Dynamic Compiler Pass Injection for Julia

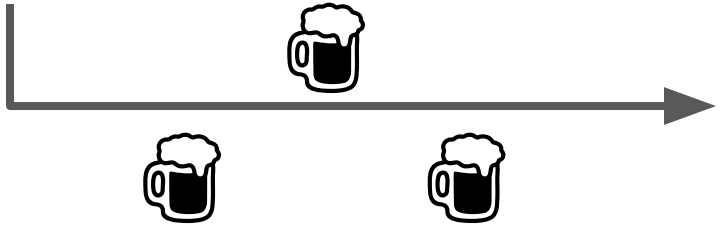
Jarrett Revels, MIT

First Things First: Hello!

- I’m Jarrett, I work at MIT on Julia
- I’ve authored a bunch of AD packages, some performance tooling packages, and a smattering of other things
- Cassette was originally motivated by AD. I needed to extend Julia at the language-level - existing interfaces were insufficient

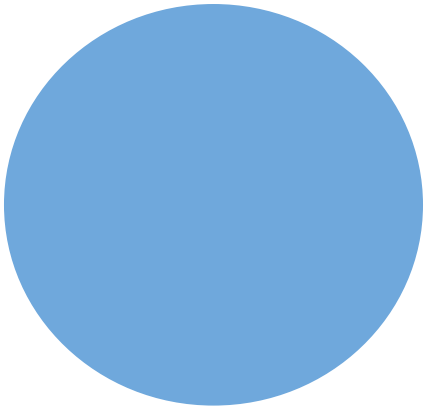
First Things First: Hello!

- I'm Jarrett, I work at MIT on Julia
- I've authored a bunch of AD packages, some performance tooling packages, and a smattering of other things
- Cassette was originally motivated by AD. I needed to extend Julia at the language-level - existing interfaces were insufficient
- I owe Jameson Nash several beers



many julia packages try to fit a

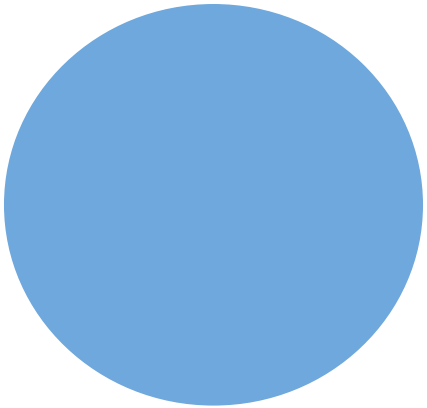
peg into a



hole

many julia packages try to fit a

peg into a



hole

many julia packages try to fit a

peg into a

hole

method overloading

```
immutable Dual{T}
   value::T
   deriv::T
end

sin(d::Dual) =
    Dual(sin(d.value), cos(d.value) * d.deriv)

cos(d::Dual) =
    Dual(cos(d.value), -sin(d.value) * d.deriv)

(+)(a::Dual, b::Dual) =
    Dual(a.value + b.value, a.deriv + b.deriv)

(*)(a::Dual, b::Dual) =
    Dual(a.value * b.value,
         b.value * a.deriv + a.value * b.deriv)
```

nonstandard interpretation

reinterpret a Julia program into the Julia program that calculates the original's derivative

method overloading

```
immutable Interval{T}
    lo::T
    hi::T
end

(+)(x::Interval, y::Interval) =
  Interval(x.lo + y.lo, x.hi + y.hi)

(-)(x::Interval, y::Interval) =
  Interval(x.lo - y.hi, x.hi - y.lo)

(*)(x::Interval, y::Interval) = begin
  a,b,c,d = x.lo*y.lo, x.hi*y.lo, x.lo*y.hi, x.hi*y.hi
  Interval(min(a,b,c,d), max(a,b,c,d))
end
```

nonstandard interpretation

reinterpret a Julia program into the Julia program that calculates rigorous bounds on the original's output

method overloading

```
immutable Track{T}
    value::T
    tape::Vector{Any}
end

function sin(x::Track)
    push!(x.tape, (sin, x))
    Track(sin(x.value), x.tape)
end

function (*)(x::Track, y::Track)
    tape = mergetapes(x.tape, y.tape)
    push!(tape, (sin, x))
    Track(x.value * y.value, tape)
end
```

