Static Typing Without Static Types — Typing Inheritance from the Bottom Up

Benjamin Chung Paley Li Jan Vitek
Northeastern University
bchung@ccs.neu.edu {pa.li,j.vitek}@neu.edu

Abstract

Julia is an untyped imperative programming language designed for scientific computing. Despite being untyped, Julia provides a rich runtime type system that includes features such as inheritance, but lacks the mechanisms to ensure compliance with interfaces. We propose a static type system for a subset of Julia, called `Jolt`, ruling out functional interface mismatches by synthesizing abstract interfaces from concrete implementations. `Jolt` can rule out some type errors in existing code without any new annotations, providing additional safety for free.

1. Introduction

Traditional statically typed object-oriented languages have a series of common idioms: single dispatch [7], figuring out which method to use in which situation, interfaces [4, 11], to abstract over common means of access, and the means to statically ensure that those interfaces are adhered to with the correct dispatch call.

These practices are perfectly suitable for many contexts, as is demonstrated by the success of Java [7], C# [9], and C++ [13], among others. These features are not universally applicable, however. Languages such as Javascript [6] and Lua [10] have no mechanism to define enforced interfaces, but commonly use documentation-defined interfaces to define behavior for classes of objects [10].

```
1  abstract A1
2  type C1 <: A1 end
3  type C2 <: A1 end
4  type C3 <: A1 end
5
6  function a(arg::C1) print("Hello ") end
7  function a(arg::C2) print("World\n") end
8  function problem(arg::A1)
9    return a(arg)
10 end
11
12 problem(C1()) # Hello
13 problem(C2()) # World
14 problem(C3()) # error

ERROR: MethodError: no method matching a(::C3)
Closest candidates are:
  a(::C1)
  a(::C2)
```

Figure 1. Object inheritance example in Julia.

Julia is another such programming language, as it provides multi-method dispatch and an interface system that focuses on an abstract struct-like construct. Julia was originally designed for the purpose of scientific computation, in the vein of R or MATLAB[2], containing a number of features designed specifically to support numeric computation and other tasks common in scientific programs.

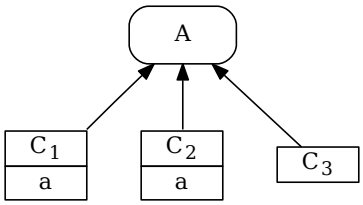

Figure 2. Inheritance hierarchy diagram for Figure 1.

2. Julia

From the perspective of object-oriented language design, Julia has several interesting features:

- Julia is *dynamically typed*, and has no mechanism for statically checking type correctness. However, as illustrated in Figure 1, Julia code does have many types.
- The purpose behind all of these types is *dispatch*. Julia provides full multi-method dispatch based on *runtime type tags*. This lets programmers write code that is highly specified for a specific value.
- Despite having types with methods that operate over them, Julia does not allow explicit procedural interfaces, a key feature of traditional object systems. Julia's interfaces are called *abstract types*, and they define no explicit methods, instead, the abstract types rely upon "a collection of informal interfaces" [1] to abstract over implementations.

Figure 1 provides an illustration of how these features interact, and how they can be used to do untyped object-oriented programming. The diagram begins by constructing the object hierarchy shown in Figure 2, as well as a function `problem`, which calls the function `a` on an argument of type A.

The next step is to actually call the methods we have defined. Julia performs dispatch by looking for the *most specific method* whose arguments are satisfied by the given value - in essence, providing a type guarantee that the argument will always be of the declared type. In this way, the implementation of `a` on line 6 is called when we call `a` with an instance of C_1 on line 13, and the implementation of `a` on line 7 is called when C_2 is passed on line 13.

In this manner, and if we ignore C_3 briefly, the programmer has effectively built a functional interface. Any `A` will have an `a` method associated with it, and we can use `a` on a `A` with complete assurance that it will actually exist. As a result, we have built an interface by abstracting directly from the concrete implementations.

However, the assumption that `a` is always safe on an `A` is not true in this actual program, once we add C_3 back in. Julia does not

$$
\begin{aligned}
t &::= C \mid s \\
s &::= A \mid \texttt{any} \\
d &::= \texttt{abstract}\ A <: s \mid \texttt{type}\ C <: s \\
&\quad \mid m(a :: t,\ ...) = e \\
e &::= x \mid \texttt{new}\ C() \mid m(e,\ ...)
\end{aligned}
$$

Figure 3. Static syntax for `Jolt`.

$$
\textsc{TAbsSelf}\ \frac{m(..., a :: A, ...) \in [\![A]\!]}{m(..., a :: A, ...) \Subset A} \qquad \textsc{TConSelf}\ \frac{m(..., a :: C, ...) \in [\![C]\!]}{m(..., a :: C, ...) \Subset C}
$$

$$
\textsc{TAbsVirtual}\ \frac{\forall\, C <: A : m(..., a :: C, ...) \in [\![C]\!] \quad \forall\, A' <: A : m(..., a :: A', ...) \Subset A'}{m(..., a :: A, ...) \Subset A}
$$

Figure 4. Method enclosure over types.

impose any constraints on the types in a program, so C_3 is perfectly valid while lacking `a`. As a result, line 15 produced the listed error, despite the argument to `problem` being type correct.

