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 convenience 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 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 often appear in hot loops, 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 optimizations of their callers: in particular, instability prevents dispatch elimination. For example, a `Float64`-specialized version of `h(x) = f(pos(x))` can optimize only the call to `pos` but not to `f`: because of `pos`'s instability, the 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 these types, the compiler is able to infer a *concrete* return type, such as `Int64` or

`Vector{Float64}`. Furthermore, stability is not a property of the source language but rather of an intermediate representation targeted by the type inference algorithm. To debug type stability, Julia programmers need to inspect type-inferred IR produced by the compiler. For this, the compiler provides easy access to compiled code: for instance, `code_warntype(pos, (Float64,))` would show the result of type inference for a call to `pos` with a float. Nevertheless, type-stable development can be a tedious process that requires understanding the IR and manually querying the compiler.

Proposed work

To make the development of type-stable and efficient Julia code more accessible, we propose to devise a source code analysis, as well as program transformations for type-unstable code. The analysis and transformations can both be incorporated into an interactive IDE tool. To guide the design of the tool, we will interview Julia programmers with different levels of expertise in Julia and familiarity with compilers.

To give an example of a transformation, consider the unstable `pos` function discussed above. There, type instability can be fixed by replacing the integer literal `0` with `zero(x)`, which returns a zero value of the same type as `x`.

The impact of instability can also be mitigated by introducing so-called function barriers (ref). A function barrier factors out the code depending on a type-unstable call into a separate function. Thus, in the following example,

```
x = pos(...)      =>      h(x) = g1(x) + g2(x)
g1(x) + g2(x)
                          x = pos(...)
                          h(x)
```

dispatch cannot be eliminated for `g1(x) + g2(x)` in the original program because of the call to unstable `pos`. After a function-barrier transformation that introduces a new function `h`, the expression `g1(x) + g2(x)` can be optimized in a version of `h` specialized for the run-time type of `x`. Thus, the new program will have one dynamically dispatched call to `h` instead of the three dynamic calls to `g1`, `g2`, and `+`. Currently, Julia does not provide an automated way of creating function barriers.

References