nonstandard interpretation

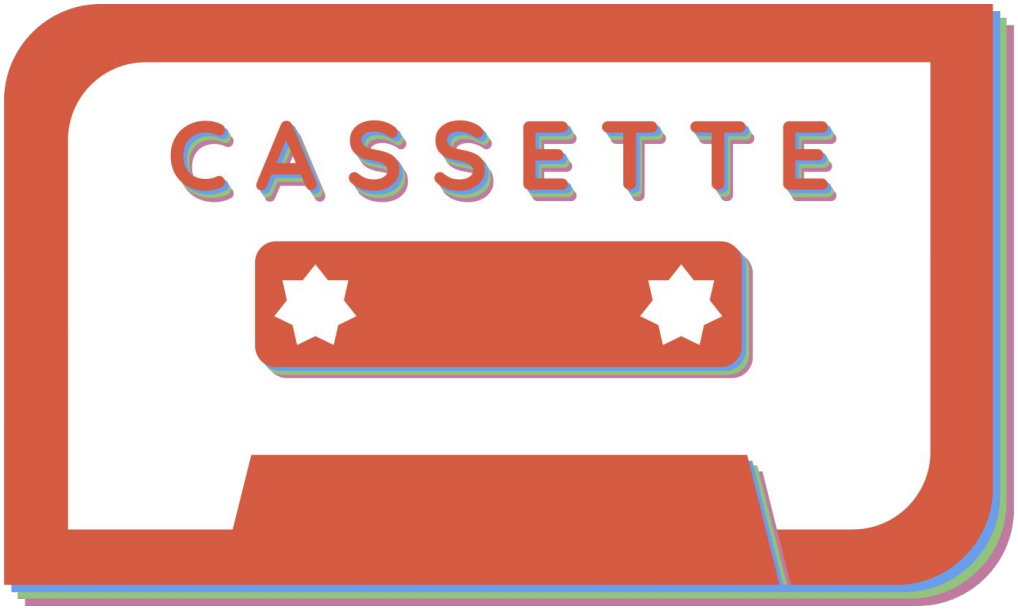
reinterpret a Julia program into the Julia program that produces a linear “instruction tape” for the original

This Approach Stinks

- commonly overloaded methods with multiple arguments will quickly run into ambiguity errors when composing with other packages (potentially load order dependent behavior)

- structural and/or dispatch type constraints in target programs can easily thwart these implementations

- not all relevant language-level mechanisms are exposed via overloadable method calls (e.g. control flow, literals, bindings)



mmm...smells nice

like a warm tape machine or baked bread or something

But what *is* Cassette?

- Cassette allows you to inject your own **code transformation passes** into Julia's JIT compilation cycle, enabling normal Julia packages to **analyze, optimize, and modify Cassette-unaware Julia programs.**

- On top of this pass injection mechanism, Cassette exposes **contextual dispatch**. With Cassette, you can overload arbitrary Julia methods - even builtins like `throw` - by dispatching on hidden "context" type parameters.

- Cassette solves the aforementioned problems by allowing Julia package developers to **arbitrarily redefine the execution of Julia programs within a given "context"**, effectively exposing a nice interface to nonstandard interpretation.

...wat

Example: Simple Logging

```
julia> using Cassette: @context, @prehook, @overdub

julia> @context PrintCtx

julia> @prehook (f::Any)(args...) where {__CONTEXT__<:PrintCtx} = println(f, args)

julia> @overdub(PrintCtx(), sin(1))
sin(1,)
float(1,)
AbstractFloat(1,)
Float64(1,)
sitofp(Float64, 1)
⋮ # skipped for brevity
+(-5.551115123125783e-17, 0.004375208149169746)
add_float(-5.551115123125783e-17, 0.004375208149169746)
+(0.8370957766587268, 0.004375208149169691)
add_float(0.8370957766587268, 0.004375208149169691)
0.8414709848078965
```

Example: Counting Calls

```
julia> using Cassette: @context, @overdub, @prehook

julia> mutable struct Count{T}
           count::Int
       end

julia> @context CountCtx

# Here we are dispatching on the type of the context's
# metadata to define a prehook that increments a counter
# every time one or more arguments of type `T` are
# encountered in the execution trace.
julia> @prehook function (::Any)(arg::T, args::T...)
           where {T,__CONTEXT__<:CountCtx{Count{T}}}
           __context__.metadata.count += 1
       end
```

```
# let's count the number of calls that have
# arguments that are subtypes of `Union{String,Int}`
julia> c = Count{Union{String,Int}}(0)
Count{Union{Int64, String}}(0)

julia> @overdub(CountCtx(metadata = c),
                map(string, 1:10))
10-element Array{String,1}:
 "1"
 "2"
 "3"
 "4"
 "5"
 "6"
 "7"
 "8"
 "9"
 "10"

julia> c
 Count{Union{Int64, String}}(1643)
```

Example: GPU primitives

```
using Cassette: @context, @overdub, @primitive

using CUDAnative, CuArrays

# Define a new context type `GPUCtx`.
@context GPUCtx

# Define some `GPUCtx` "primitives". If, while executing
# code in a GPU context, some method is encountered that
# matches the signature of one of these primitives, that
# method call will dispatch to the primitive definition
# provided here.
@primitive Base.tanh(x::Number) where
      {__CONTEXT__<:GPUCtx} = CUDAnative.tanh(x)
@primitive Base.exp(x::Number) where
      {__CONTEXT__<:GPUCtx} = CUDAnative.exp(x)
```

```
sigm(x) = 1.0 / (1.0 + exp(-x))

function hmlstm_kernel(z, zb, c, f, i, g)
   if z == 1 # FLUSH
       return sigm(i) * tanh(g)
   elseif zb == 0 # COPY
       return c
   else # UPDATE
       return sigm(f) * c + sigm(i) * tanh(g)
   end
end

n = 2048
z, zb = cu(rand(n)), cu(rand(n))
c, f, i, g = ntuple(i -> cu(rand(n, n)), 4)

# execute the given code in a `GPUCtx`.
@overdub(GPUCtx(), hmlstm_kernel.(z, zb, c, f, i, g))
```

