A Cure for Type-Unstable Code

Julia Belyakova

The key to effective compilation of Julia programs, where every function call is resolved by multiple dynamic dispatch, is dispatch elimination. Dispatched calls prevent further optimizations and are especially expensive for functions like multiplication that have hundreds of methods and often appear in hot loops. To deliver performance comparable to that of C, Julia relies on a type-specializing just-in-time compiler; a function like `mul42(x) = x * 42`, when called with an integer, is compiled to the following LLVM code without any dynamic dispatch:

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

However, to be compiled effectively, Julia programs need to be written in a particular style that is conducive to optimizations. Thus, 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, consider

```
pos(x::Number) = x < 0 ? 0 : x
```

which can be called with any subtype [3] of `Number`. This function is not type stable: when called with a float, it returns either a float or an integer depending on the *value* of the argument `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.

Notably, type stability is a compiler-dependent property [2]: 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, type 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. To reason about transformations, our model of JIT-compilation [2] needs to be extended to account for new function definitions [1]. 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. 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 `+`.

[1] J. Belyakova, B. Chung, J. Gelinas, J. Nash, R. Tate, and J. Vitek. World age in julia: Optimizing method dispatch in the presence of eval. *OOPSLA 2020*. doi:10.1145/3428275.

[2] A. Pelenitsyn, J. Belyakova, B. Chung, R. Tate, and J. Vitek. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *OOPSLA 2021*. doi:10.1145/3485527.

[3] F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, and J. Vitek. Julia Subtyping: A Rational Reconstruction. *OOPSLA 2018*. doi:10.1145/3276483.