relationalAI

If Runtime isn't Funtime:

Controlling Compile-time Execution

Nathan Daly — @nhdaly
Software Engineer at RelationalAI

July 25th, 2019

Our Experiences at **RelationalAI**

We're building a super high-performance database server and machine learning platform.

- Things need to be fast.

This talk is about our experiences trying to move specific execution from runtime to compile-time

Presentation **Overview**

What am I talking about?

- Background
- Current mechanisms

Why am I talking about it?

- Problematic examples
- Lessons learned: our experience with `@generated` functions
- Call for consideration

Background: **Julia and compile-time computation**

- Julia is a dynamic language.
- Julia is *fast* because it can optimize code to be static, when it won't affect behavior.
- This includes precomputing static values so that your code has less to do when you actually run it.

Background: **An Example of Specialization**

```
julia> f(x) = x * sizeof(x)
f (generic function with 1 method)


julia> @code_typed f(2)
CodeInfo(
1 — %1 = (Base.mul_int)(x, 8)::Int64
└──      return %1
) => Int64
```

Background: **Specialization**

This is fast because the compiler **knows the types**, so it knows what code to generate.

This is called **specialization**.

Background: **Type Instability**

If it can't determine all the types of arguments for an operation, it would be slower.

This situation is conventionally called **type instability**.

Background: **Type Instability Example**

```
julia> const values = (2, 3.0, 4);

julia> function unstable(i)
           2 * values[i]
       end;

julia> @show unstable(1); @show unstable(2);
unstable(1) = 4
unstable(2) = 6.0
```

```
julia> @code_typed unstable(1)
CodeInfo(
1 — %1  = Main.values::Const((2, 3.0, 4), false)
│   %2  = (Base.getfield)(%1, i, true)::Union{Float64, Int64}
│   %3  = (isa)(%2, Int64)::Bool
└──       goto #3 if not %3
2 — %5  = π (%2, Int64)
│   %6  = (Base.mul_int)(2, %5)::Int64
└──       goto #6
3 — %8  = (isa)(%2, Float64)::Bool
└──       goto #5 if not %8
4 — %10 = π (%2, Float64)
│   %11 = (Base.sitofp)(Float64, 2)::Float64
│   %12 = (Base.mul_float)(%11, %10)::Float64
└──       goto #6
5 —       (Core.throw)(ErrorException("fatal error in type inference (type
bound)"))::Union{}
└──       $(Expr(:unreachable))::Union{}
6 ┄ %16 = φ (#2 => %6, #4 => %12)::Union{Float64, Int64}
└──       return %16
) => Union{Float64, Int64}
```

Background: **Control over the compiler**

The previous example was not a bug! In fact it's a feature!

- That function could not be written *at all* in a static language. Julia allows you to write high-level code *that works as you expect*, and then tries to make it fast.
  - In this case, the slow code is the best we can do.

But there are some cases that in principle *could* be computed statically, but the compiler doesn't do it.

And sometimes *we the programmers* know exactly how to compute those answers, and we want to tell the compiler. That's what this talk is about.

Background: **FixedPointDecimals.jl**

Many of the following examples come from optimizing FixedPointDecimals.jl.

Problem: Floats can represent a *very large* range of numbers, but are imprecise

- ```
  julia> 5e-324, 3e307
  (5.0e-324, 3.0e307)
  ```

- ```
  julia> 0.1 + 0.2
  0.30000000000000004
  ```

For some tasks (e.g. operations w/ money), we must be precise, but only need a small range of numbers

Background: **FixedPointDecimals.jl**

Solution: `FixedDecimal{T,f}` uses an integer and a precision to represent decimals:

```
julia> dump(  FixedDecimal{Int, 3}(2.001)  )
FixedDecimal{Int64,3}
  i: Int64 2001
```

- `FixedDecimal{Int,3}`:
  - `"1"     ==> 1000`
  - `"0.3"   ==>  300`
  - `"0.05"  ==>   50`
  - `"2.001" ==> 2001`

- `FixedDecimal{Int32,5}`:
  - `"1"     ==> Int32(100000)`
  - `"0.3"   ==>  Int32(30000)`
  - `"0.05"  ==>   Int32(5000)`
  - `"2.001" ==> Int32(200100)`

Existing mechanisms in Julia

Julia already provides several mechanisms for this kind of control, via staged programming and metaprogramming:

- specialization
- macros
- @generated
- @pure
- @eval

Next, we will look at some weird cases where the right solution is not obvious.