Example: Literal Translation

```
using Cassette: @context, @overdub, @pass


is32bit(x) = false
is32bit(x::Integer) = (typemin(Int32) <= x <= typemax(Int32))
is32bit(x::AbstractFloat) = (typemin(Float32) <= x <= typemax(Float32))

to32bit(x::Integer) = convert(Int32, x)
to32bit(x::AbstractFloat) = convert(Float32, x)

@context Bit32Ctx

bit32pass = @pass (sig, codeinfo) -> begin
    # applies the first function to any piece of the
    # IR for which the second function returns `true`
    Cassette.replace_match!(to32bit, is32bit, codeinfo.code)
    return codeinfo
end

z, zb, c, f, i, g = rand(Float32, 6)

@overdub(Bit32Ctx(pass = bit32pass), hmlstm_kernel(z, zb, c, f, i, g))
```

Example: Nested Trace

```
julia> using Cassette: @context, @primitive, @overdub

julia> @context TraceCtx

julia> @primitive function (f::Any)(args...) where {__CONTEXT__<:TraceCtx}
           subtrace = Any[]
           push!(__context__.metadata, (f, args) => subtrace)
           if Cassette.is_core_primitive(__context__, f, args...)
               return f(args...)
           else
               newctx = Cassette.similar_context(__context__, metadata = subtrace)
               return Cassette.overdub_recurse(newctx, f, args...)
           end
       end

julia> trace = Any[]; x, y, z = rand(3);

julia> f(x, y, z) = x*y + y*z;

julia> @overdub(TraceCtx(metadata = trace), f(x, y, z));

julia> trace == Any[(f,(x,y,z)) => Any[
                        (*,(x,y)) => Any[(Base.mul_float,(x,y))=>Any[]]
                        (*,(y,z)) => Any[(Base.mul_float,(y,z))=>Any[]]
                        (+,(x*y,y*z)) => Any[(Base.add_float,(x*y,y*z))=>Any[]]]]
true
```

Nifty. But how does Cassette *work*?

```
@overdub(Ctx(), f(x))
```

```
overdub_recurse(Ctx(), f, x)
```

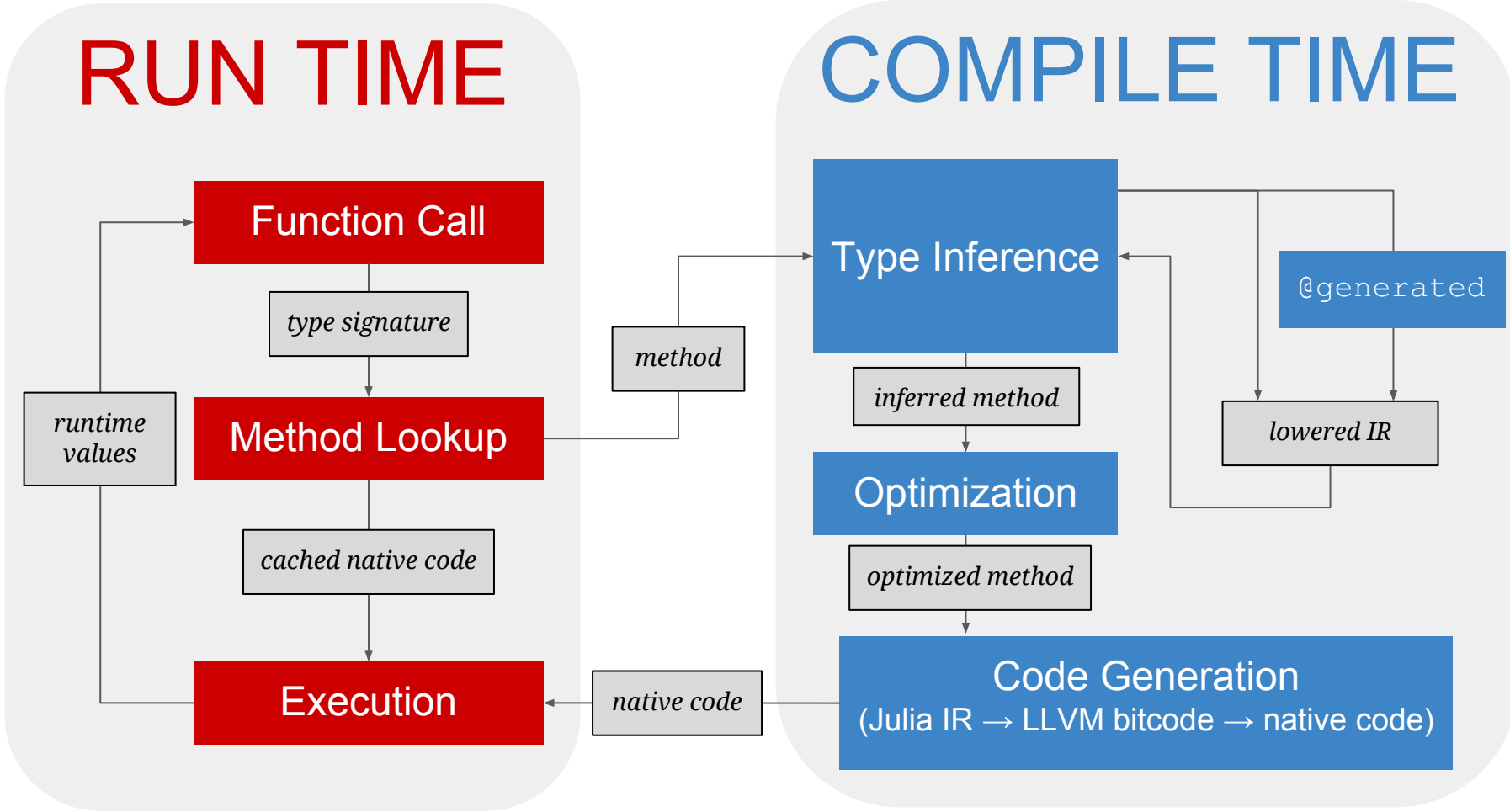
overdub_recurse

```
function f(x)
    T = eltype(x)
    n = length(x)
    result = zero(T)
    oneT = one(T)
    k = 100 * oneT
    for i in 1:n
        tmp1 = oneT - i
        tmp2 = k * tmp1
        result += tmp2
    end
    return result
end
```

