Making dynamic programs run fast

Valentin Churavy (@vchuravy)

Yet another high-level language?

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end

julia> mandel(complex(.3, -.6))
14
```

Yet another high-level language?

Typical features

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

Unusual features

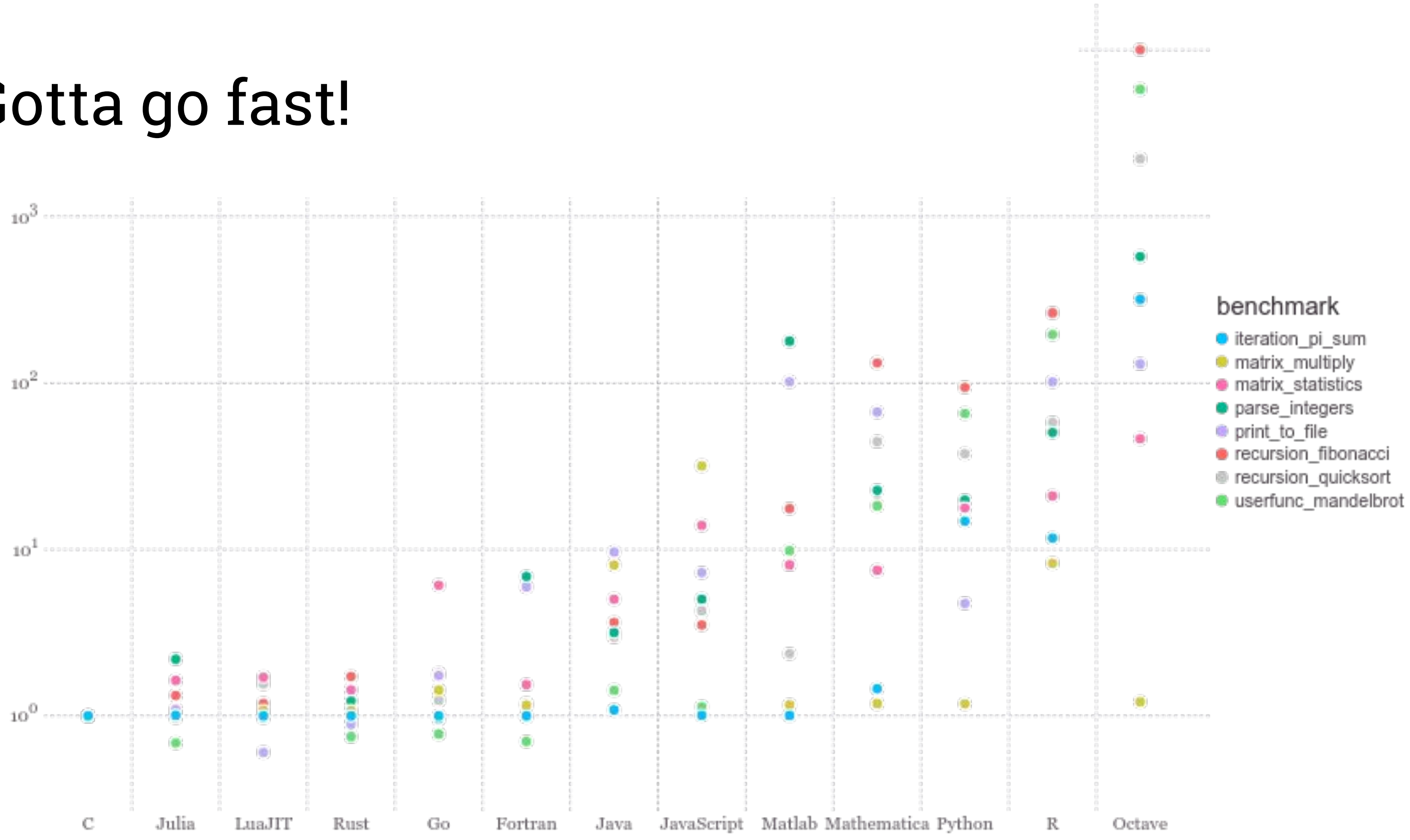
Great performance!

JIT AOT-style compilation

