Title

Background

Originally designed for scientific computing, Julia is a dynamic, high-level programming language, which can provide an order-of-magnitude better performance compared to other dynamic languages such as Python.

To achieve performance, Julia relies on the following three ingredients:

1. language restrictions,
2. an optimizing just-in-time (JIT) compiler that makes use of the restrictions,
3. the ability of language users to query the compiler to make their programs more conducive to optimizations.

As an example of (3), language users are encouraged to write so-called *type-stable* code, as discussed in our previous work (ref). Type stability enables one of the key optimizations performed by the Julia compiler --- method devirtualization. However, stability is not a property of the surface program but of the code in an intermediate representation, produced by the compiler, so Julia provides means of inspecting that representation. Another crucial optimization is type specialization, which is possible when the compiler can infer concrete types of expressions. Concrete data types in Julia cannot be extended via subtyping, which can be considered a restriction of the language (1). However, this restriction was introduced on purpose, to allow for a number of optimizations in the compiled code (2), including unboxing and direct field access.

Developing for performance

Performance is one of the key promises of Julia, and it can deliver on this promise for programs that are written "right". However, getting them right may not be trivial, especially for non-professional programmers using Julia for scientific computing (large part of Julia's users). For example, with type stability, folklore (refs to Julia discourse) suggests that Julia users have hard time understanding the property and developing code appropriately. Our research on type stability aimed to partially address that. TODO: something about the complexity of interacting with the compiler.

Another problem is a plethora of tools to analyze performance (profilers, etc.)

TODO: We propose to conduct a user study to understand how Julia programmers go about debugging for performance and what tools they might need.

Also, possibly checking code for performance properties, to avoid regressions after the compiler changes (e.g. type inference breaks where it used to work).

Multiple dispatch unleashed (TOFIX)

One of the distinguishing features of Julia is multiple dynamic dispatch (MDD). MDD allows functions to have multiple implementations specialized for different argument types, with the best implementation picked at run time for each function call. MDD enables extensibility and code reuse: for example, TODO (diagonal matrix).

Although Julia did not invent multiple dispatch, it is probably the first language to use MDD at such a large scale. TODO: there is an emerging consensus among Julia users that the lack of formal interfaces is a huge problem. Interfaces are not checkable and not discoverable. Another drawback of full-fledged dispatched is the lack of separate compilation.

We propose to analyze code for interfaces and work on a design of gradual interfaces for Julia, along with tools to discover them automatically in the existing code.

References