The error is easily spotted in this example, because all of the type definitions are simple and locally defined. However, in real Julia programs, types can be imported from other files and exist in more complex hierarchies. Therefore, a functional interface could be violated by a library, resulting in errors that are difficult to detect ahead of time.

3. Jolt

An interesting observation about the issue identified in Figure 1 is that it can be seen as a "message not understood" error, exactly the kind that type systems are widely applied to detect and prevent. However, untyped languages are uniquely difficult to type [5], as complex inference is typically required and the idioms are difficult to track with types.

Languages, such as Rust, with traits [12], and Haskell, with typeclasses [8], have statically checked interfaces for multimethods, but cannot handle "orphan" implementations - a common idiom in Julia. As a consequence, we need to use a new approach.

Despite being untyped, Julia code uses a lot of type annotations, though they are not used statically. We propose a type system called `Jolt` for *existing* Julia code that can statically infer interfaces and detect "method not found" errors, over a heavily pared down version of Julia. `Jolt` does not introduce any new types to Julia, instead using the type annotations, which already exist in Julia code, for static typing. In the vein of optional typing [3], `Jolt` statically type checks the program without adding any new syntax or semantics, while detecting dynamic errors statically.

`Jolt` formalizes a minimal subset of Julia. In Figure 3, we present the entire syntax for `Jolt`, which consists of all types (t), heritable types (s), declarations (d), and expressions (e). Types in `Jolt` are either a name, which can be the name of an abstract type (A) or a concrete type (C), or the `any` keyword, which denotes the top type. The declarations of `Jolt` declare abstract types, concrete types, and methods. The three expressions in `Jolt` are local variables, object creation, and method invocation.

Due to the concise nature of this formalism, the expression typing and operational semantics rules for `Jolt`have been omitted, as they are straightforward and does not offer much insight into the inheritance structure of Julia. Instead, we will focus our attention on highlighting how `Jolt` generates and ensures correctness of the inheritance hierarchy created from its abstract and concrete types.

In Figure 4, we present our rules for when methods are enclosed and/or inside a type. The symbol $\in$ denotes the standard set notion for an element being inside a set. In `Jolt`, this means the method is syntactically defined for the type it is in. The symbol $\Subset$ denotes a method being enclosed inside a type. In `Jolt`, a method is enclosed in a type if that method exists in the inheritance hierarchy, but there might not necessarily exist an actual implementation of that method in the enclosing type. It is important to note the difference between $\Subset$ denoting the semantical relation of methods inside a type that represents its place on the inheritance structure, while $\in$ denoting when a method is syntactically defined for a type.

The *TAbsSelf* rule describes when an enclosing method is defined inside that type. For concrete types, their enclosing methods are always defined inside themselves, as reflected by the *TConSelf* rule. The *TAbsVirtual* rule describes the case when an enclosing method in an abstract type is not inside that abstract type, which means its types must have this method inside them and that all abstract types below this abstract must have this method enclosed within them.

An abstract type is considered correct when three separate components are shown. The first component requires the name of the type to be well-formed, the second component requires every method in the type to be well-formed, and the final component requires every method in the type to be enclosed within that type. Three similar components are required to show a concrete type is correct.

The key to the implementation of this static type system is computing the enclosure relation to satisfy the requirements laid out in Figure 4. We propose the straightforward approach, where methods enclosed in an abstract type A are computed by taking the intersection of all abstract subtypes A' and concrete subtypes C, and combined with the methods that are defined for A itself.

This bottom-up approach to producing interfaces is the dual of the typical mechanism for ensuring that classes actually implement their declared interface. In a more traditional setting, we go from top-to-bottom, checking that the children implement a strict superset of the methods on the parent, while our approach ensures that the parent has a subset of the methods on the children.

4. Conclusion

Julia provides an interesting alternative to traditional functional interface definition, whereby the methods of an interface are solely defined by the concrete implementations of that interface, instead of having the interface specifying the methods of its concrete implementation. Despite being untyped, we are able to utilize this property to create and reason statically about an inheritance hierarchy within Julia. We demonstrate this approach in `Jolt` , a minimal subset of Julia. `Jolt` provides static type checking for safe calls of multi-method.

Our approach towards `Jolt` originated from the typed runtime of Julia, which has the potential to provide the basis for other static analyses. Future work could include extending the type system to encompass the rest of Julia, as well as handling dynamic behavior such as reflection and dynamic loading.

References

[1] Julia - interfaces. http://docs.julialang.org/en/release-0.4/manual/interfaces/.

[2] Jeffrey Werner Bezanson. *Abstraction in technical computing*. PhD thesis, Massachusetts Institute of Technology, 2015.

[3] Gilad Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages*, 2004.

[4] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 457–467, New York, NY, USA, 1989. ACM.

[5] Brett Cannon. *Localized type inference of atomic types in python*. PhD thesis, California Polytechnic State University San Luis Obispo, 2005.

[6] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems)*, 1999.

[7] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[8] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. In *European Symposium On Programming*, pages 241–256. Springer, 1994.

[9] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[10] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[12] Eric Reed. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.