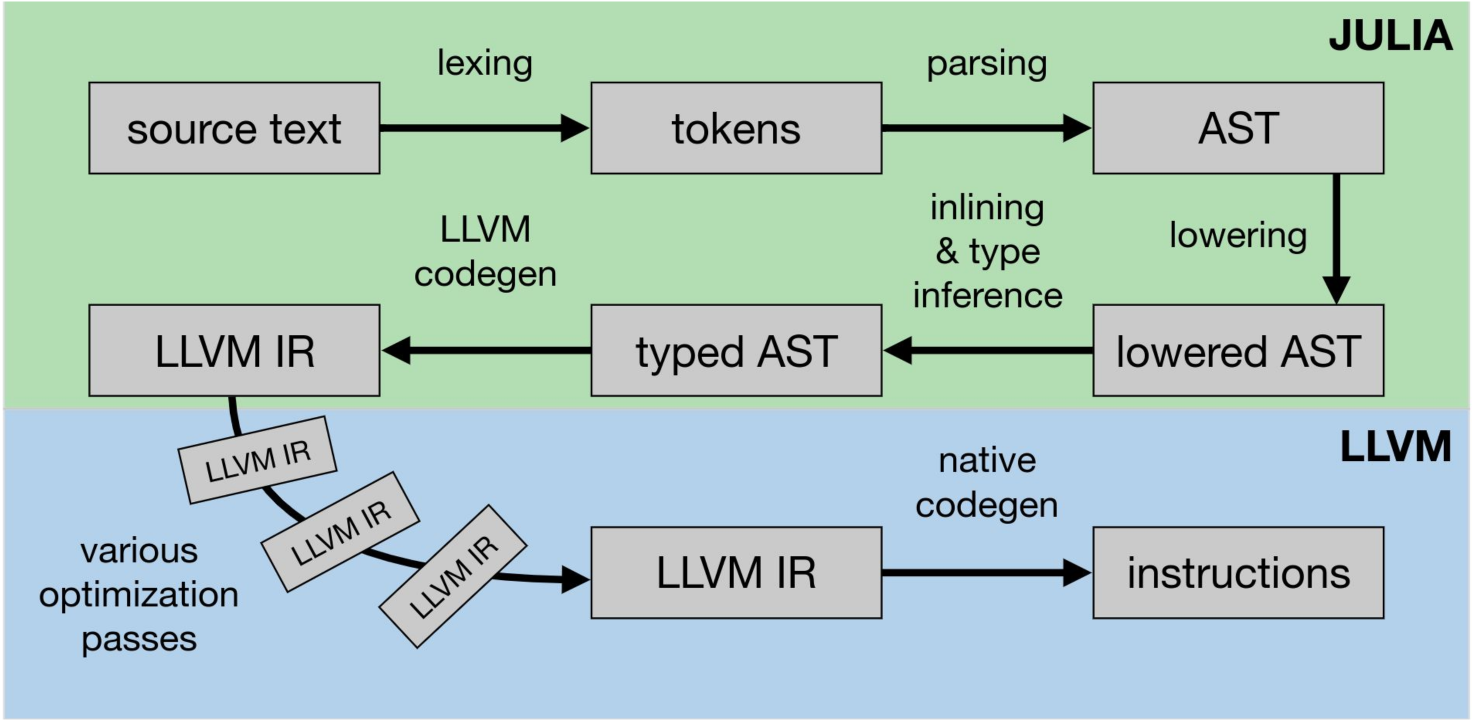
Most of Julia is written in Julia

Reflection and metaprogramming

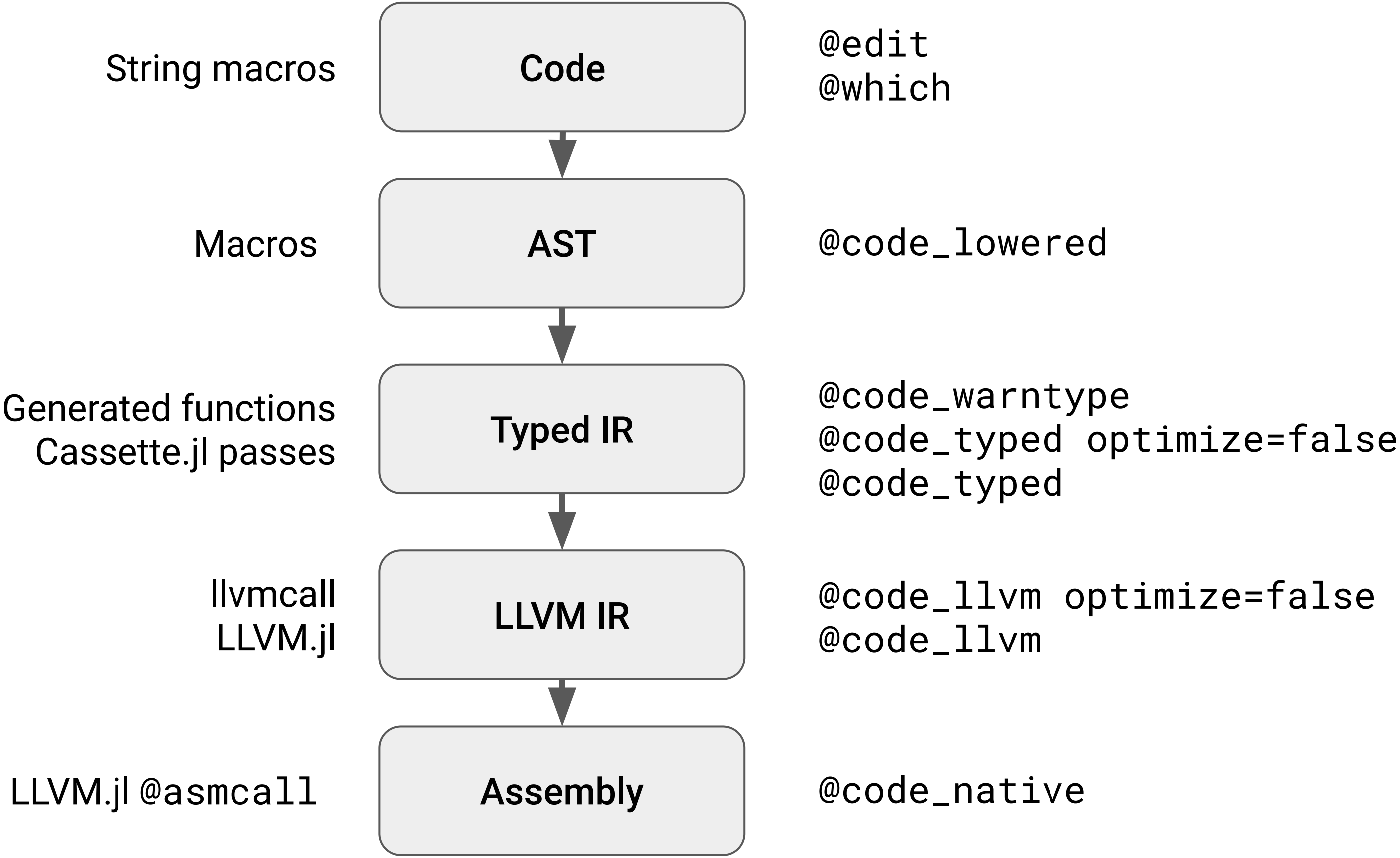
Gotta go fast!



Compiling Julia



Introspection and staged metaprogramming



Avoid runtime uncertainty

1. Sophisticated type system
2. Type inference
3. Multiple dispatch
4. Specialization
5. JIT compilation

Julia: Dynamism and Performance Reconciled by Design (doi:10.1145/3276490)

Dynamic semantics + Static analysis

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end
julia> mandel(UInt32(1))
2
```

```
julia> methods(abs)
# 13 methods for generic function "abs":
[1] abs(x::Float64) in Base at float.jl:522
[2] abs(x::Float32) in Base at float.jl:521
[3] abs(x::Float16) in Base at float.jl:520
…
[13] abs(z::Complex) in Base at complex.jl:260
```