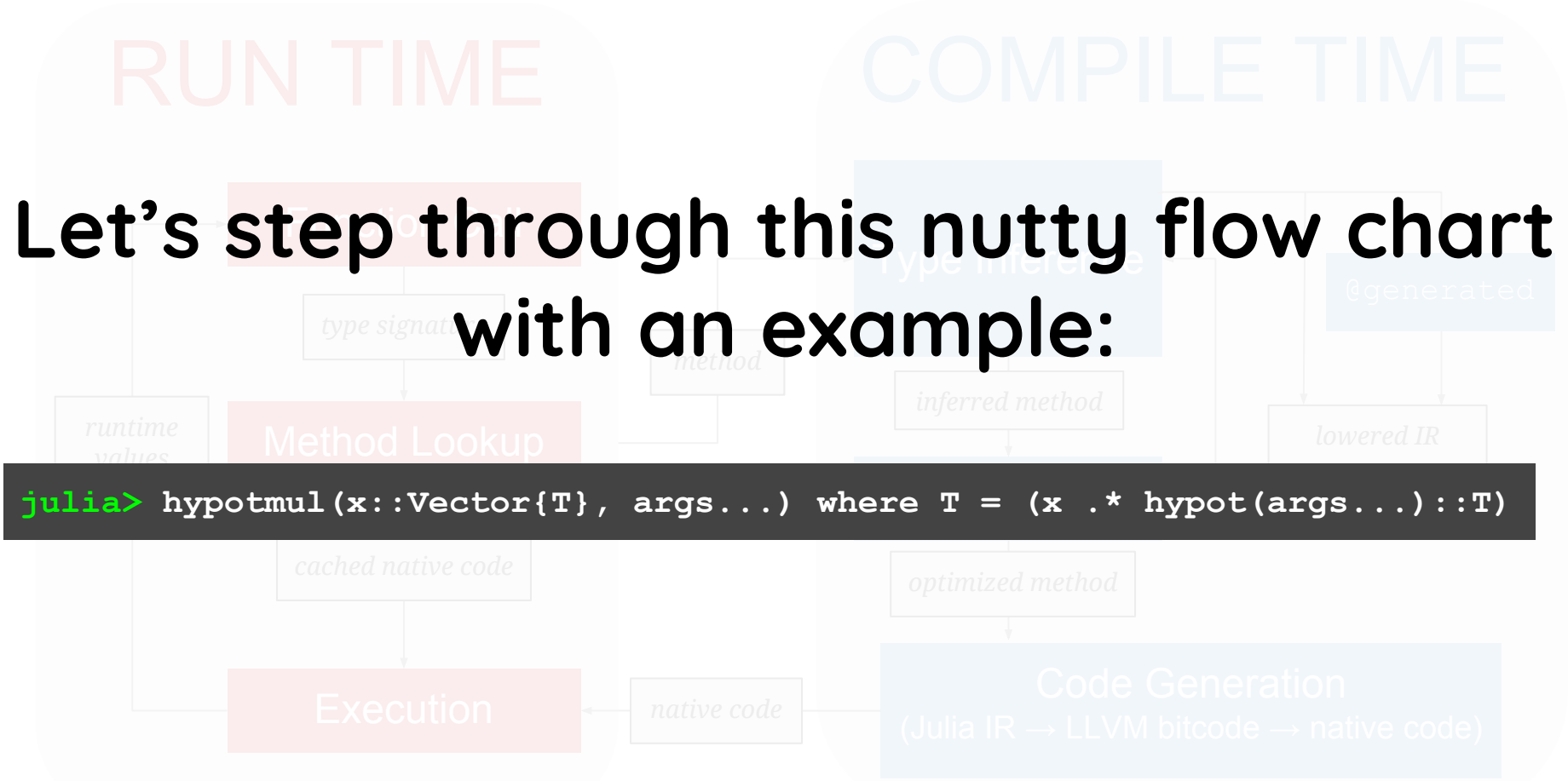
```
function overdub_recurse(ctx, ::typeof(f), x)
    T = overdub_execute(ctx, eltype, x)
    n = overdub_execute(ctx, length, x)
    result = overdub_execute(ctx, zero, T)
    oneT = overdub_execute(ctx, one, T)
    k = overdub_execute(ctx, *, 100, oneT)
    for i in overdub_execute(ctx, :, 1, n)
        tmp1 = overdub_execute(ctx, -, oneT, i)
        tmp2 = overdub_execute(ctx, *, k, tmp1)
        result = overdub_execute(ctx, +, result, tmp2)
    end
    return result
end
```

