Towards Typing Julia

Ben Chung
Northeastern University

Paley Li
Czech Technical University

Abstract

Julia is a dynamic programming language designed for scientific programming. Unlike traditional dynamic languages, Julia contains optional type annotations for certain expressions, which are used primarily to type function arguments. To statically type Julia, it would be advantageous to use the user-provided type annotations to infer any missing types. In this abstract, we present an algorithm for type inference of functions (and variable assignment) in Julia, a first step towards statically typing Julia.

Keywords Multi-dispatch, gradual typing, type inference, pattern matching

1 Introduction

Julia presents a paradox of different typing paradigms. Julia's compiler produces no static type errors, only 'method not found' errors at runtime, a hallmark of dynamic typing. However, like a statically typed language, Julia allows programmers to write types in function signatures, and Julia's dispatch mechanism guarantees that declared types are sound. This would be an ideal opportunity for type inference, but for Julia's use of multi-dispatch, popularized by the Cecil language [2], to solve the expression problem [5]. This feature allows Julia libraries to expose fluent and natural interfaces that are defined exclusively through documentation, unlike Cecil, where external type checking guaranteed adherence to interfaces, complicating type checking.

In this abstract, we outline an approach for type checking Julia code, combining a bidirectional type system [3] with a dispatch resolution mechanism based on pattern matching. This approach allows us to handle several of the key difficulties in type checking Julia code, including parametricity and tuples.

2 Checking Calls

Invocations make up virtually all functionality in Julia, even fundamental operations such as addition or indexing are function calls. A Julia invocation will be dispatched to the method with the *most specific* type matching the argument values. During this stage two types of errors can occur, if there is no implementation for the value passed, or if there are multiple equally specific (ambiguous) implementations.

The first kind of error is eliminated by our definition of type soundness, whereby an implementation must exists for every well-typed function call. To ensure this guarantee, we are required to ensure an implementation exists for any argument of the known type, to infer its return type, and to determine which implementation might be called to handle that argument. A naive approach would fail, either because of imprecise computation of intersection types or due to an infinite regress of the subtyping hierarchies. Our solution adapts prior work on checking the exhaustiveness of pattern matching, and extents it to apply over multi-methods. We define our reduction in the notation of Maranget [4], where $\mathcal{U}(P, q)$ is true if and only if there is some value v that is matched by q but not P, which can be thought of as "q is useful if added to P." Note, we assume v is well-typed with respect to the inferred argument types.

A call will succeed if the implementations handle every possible well-typed argument, or if the call is exhaustive. Exhaustivity is determined by applying the clause algorithm [4] to the list of implementations I and the argument type a. The argument type is treated as a new implementation whose usefulness we want to check, or equivalently checking that $\mathcal{U}(I, a)$ is false. Likewise, if m_i denotes the ith most specific method, then m_i would be called if and only if $\mathcal{U}((m_1, \ldots, m_{i-1}), m_i)$ is true, which implies there is some value v that m_i matches that the more specific implementations do not.

Using this approach, we are able to ensure a call site will always find an implementation, and determine which implementations (and return types) will be invoked with no loss of precision.

3 Implementation

We have implemented this inference algorithm for a subset of Julia's syntax, including function definitions and calls, variables, comprehensions, as well as standard control flow constructs, but without parameterized type constructors and modules. We are able to type check a limited amount of Julia code with this subset. Our type checker, implemented in Julia, loads the file (including dependencies) into the Julia REPL, then uses the reflective capabilities of Julia to discover available implementations, and relies on Julia's runtime type inference mechanism to find return types. This approach is guaranteed to be representative of the actual runtime environment, and ties our implementation to the Julia environment.

4 Conclusion

We have presented an algorithm in the presence of multi-methods to determine if function calls in Julia will always find an implementation, and identifies the relevant implementations. The algorithm is extensible to any multi dispatch system with constraints that can be represented in the form of pattern matching. We plan to extend our system to support more of Julia, remove the dependency on Julia's runtime by building a static representation of the program, and develop tools that would use the inferred typing information.

References

[1] Lennart Augustsson. 1985. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*. Springer.

NOOL'17, ,
2017. ACM ISBN ...$15.00
https://doi.org/

[2] Craig Chambers. 1993. *The Cecil Language.* Technical Report. Department of Computer Science and Engineering, University of Washington.

[3] Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN Notices*. ACM.

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

[5] Matthias Zenger and Martin Odersky. 2004. *Independently extensible solutions to the expression problem.* Technical Report.