Problematic cases for controlling compiler execution

- Perform error-checking on type parameters
- Precompute/cache expensive computations for different types (and make available to compiler)
- Generate different code, or different algorithms, for different types

Problematic cases for controlling compiler execution

- **Perform error-checking on type parameters**
- Precompute/cache expensive computations for different types (and make available to compiler)
- Generate different code, or different algorithms, for different types

Problematic example

- Perform error-checking on type parameters

```
julia> using FixedPointDecimals  # (v0.3.0)

julia> FixedDecimal{Int8, 5}(1)  # ArgumentError: 5 decimals is too many!
ERROR: ArgumentError: Requested number of decimal places 5 exceeds the max
allowed for the storage type Int8: [0, 2]

julia> FixedDecimal{Int8, 2}(1)  # 2 decimals (100) fits in Int8 ✔
FixedDecimal{Int8,2}(1.00)
```

- Understandable error messages are great!
- Paying runtime cost for them is not great.

Problematic example

- Perform error-checking on type parameters

```
julia> # Oh boy....
       @code_typed reinterpret(FixedDecimal{Int8, 2}, 100)
CodeInfo(
1 ──       nothing::Nothing
2 ┄─ %2  = φ (#1 => 1, #3 => %6)::Int16
│    %3  = φ (#1 => 0, #3 => %7)::Int64
│    %4  = Base.slt_int(%2, 127)::Bool
└────       goto #4 if not %4
3 ── %6  = Base.mul_int(%2, 10)::Int16
│    %7  = Base.add_int(%3, 1)::Int64
└────       goto #2
4 ── %9  = Base.sub_int(%3, 1)::Int64
└────       goto #5
5 ── %11 = Base.slt_int(%9, 0)::Bool
└────       goto #7 if not %11
6 ──       goto #8
7 ── %14 = $(Expr(:static_parameter, 2))::Core.Compiler.Const(2, false)
└──── %15 = Base.sle_int(%14, %9)::Bool
8 ┄─ %16 = φ (#6 => %11, #7 => %15)::Bool
└────       goto #10 if not %16
9 ── %18 = Base.trunc_int(Int8, i)::Int8
│    %19 = %new(FixedDecimal{Int8,2}, %18)::FixedDecimal{
└────       return %19
10 ─       invoke
FixedPointDecimals._throw_storage_error($(Expr(:static_pa
2))::Int64, $(Expr(:static_parameter, 1))::Type, %9::Int6
└────       $(Expr(:unreachable))::Union{}
) => FixedDecimal{Int8,2}
```

Problematic example

- Perform error-checking on type parameters

The authors of this function expected that check to compile away.

```
84  struct FixedDecimal{T <: Integer, f} <: Real
85      i::T
86
87      # inner constructor
88      function Base.reinterpret(::Type{FixedDecimal{T, f}}, i::Integer) where {T, f}
89          n = max_exp10(T)
90          if f >= 0 && (n < 0 || f <= n)
91              new{T, f}(i % T)
92          else
93              # Note: introducing a function barrier to improve performance
94              # https://github.com/JuliaMath/FixedPointDecimals.jl/pull/30
95              _throw_storage_error(f, T, n)
96          end
97      end
98  end
```

https://github.com/JuliaMath/FixedPointDecimals.jl/blob/v0.3.0/src/FixedPointDecimals.jl#L88

Problematic example

- Perform error-checking on type parameters

We can get the performance we expected by removing the error checking:

```
    # inner constructor
    function Base.reinterpret(::Type{FixedDecimal{T, f}}, i::Integer) where {T, f}
        new{T, f}(i % T)
    end
```

```
julia> @code_typed reinterpret(FixedDecimal{Int8, 2}, 100)
CodeInfo(
1 — %1 = (Base.trunc_int)(Int8, i)::Int8
│   %2 = %new(FixedDecimal{Int8,2}, %1)::FixedDecimal{Int8,2}
└──      return %2
) => FixedDecimal{Int8,2}
```

Problematic example

- Perform error-checking on type parameters

Ideally, we want some way to ***stage the error checking***

We want to tell the compiler to perform the entire error checking operation *ahead of time*, when compiling the function, not at runtime.

This is similar to the desire to manually control inlining via `@inline` and `@noinline`: **it's usually best to let the compiler decide, but sometimes we know better.**

Solutions Considered

- **Perform error-checking on type parameters**

- specialization
- @pure
- @generated

Problematic cases for controlling compiler execution

- Perform error-checking on type parameters
- **Precompute/cache expensive computations for different types (and make available to compiler)**
- Generate different code, or different algorithms, for different types

Problematic example

• Precompute/cache expensive computations for different types

Similar to the previous example, there are some values needed for computations, that **always have the same values for the same types**.