overdub_execute

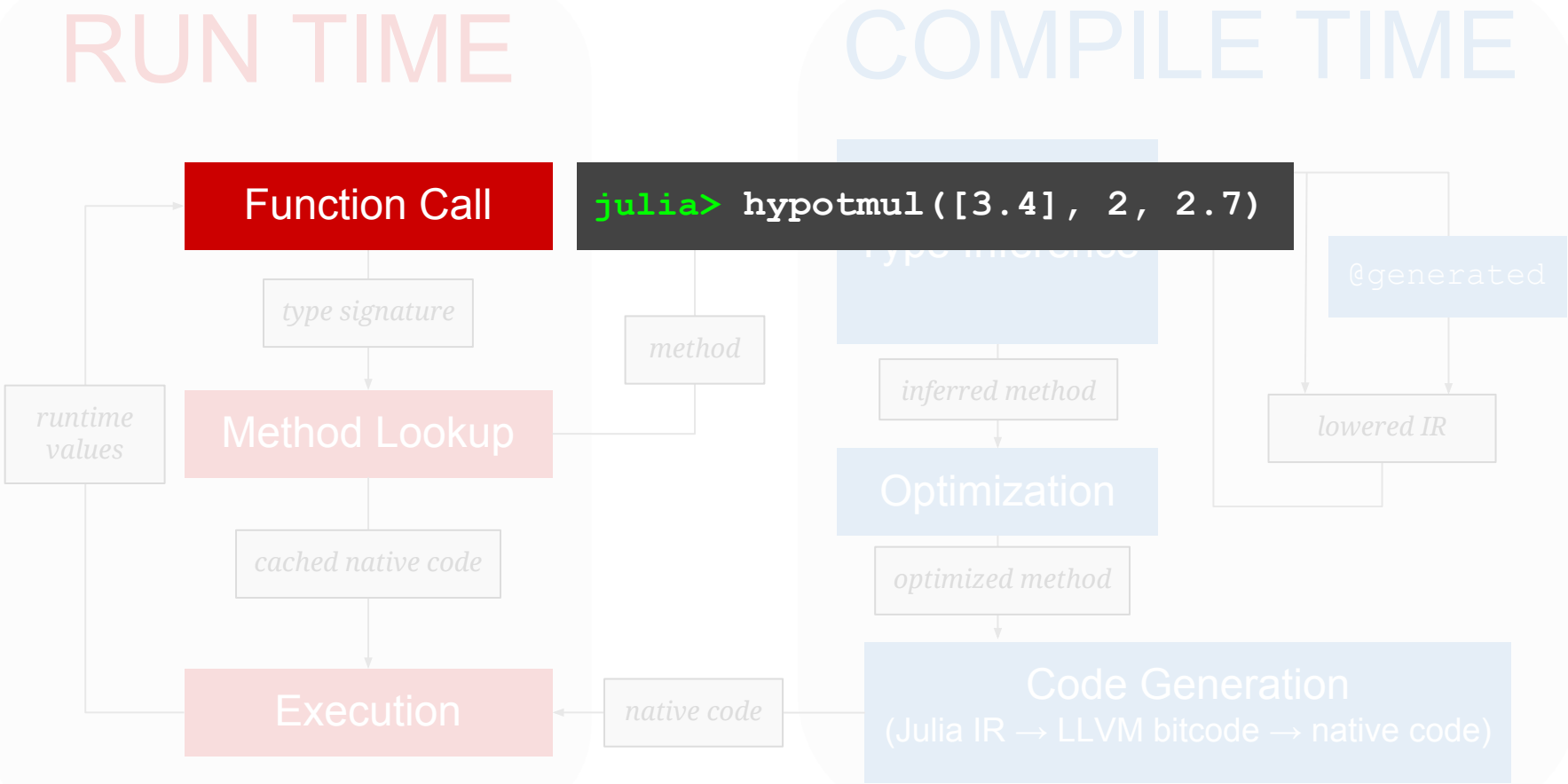
```
function overdub_execute(ctx, f, args...)
    prehook(ctx, f, args...)
    if is_user_primitive(ctx, f, args...)
        output = execution(ctx, f, args...)
    else
        output = overdub_recurse(ctx, f, args...)
    end
    posthook(ctx, output, f, args...)
    return output
end
```