Everything is a virtual function call?

What happens on a call

```
sin(x)


typeof(x) == Float64
```

```
julia> methods(sin)
# 12 methods for generic function "sin":
[1] sin(x::BigFloat) in Base.MPFR at mpfr.jl:743
[2] sin(::Missing) in Base.Math at math.jl:1072
[3] sin(a::Complex{Float16}) in Base.Math at math.jl:1020
[4] sin(a::Float16) in Base.Math at math.jl:1019
[5] sin(z::Complex{T}) where T in Base at complex.jl:796
[6] sin(x::T) where T<:Union{Float32, Float64} in Base.Math at
special/trig.jl:30
[7] sin(x::Real) in Base.Math at special/trig.jl:53
```

The right method is chosen using dispatch and then a method specialization is compiled for the signature

Method specialization

```
julia> ml = methods(sin);
julia> m = ml.ms[6]
sin(x::T) where T<:Union{Float32, Float64} in Base.Math at special/trig.jl:30
julia> m.specializations

julia> sin(1.0);
julia> m.specializations
TypeMapEntry(..., Tuple{typeof(sin),Float64}, ..., MethodInstance for sin(::Float64), ...)
```

Multiple dispatch

Rule: Call most specific method

```julia
f(x, y::Int) = 0
f(y::Int, 1) = 1
f(x, y::Float64) = 2

@show f(1.0, 1)
@show f(1, "hello")
@show f("hello", 1.0)
@show f(1, 1.0)
```

Dynamic semantics + Static analysis

```julia
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end
julia> mandel(UInt32(1))
2
```

```julia
julia> @code_lowered mandel(UInt32(1))
1 ─       z@_7 = z@_2
│         c = z@_7
│         maxiter = 80
│   %4  = 1:maxiter
│         @_5 = Base.iterate(%4)
│   %6  = @_5 === nothing
│   %7  = Base.not_int(%6)
└──       goto #6 if not %7
2 ┄ %9  = @_5
│         n = Core.getfield(%9, 1)
│   %11 = Core.getfield(%9, 2)
│   %12 = Main.abs(z@_7)
│   %13 = %12 > 2
└──       goto #4 if not %13
3 ─ %15 = n - 1
└──       return %15
4 ─ %17 = z@_7
│   %18 = Core.apply_type(Base.Val, 2)
│   %19 = (%18)()
│   %20 = Base.literal_pow(Main.:^, %17, %19)
│         z@_7 = %20 + c
│         @_5 = Base.iterate(%4, %11)
│   %23 = @_5 === nothing
│   %24 = Base.not_int(%23)
└──       goto #6 if not %24
5 ─       goto #2
6 ┄       return maxiter
```

Dynamic semantics + Static analysis

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end
julia> mandel(UInt32(1))
2
```

```
julia> @code_lowered mandel(UInt32(1))
1 ─       z@_7 = z@_2
│         c = z@_7
│         maxiter = 80
│   %4  = 1:maxiter
│         @_5 = Base.iterate(%4)
│   %6  = @_5 === nothing
│   %7  = Base.not_int(%6)
└──       goto #6 if not %7
2 ┄ %9  = @_5
│         n = Core.getfield(%9, 1)
│   %11 = Core.getfield(%9, 2)
│   %12 = Main.abs(z@_7)
│   %13 = %12 > 2
└──       goto #4 if not %13
3 ─ %15 = n - 1
└──       return %15
4 ─ %17 = z@_7
│   %18 = Core.apply_type(Base.Val, 2)
│   %19 = (%18)()
│   %20 = Base.literal_pow(Main.:^, %17, %19)
│         z@_7 = %20 + c
│         @_5 = Base.iterate(%4, %11)
│   %23 = @_5 === nothing
│   %24 = Base.not_int(%23)
└──       goto #6 if not %24
5 ─       goto #2
6 ┄       return maxiter
```

Dynamic semantics + Static analysis

```julia
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end
julia> mandel(UInt32(1))
2
```