- Always the same for the same types
- Fully computable based only on static information (types and constants)
- Need to be available to the compiler for inlining, and/or other static computations.

Problematic example

• Precompute/cache expensive computations for different types

Examples:

- Compute a tuple of zeros for each type in Tuple type:
  - ```
    julia> zero_tuple(Tuple{Int, Float32, Float64})
    (0, 0.0, 0.0)
    ```
- FixedPointDecimals.jl needs to precompute some constants ("magic numbers") per `{T,f}` pair for performance:
  - ```
    # Largest n where 10^n fits in T
    max_exp10(::Type{T <: Integer})
    ```
  - ```
    # T(10)^f
    coefficient(::Type{FixedDecimal{T, f}})
    ```
  - ```
    # Compute (2^64 ÷ C) as a "magic number"
    inverse_coeff(::Type{T <: Integer}, C)
    ```

Problematic example

• Precompute/cache expensive computations for different types

- Ex: We want this function to completely const-fold:
  - `zero_tuple(Tuple{FixedDecimal{Int,2}}) == (FD{Int,2}(0.0),)`
- The conundrum:
  - With some clever tricks, we can write a definition that const-folds correctly in *some cases*, but with complex types it doesn't.
  - But we as the programmers know that this can be pre-computed and won't change at runtime (within a given world age of course)
- There is no way for us to tell the compiler to *try harder* to optimize a block away
  - Ideally, we'd have some way to explicitly request staging.

Solutions Considered

- **Precompute/cache expensive computations for different types**

  - specialization
  - @pure
  - @generated
  - @eval

Aside:

- Precompute/cache expensive computations for different **constants**

```
coefficient(::Type{FixedDecimal{T,f}}) where {T,f} = T(10)^f
```

```
julia> @code_typed coefficient(FixedDecimal{Int,2})
CodeInfo(
1 —     return 100
) => Int64
```

Aside:

- Precompute/cache expensive computations for different **constants**

```
julia> @code_typed coefficient(FixedDecimal{Int,5})
 CodeInfo(
 1 —       nothing::Nothing
 2 ┄ %2  = φ (#1 => 10, #6 => %16)::Int64
 │   %3  = φ (#1 => 10, #6 => %23)::Int64
 │   %4  = φ (#1 => 2, #6 => %15)::Int64
 │   %5  = (Base.slt_int)(0, %4)::Bool
 └──       goto #7 if not %5
 3 — %7  = (Base.cttz_int)(%4)::Int64
 │   %8  = (Base.add_int)(%7, 1)::Int64
 │   %9  = (Base.sle_int)(0, %8)::Bool
 │   %10 = (Base.bitcast)(UInt64, %8)::UInt64
 │   %11 = (Base.ashr_int)(%4, %10)::Int64
 │   %12 = (Base.neg_int)(%8)::Int64
 │   %13 = (Base.bitcast)(UInt64, %12)::UInt64
```

Aside:
• Precompute/cache expensive computations for different **constants**

- `coefficient(::Type{FixedDecimal{T,f}}) where {T,f} = T(10)^f`

- Currently Julia does not have any mechanism to specialize on Constants.
- Some multistaged languages (such as *Stratego*) let you mark a variable as **a value that should be specialized-on *if possible***. For example:
  - `power(x, <y>) = <y == 0> ? 1 : x * power(x, <y-1>)`

- In this example, *iff y is known at compile time*, it is specialized on, otherwise it produces normal, dynamic code.

Problematic cases for controlling compiler execution

- Perform error-checking on type parameters
- Precompute/cache expensive computations for different types (and make available to compiler)
- **Generate different code, or different algorithms, for different types**

Problematic example

- Generate different code, or different algorithms, for different types

In most examples, we can achieve this via control-flow and/or method dispatch.

```
#  this check will compile away
if x isa Union{Float64, Float32} ... else ... end
```

But sometimes we need to generate more complex code structure based on input types.

Problematic example

- Generate different code, or different algorithms, for different types

```
julia> @generated function add_tuples(x::T, y::T) where {T<:Tuple}
           exprs = []
           for n in 1:fieldcount(T)
               push!(exprs, :(x[$n] + y[$n]))
           end
           :( ($(exprs...),) )
       end
add_tuples (generic function with 1 method)

julia> add_tuples((1,2,3), (4,5,6))
(5, 7, 9)
```

Here, add_tuples, generates this:

```
:( (x[1] + y[1], x[2] + y[2], x[3] + y[3],) )
```

Problematic example

- Generate different code, or different algorithms, for different types

What we get with `@generated` functions is super cool!