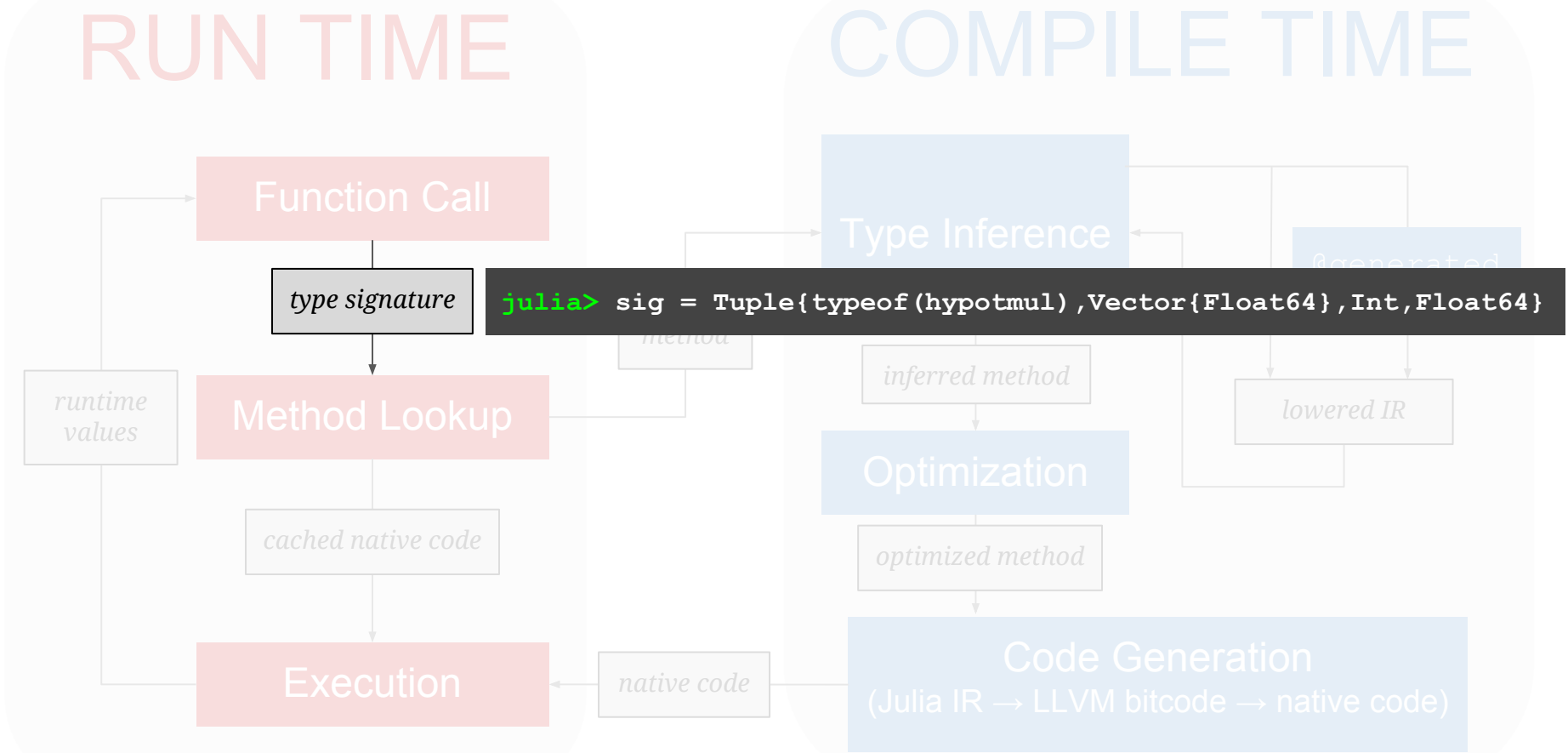

RUN TIME
COMPILE TIME
Function Call
type signature
Method Lookup
cached native code
Execution
runtime values
method
native code
Type Inference
inferred method
Optimization
optimized method
Code Generation
(Julia IR → LLVM bitcode → native code)
@generated
lowered IR

