code in question.

[1] Tuple types encode several input arguments in Julia.

3 Inferring Type Stability Versus Inferring Types

A natural idea for inferring type stability in Julia would be to formulate it as a forward static analysis: being an abstract or concrete type is one bit of information that has a known value at the input (concrete) and should be propagated to the output, possibly changing on the way.

To test the static analysis idea, consider a positive example first: the identity function.

```
function id(x)
  x
end
```

It is straightforward to infer that, given any concrete input type, the return value is also concretely typed: the one bit of information carries over to the result in one step.

However, another example—the increment function—shows that the task quickly becomes unwieldy.

```
function inc(x)
  x + 1
end
```

Concreteness of the result returned by `inc` depends on concreteness of the result of the call to `+`. In turn, the property of the return type of `+` depends on which `+` method Julia will dispatch to at run time. There are about two hundred method implementations of `+` in the standard library alone, and packages add more. Some of those methods are type stable (e.g. `+(::Int64,::Int64)`), and some of them are not (e.g. `+(::Rational{Bool},::Rational{Bool})` [2]). Therefore, to infer the property of interest, in general, we need to predict which methods are selected at run time.

The `inc` example shows that inferring type stability of Julia code requires reasoning about multiple dynamic dispatch, which leads to reasoning about the *types* of intermediate values rather than only the concreteness bit. But if there was a tool for computing type information beforehand, a special-purpose analysis for type stability would not be needed: it suffices to ask the tool for the type of the return value and check if that type is concrete. This observation leads to the following conjecture:

Conjecture 3.1. *Inferring type stability of a Julia method statically is no easier than performing type inference of that method.*

[2]The reason for the `+(::Rational{Bool},::Rational{Bool})` method to be type unstable is not important, but in a nutshell, Julia has made a questionable design decision about the return type of `+(::Bool,::Bool)`, which in the current implementation is `Int` (see discussion https://github.com/JuliaLang/julia/issues/19168), and when adding two rational numbers with boolean components, depending on the values of the summands, you get back either `Rational{Bool}` or `Rational{Int}`.

A complete type inference algorithm would allow for checking type stability of Julia code. But should type inference be implemented from scratch? There are two reasons to not go this way.

1. It is not clear that inferring types for source-level Julia code without changing anything in the language can yield a meaningful result (more on this see [4]). For instance, in the `inc` example, a sound return type cannot be much better than `Any`.
2. Julia already has a built-in type inference engine, which is modeled as a black box in [5]. The engine is used for code optimizations. Thus, analyzing type stability based on a custom type inference algorithm can produce results that diverge from Julia, misleading the users about potential optimizations. This would be of limited usage for Julia users.

4 An Algorithm To Approximate Type Stability

If our predictions for type stability are to align with the Julia implementation, the analysis should closely model Julia's run-time behavior, as described in [5]. The type-specializing JIT-compiler from [5] makes optimization decisions based on *concrete input types* with the help of Julia's type inference engine. Therefore, the algorithm for predicting these decisions statically considers all (or as many as possible) allowed concrete input types of a method. Fig. 1 describes this algorithm at a high level.

Let us consider every step of the algorithm described on Fig. 1 and explain its meaning using an example. The list below also assigns the numbers to each step in the algorithm.

1. The input of the algorithm is a Julia method. Methods in Julia are represented by run-time objects of type `Method` and can be manipulated as all other objects (e.g. stored in collections, responding to field accesses, etc.). For example, consider the `length` method from Julia's standard library. We can get the corresponding `Method` object using the standard `@which` macro applied to an application of the `length` method. This can be done in the Julia REPL (signified by the `julia>` prefix).

   ```
   julia> @which length([1,2,3])
   length(a::Array) in Base at array.jl:215
   ```

   The output shows that the given `length`-call will dispatch to the method defined in the `Base` module (Julia-speak for the standard library). The output also shows the location of the method in the standard library and, most importantly for us, the signature of the method. In fact, what we see here is a pretty-printed representation of the `Method` object representing a particular Julia method.

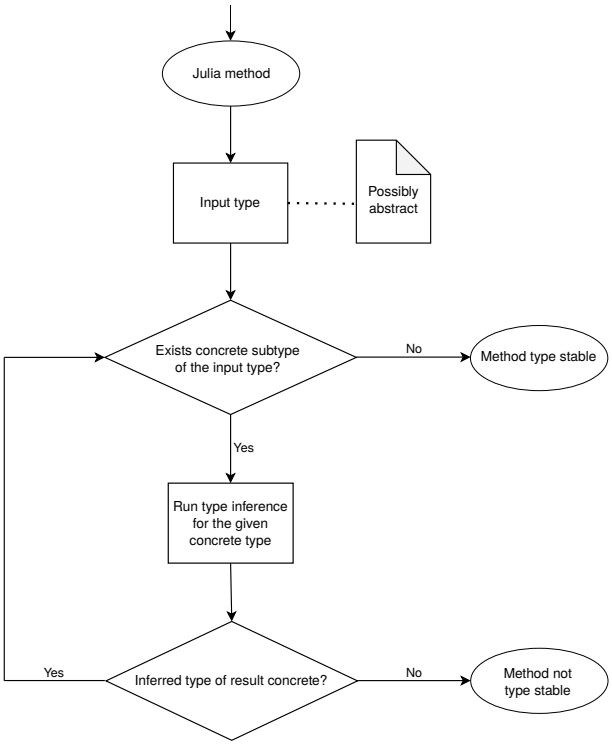

Figure 1. Inferring type stability of a Julia method

2. The first task of the algorithm is to get the input type of the given method. This is possible through querying the `sig` field of the method object.
   Building on the example above, we can get the signature of the `length` method as follows:

```
julia> m = @which length([1,2,3]);

julia> m.sig
Tuple{typeof(length), Array}
```

