Title

Background

Julia (ref) is a dynamic, high-level, yet high-performance programming language, originally designed for scientific computing. Julia aims to solve the two-language problem, combining the ease of use of productivity languages such as Python and R, and performance of languages such as Fortran and C (ref).

Although Julia is dynamically typed and does not require any type annotations, types have a profound role in the language semantics and performance. Thus, Julia is designed around symmetric multiple dynamic dispatch. Multiple dispatch allows a function to have multiple implementations, called methods, tailored to different argument types; at run time, a function call is dispatched to the most specific method for the given arguments. To deliver performance, Julia relies on a type-specializing just-in-time compiler: with a few exceptions (ref), every time a method is called with a new set of argument types, it is specialized for those types.

The key to efficient compilation of Julia programs is dispatch elimination: dispatched calls are expensive, especially for functions like `(*)` that have hundreds of methods, and they prevent further optimizations, e.g. inlining. With type specialization and dispatch elimination, a function like

```
mul42(x) = x * 42
```

when called with an integer, is efficiently compiled to the following LLVM code:

```
define i64 @julia_mul42_503(i64 signext %0) #0
{ %1 = mul i64 %0, 42
  ret i64 %1 }
```

Type stability

While Julia can achieve performance comparable to that of C (ref), to be compiled effectively, programs need to be written in a particular style that is conducive to optimizations. Thus, for example, Julia programmers are encouraged to write so-called *type-stable* code. Informally, a function is type stable if the type of its output can be predicted from the input types. For instance, function `pos(x::Number) = x < 0 ? 0 : x` is not type stable: when called with a float, it returns either a float or an integer depending on the *value* of `x`. Type-unstable functions impede optimization of their callers: in particular, they prevent dispatch elimination. For example, `Float64`-specialized version of `h(x) = f(pos(x))` can optimize only the `pos` call but not `f`: because of `pos`'s instability, run-time type of `pos(x)` cannot be predicted, and thus, dispatch cannot be resolved for `f` ahead of time.

As discussed in our paper (ref), type stability is a compiler-dependent property: formally, a function is type-stable for the given input types if for those types, the compiler is able to infer a *concrete* return type, e.g. `Int64` or `Vector{Float64}`. Furthermore, it is not a property of the source language but rather of an intermediate representation where type inference is performed. To debug type stability, Julia

programmers need to inspect the result of type inference for a function call by querying the compiler. For example, `code_warntype(pos, (Float64,))` provides information on calling `pos` with a float.

References