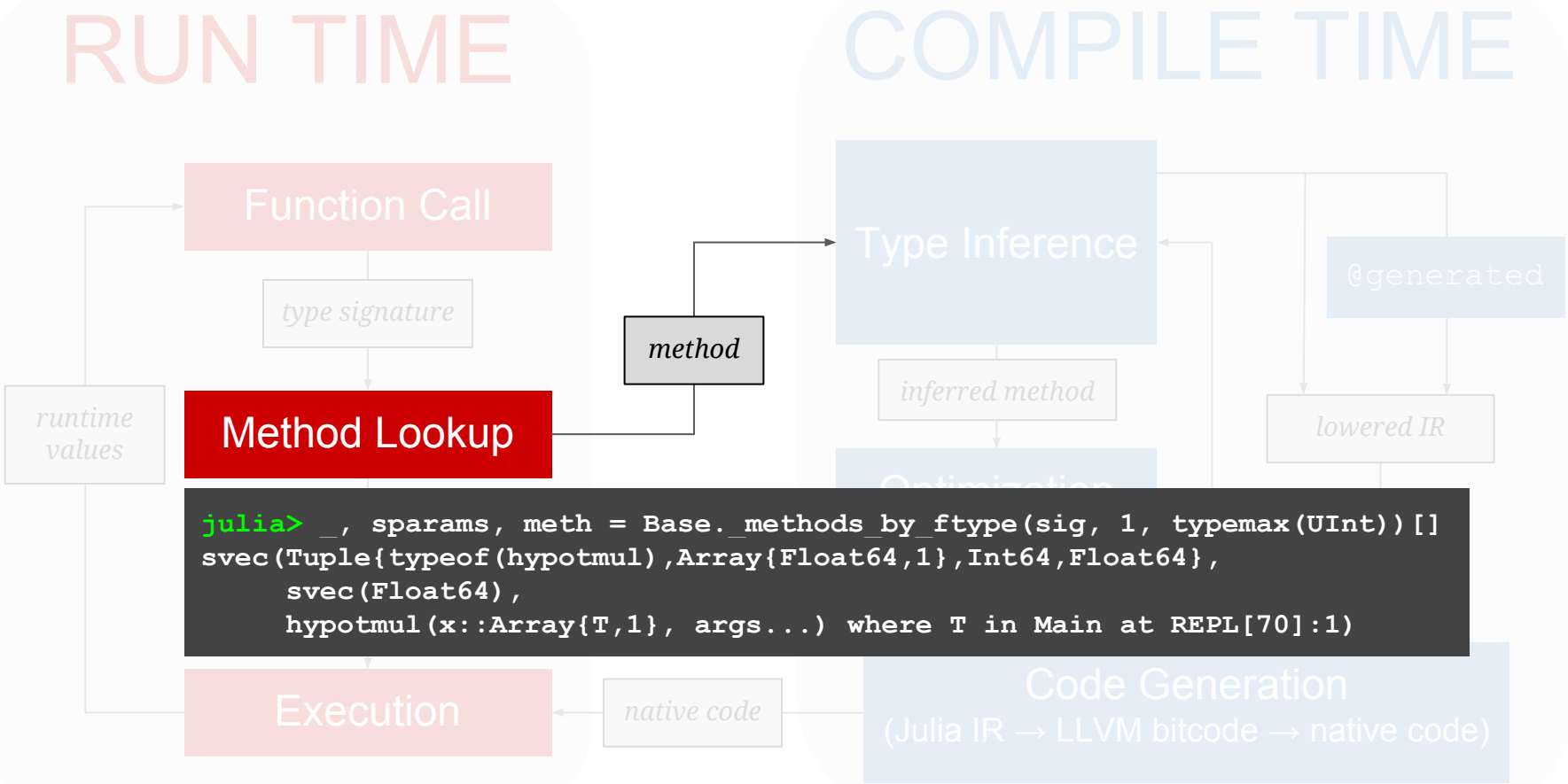
Let’s step through this nutty flow chart with an example:




```
julia> hypotmul(x::Vector{T}, args...) where T = (x .* hypot(args...)::T)
```


RUN TIME
COMPILE TIME
Function Call
julia> hypotmul([3.4], 2, 2.7)
type signature
method
@generated
runtime values
Method Lookup
inferred method
lowered IR
Optimization
cached native code
optimized method
Execution
native code
Code Generation
(Julia IR → LLVM bitcode → native code)


RUN TIME
COMPILE TIME
Function Call
type signature
julia> sig = Tuple{typeof(hypotmul),Vector{Float64},Int,Float64}
Type Inference
inferred method
lowered IR
Optimization
optimized method
Code Generation
(Julia IR → LLVM bitcode → native code)
native code
Execution
cached native code
Method Lookup
runtime values


RUN TIME
COMPILE TIME
Function Call
type signature
runtime values
Method Lookup
Execution
native code
method
Type Inference
inferred method
@generated
lowered IR
Code Generation
(Julia IR → LLVM bitcode → native code)
julia> _, sparams, meth = Base._methods_by_ftype(sig, 1, typemax(UInt))[]
svec(Tuple{typeof(hypotmul),Array{Float64,1},Int64,Float64},
svec(Float64),
hypotmul(x::Array{T,1}, args...) where T in Main at REPL[70]:1)