```julia
julia> @code_typed optimize=false mandel(UInt32(1))
1 ─       (z@_7 = z@_2)::UInt32
│         (c = z@_7)::UInt32
│         (maxiter = 80)::Compiler.Const(80, false)
│   %4  = (1:maxiter)::Compiler.Const(1:80, false)
│         (@_5 = Base.iterate(%4))::Compiler.Const((1, 1), false)
│   %6  = (@_5 === nothing)::Compiler.Const(false, false)
│   %7  = Base.not_int(%6)::Compiler.Const(true, false)
└──       goto #6 if not %7
2 ┄ %9  = @_5::Tuple{Int64,Int64}::Tuple{Int64,Int64}
│         (n = Core.getfield(%9, 1))::Int64
│   %11 = Core.getfield(%9, 2)::Int64
│   %12 = Main.abs(z@_7)::UInt32
│   %13 = (%12 > 2)::Bool
└──       goto #4 if not %13
3 ─ %15 = (n - 1)::Int64
└──       return %15
4 ─ %17 = z@_7::UInt32
│   %18 = Core.apply_type(Base.Val, 2)::Compiler.Const(Val{2}, false)
│   %19 = (%18)()::Compiler.Const(Val{2}(), false)
│   %20 = Base.literal_pow(Main.:^, %17, %19)::UInt32
│         (z@_7 = %20 + c)::UInt32
│         (@_5 = Base.iterate(%4, %11))::Union{Nothing, Tuple{Int64,Int64}}
│   %23 = (@_5 === nothing)::Bool
│   %24 = Base.not_int(%23)::Bool
└──       goto #6 if not %24
5 ─       goto #2
6 ┄       return maxiter::Core.Compiler.Const(80, false)
```

Dynamic semantics + Static analysis

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end
julia> mandel(UInt32(1))
2
```

```
julia> @code_typed optimize=true mandel(UInt32(1))
1 ─       goto #9 if not true
2 ┄ %2  = φ (#1 => 1, #8 => %18)::Int64
│   %3  = φ (#1 => 1, #8 => %19)::Int64
│   %4  = φ (#1 => _2, #8 => %12)::UInt32
│   %5  = Core.zext_int(Core.UInt64, %4)::UInt64
│   %6  = Base.ult_int(0x0000000000000002, %5)::Bool
│   %7  = Base.or_int(false, %6)::Bool
└──       goto #4 if not %7
3 ─ %9  = Base.sub_int(%2, 1)::Int64
└──       return %9
4 ─ %11 = Base.mul_int(%4, %4)::UInt32
│   %12 = Base.add_int(%11, z@_2)::UInt32
│   %13 = (%3 === 80)::Bool
└──       goto #6 if not %13
5 ─       goto #7
6 ─ %16 = Base.add_int(%3, 1)::Int64
└──       goto #7
7 ┄ %18 = φ (#6 => %16)::Int64
│   %19 = φ (#6 => %16)::Int64
│   %20 = φ (#5 => true, #6 => false)::Bool
│   %21 = Base.not_int(%20)::Bool
└──       goto #9 if not %21
8 ─       goto #2
9 ┄ %24 = π (80, Core.Compiler.Const(80, false))
└──       return %24
```

Using the power of LLVM

```julia
function popcount(x)
    count = 0
    while x != 0
        x &= x - 1
        count += 1
    end
    count
end
```

```
@code_native popcount(UInt64(3))
    popcntq     %rdi, %rax
    cmoveq      %rdi, %rax
    retq
    nopw        (%rax,%rax)
```

Using the power of LLVM

```julia
function popcount(x)
    count = 0
    while x != 0
        x &= x - 1
        count += 1
    end
    count
end
```

```
@code_native popcount(UInt128(3))
    popcntq    %rsi, %rcx
    popcntq    %rdi, %rax
    addq    %rcx, %rax
    cmoveq    %rax, %rax
    retq
    nopw    %cs:(%rax,%rax)
```

Lessons learned — 1

“Julia is a dynamic language and follows dynamic semantics — Never forget”

Type-inference as an optimization to find static (or near static) subprograms

- Aggressive de-virtualization
- Inlining
- Constant propagation

Raises problem of cache invalidation.

Lessons learned — 2

"Julia is a dynamic language and follows dynamic semantics — Never forget"

```
julia> f(t) = ntuple(Val(length(t))) do i
                  Base.@_inline_meta
                  sin(t[i])
              end
f (generic function with 1 method)

julia> @code_typed f((1.0, 2.0f0, 3+1im))
CodeInfo(
1 ─ %1 = Base.getfield(t, 1, true)::Float64
│   %2 = invoke Main.sin(%1::Float64)::Float64
│   %3 = Base.getfield(t, 2, true)::Float32
│   %4 = invoke Main.sin(%3::Float32)::Float32
│   %5 = Base.getfield(t, 3, true)::Complex{Int64}
│   %6 = invoke Main.sin(%5::Complex{Int64})::Complex{Float64}
│   %7 = Core.tuple(%2, %4, %6)::Tuple{Float64,Float32,Complex{Float64}}
└──      return %7
) => Tuple{Float64,Float32,Complex{Float64}}
```

```
julia> f() = sin(2.0)
f (generic function with 1 method)

julia> @code_typed f()
CodeInfo(
1 ─     return 0.9092974268256817
) => Float64
```

Lessons learned — 3

"Julia is a dynamic language and follows dynamic semantics — Never forget"

Julia 0.3

```
julia> f() =  1
julia> g() = f()
julia> g()
1

julia> f() = 2
julia> g()
1
```

Julia 1.0

```
julia> f() =  1
julia> g() = f()
julia> g()
1

julia> f() = 2
julia> g()
2
```

https://github.com/JuliaLang/julia/issues/265

Lessons learned

"Julia is a dynamic language and follows dynamic semantics — Never forget"

Concrete types are not extendable Int64 <: Number <: Any

- Dynamic semantics implies no closed-world semantics
- Enables more aggressive de-virtualization
- Data can be stored inline/consecutively in memory

Much harder to pull off in a object-oriented language.