   A signature of a method contains the special singleton function type (`typeof(...)`) as the first component, and the rest is (easy to convert to) the type of the input — an n-tuple. In this example, the type of the input is 1-tuple, consisting of the existential array type `Array{T, N} where T where N` abbreviated simply as `Array`[3].
3. The input type can be either concrete, which, in Julia, means that there can be no proper subtypes of that type, or abstract. In either case, the choice on the current step will enter the loop at least once, because for concrete input type, the check holds once trivially (e.g. there is exactly one concrete subtype of the concrete type `Int` — it is `Int` itself).

[3]A user can always look under the abbreviation using the `dump` function.

```
julia> code_typed(m.sig.parameters[1].instance,
                  (Array{Float64, 1},),
                  optimize=false)
1-element Vector{Any}:
 CodeInfo(
1 - %1 = Base.arraylen(a)::Int64
+--      return %1
) => Int64
```

Figure 2. Running Julia's built-in type inferencer

   If the input type is abstract, we need a procedure enumerating all concrete subtypes of it. An implementation for this procedure is discussed in the next section, but it suffices to treat is as a black box for now.
   In the case of the `length` method, the input type, `Array`, is an existential type and hence abstract. Therefore, the enumeration procedure should have yielded a concrete subtype of `Array`. Assume that the concrete type is `Array{Float64, 1}`.
4. Running Julia's type inference for a given method and a given concrete input type is done by calling Julia's standard `code_typed` function. The only issue with the function is that it expects a function object as a part of the input, not a method object. But getting from a method to the corresponding function is possible using the signature field discussed above, and, in particular, the singleton function type contained in the first component of the `sig` field: accessing the single function object using the function type is possible via the `instance` field.
   Running type inference for the `length` method and the concrete input type `Array{Float64, 1}` could be done as shown on Fig. 2. The return value is an array of `CodeInfo` objects that represent the type-annotated method bodies of all methods that a call with the given input type could dispatch to (for a concrete input type and no ambiguities in method definitions, the resulting array always contains exactly one element). Method bodies are transformed into a lower-level intermediate representation. In the running example, the method body contains a single call to an intrinsic Julia function that is known to return a value of type `Int64`.
5. Concreteness of the inferred return type is checked with the standard Julia `isconcretetype` predicate. In the running example, the return type of `length` is inferred to be `Int64` for the concrete input type `Array{Float64, 1}`. Since `Int64` is a concrete type, following the second decision element on Fig. 1 brings us back to the start of the loop. After that, we try another concrete subtype of the input type, if there is any.

5 Implementation

The first practical consideration of the algorithm as shown is termination. It is clear that the algorithm does not terminate for some inputs. Consider the `length` example: its input type is the existential `Array` type, which means that possible concrete input types may be:

- `Array{1, Int}`
- `Array{1, Array{1, Int}}`
- `Array{1, Array{1, Array{1, Int}}}`
- etc.

The issue of termination is close to another one: a search space blow up. In the standard library alone, there are over five hundreds immediate subtypes of `Any` and every concrete subtype of those can be use to instantiate the `Array` type when inferring stability of `length`. Although finite, this space can be simply too large to exhaust in a reasonable time.

Our intention is to develop certain heuristics to perform an early termination of the algorithm. The simplest heuristic of this sort is to employ a fuel parameter that gives an arbitrary upper bound for the number of steps we are allowed to perform before we give up.

Another implementation concern is step 3 where we need to generate a concrete subtype of the input type. Although there is no built-in facility for this task, Julia allows getting immediate subtypes of a given type using the `subtypes` method. As our current implementation shows, with enough care, it is not hard to enumerate concrete subtypes by applying the `subtypes` method iteratively.

6 Evaluation

Pelenitsyn et al. [5] analyzed the type stability of a corpus of open-source Julia packages dynamically via executing test suites of the packages and inspecting the resulting method instances collected from the internal state of the virtual machine. We propose to run our algorithm for statically inferring type stability over the same packages and match the results.

Acknowledgements

Jan Vitek showed me how exciting Julia is and suggested the topic of type stability in general and its approximation in particular later on. Jan Jecmen joined in building the prototype and brought several deep insights (backed by pull requests) in how to improve it. Julia Belyakova and VMIL '23 anonymous reviewers helped to improve the text.

References

[1] Gerald Aigner and Urs Hölzle. 1996. Eliminating Virtual Function Calls in C++ Programs. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1.1.7.7766

[2] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

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

[4] Benjamin Chung. 2023. *A Type System for Julia.* Ph.D. Dissertation. Northeastern University.

[5] Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 150 (2021), 26 pages. https://doi.org/10.1145/3485527

[6] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276483

Received 2023-07-23; accepted 2023-08-28