In many languages, there is just no way to do what we did here.

A cautionary tale: Our experience w/ `@generated` functions

- The `@generated` function mechanism is a super convenient mechanism for explicit staged programming.
- At our company, we saw `@generated` functions as *the mechanism* for explicit staging, and tried to use it for **all cases where we wanted to control compiletime execution**.
- This turned out to be a problem for a few reasons:
  - Because of frozen world age, you cannot use `@generated` functions in generic interfaces, or for interactive debugging.
  - Testing generated functions are hard.
  - And we've been told `@generated` functions can sometimes be *less performant*, but we haven't verified this one.

Lessons learned about `@generated` functions

Generated functions behave differently within a package than between packages:

- The world age is frozen at the end of "precompiling" a package, so a lot of the rules that make @generated functions difficult to use correctly are only exposed between packages.
- This means that now we are unable to break our large code-base up into smaller packages, because our @generated functions stop working and start causing world age errors.

Lessons learned about `@generated` functions

Generated function bodies should not call "generic" functions because those functions' definitions may change, and the generated function will be *out of sync* with the rest of the code (due to the frozen world age)

A consequence of this: Generated functions **should *not* precompute *values***, they are only for generating *code structure*.

Call for consideration

Hopefully this talk has given you a more detailed understanding of some use-cases for staged programming in julia.

We think there is room for more creativity around this topic, and we'd love to see explorations of more solutions in this area.

We've opened a channel on Slack for discussion about this topic, here:

- #compiletime-magic

(And feel free to reach out to me individually!)

Summary

- Background: compile-time computation
  - Julia is a dynamic language, but through specialization, code generation, and optimization, it can produce fast static code.
- Problematic examples
  - There are some use-cases where the compiler isn't able to produce the optimized code we want, but we as the programmers know what it should do.
  - We want to be able to *tell the compiler* how to stage some computations.
  - There are several existing mechanisms for addressing different aspects of this topic, but the motivating examples I provided are all sufficiently weird that none of the solutions directly applies.
- Call for consideration

Thank you!

relationalAI

relationalAI

Appendix A

Potential Proposals

Proposal:

- Perform error-checking on type parameters

Desire: We could imagine some kind of control over this staging, maybe via a macro or some other mechanism. *I'm not proposing anything concrete, but **something** like this would help.*

```
# inner constructor
function Base.reinterpret(::Type{FixedDecimal{T, f}}, i::Integer) where {T, f}
    @constfold begin  # This block would never be part of the runtime code
        n = max_exp10(T)
        @assert f >= 0 && (n < 0 || f <= n) "..."
    end

    new{T, f}(i % T)
end
```

Proposal: **A safer specializer**

"Normal" specialization is allowed to access method-tables when attempting optimization (this is achieved via back-edges).

It would be nice to have a compiler hint on a block of code (either at function definition or at the call-site) that asks the compiler to "turn up specialization" to 11, similar to what `@pure` does, but is more explicit when it fails: either throwing an error or printing a warning.

This would allow an author to be confident that their code will compile-away, even if definitions change in the future, and if not, they'll *know about the regression*.

Proposal: **A safer specializer**

This could be something like:

- `@constfolded coefficient(::Type{T}, f) = T(10)^f`

Or maybe this could follow the approach taken by `` `if @generated` ``, to allow a function to provide different implementations based on whether or not some computation was optimized away

```
function convert(::Type{FixedDecimal{T, f}}, v) where {T,f}
    if @constfolded C = coefficient(T, f) "coefficient($T, $f) failed to const-fold"
        FixedDecimal{T,f}(v / C)
    else
        FixedDecimal{T,f}(v / coefficient(T, f))
    end
end
```

Proposal: **Specialize on Constants**

We could imagine a syntax that allowed julia generated functions to *dispatch on whether a value is a Compiler.Const*, which would allow us to specialize on the value, *only when it's advantagous*:

```
# Default, runtime calculation
power(x, y) = y == zero(y) ? one(x) : x * power(x, y-1)

# Specialized calculation only for when it's advantageous
power(x, y::Compiler.Const) = power(x, Val(y))
@generated power(x, ::Val{y}) where {y} =
     :(if $y == 0 1 else x * power(x, $(Val{y-1}())) end)
```

relationalAI

The End