Julia uses multiple-dispatch and for de-virtualization we need "final" call signatures.

Lessons learned

"Julia is a dynamic language and follows dynamic semantics — Never forget"

Value types and reference types

- Model semantic difference between immutable and mutable objects
- Allows users to avoid the GC and therefore avoid latency
    - Escape analysis for mutable objects
    - Immutable objects can be often stack allocated
        - References to mutable objects prevent that

Retargeting the language

With material from Tim Besard (@maleadt)

1. Powerful dispatch
2. Small runtime library
3. Staged metaprogramming
4. Built on LLVM

maleadt / **LLVM.jl**

Unwatch 4 | Unstar 40 | Fork 18

Code | Issues 5 | Pull requests 0 | Insights | Settings

Julia wrapper for the LLVM C API

Edit

julia | julia-library | llvm-bindings | llvm | Manage topics

578 commits | **10** branches | **38** releases | **1** environment | **11** contributors | View license

High Level LLVM Wrapper

```julia
using LLVM

mod = LLVM.Module("my_module")

param_types = [LLVM.Int32Type(), LLVM.Int32Type()]
ret_type = LLVM.Int32Type()
fun_type = LLVM.FunctionType(ret_type, param_types)
sum = LLVM.Function(mod, "sum", fun_type)

Builder() do builder
   entry = BasicBlock(sum, "entry")
   position!(builder, entry)

   tmp = add!(builder, parameters(sum)[1],
              parameters(sum)[2], "tmp")
   ret!(builder, tmp)

   println(mod)
   verify(mod)
end
```

```julia
julia> mod = LLVM.Module("test")
  ; ModuleID = 'test'
  source_filename = "test"

julia> test = LLVM.Function(mod, "test",
                LLVM.FunctionType(LLVM.VoidType()))
  declare void @test()

julia> bb = BasicBlock(test, "entry")
  entry:

julia> builder = Builder();
       position!(builder, bb)

julia> ret!(builder)
  ret void
```

High Level LLVM Wrapper

```julia
function runOnModule(mod::LLVM.Module)
    # ...
    return changed
end

pass = ModulePass("SomeModulePass", runOnModule)
ModulePassManager() do pm
    add!(pm, pass)
    run!(pm, mod)
end
```

Descent into madness

```julia
function add(x::T, y::T) where {T <: Integer}
   return x + y
end

@test add(1, 2) == 3
```

Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
    return quote
        x + y
    end
end

@test add(1, 2) == 3
```

Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
   T_int = "i$(8*sizeof(T))"

   return quote
       Base.llvmcall($"""%rv = add $T_int %0, %1
                         ret $T_int %rv""", T,
                     Tuple{T, T}, x, y)
   end
end

@test add(1, 2) == 3
```

Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
   T_int = convert(LLVMType, T)

   param_types = LLVMType[T_int, T_int]
   llvm_f, _ = create_function(T_int, [T_int, T_int])
   mod = LLVM.parent(llvm_f)

   Builder() do builder
       entry = BasicBlock(llvm_f, "top")
       position!(builder, entry)
       rv = add!(builder, parameters(llvm_f)...)
       ret!(builder, rv)
   end

   call_function(llvm_f, T, Tuple{T, T}, :((x, y)))
end

@test add(1, 2) == 3
```

```
julia> @code_llvm add(UInt128(1),
                      UInt128(2))

define void @julia_add(i128* sret,
                       i128, i128) {
top:
 %3 = add i128 %2, %1
 store i128 %3, i128* %0, align 8
 ret void
}
```

JuliaCon: Yearly user and developer meetup

2019: Baltimore, MD ~360 atteendees

2020: 27th - 31st of July, 2020, Lisbon, Come join us

Valentin Churavy (@vchuravy)

Thanks to: Tim Besard, Peter Ahrens, Jarret Revels, Jameson Nash, Nathan Daly, Jane Herriman, Jeff Bezanson, Keno Fischer, Lucas Wilcox, Simon Bryne, Kiran Pamnany, Andreas Noack, Alan Edelman and many others

vchuravy@csail.mit.edu