RUN TIME

COMPILE TIME

Function Call

type signature

method

Type Inference

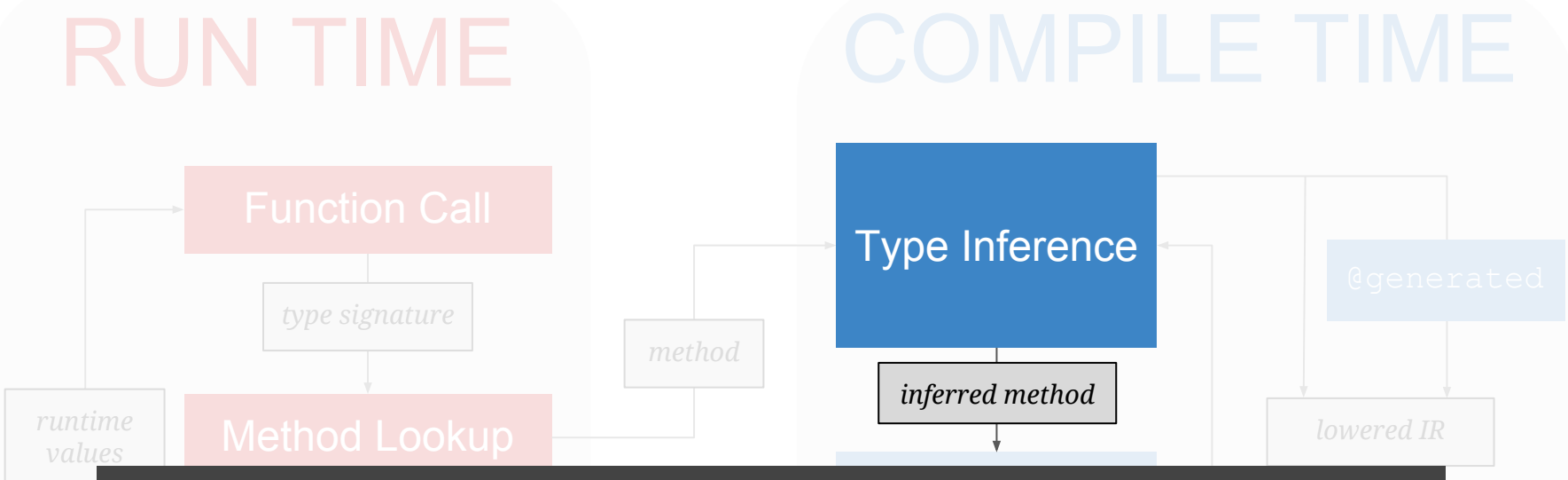
@generated

lowered IR

```
julia> Core.Compiler.uncompressed_ast(meth)
CodeInfo(:(begin
nothing
SSAValue(0)=Base.Broadcast.materialize
SSAValue(1)=Base.Broadcast.broadcasted
SSAValue(2)=(Core._apply)(Main.hypot, args)
SSAValue(3)=(Core.typeassert)(SSAValue(2), $(Expr(:static_parameter, 1)))
SSAValue(4)=(SSAValue(1))(Main.:*, x, SSAValue(3))
SSAValue(5)=(SSAValue(0))(SSAValue(4))
return SSAValue(5)
end))
```

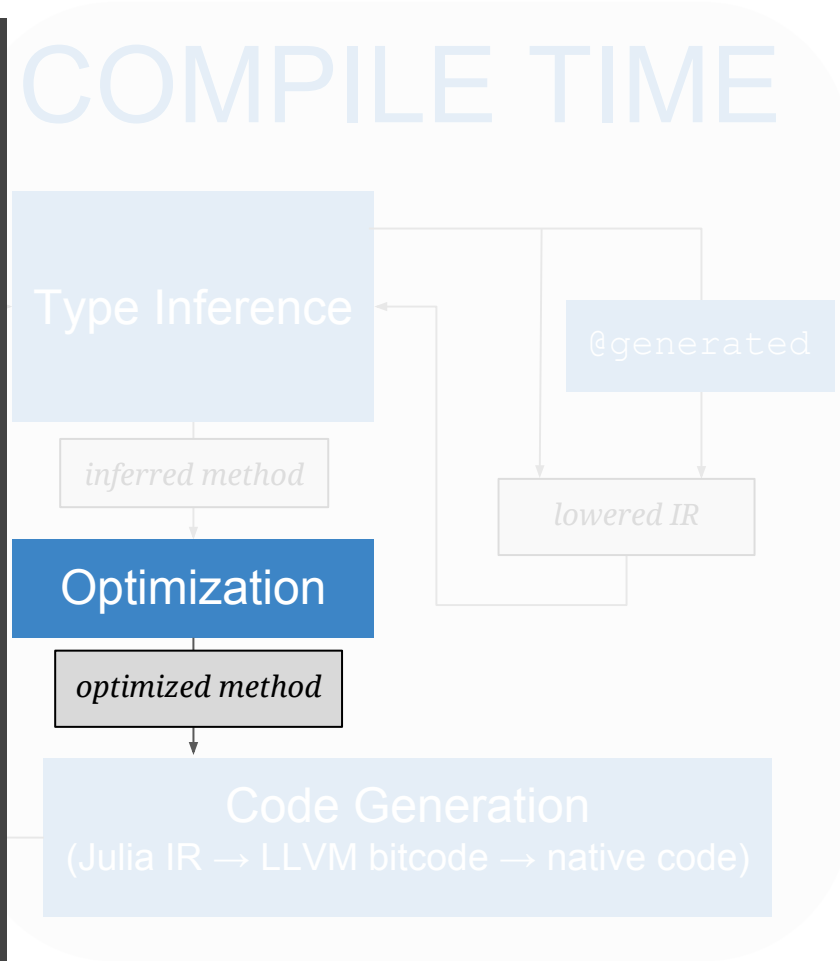
Generation

(Julia IR → LLVM bitcode → native code)




```
julia> code_typed(hypotmul, (Vector{Float64},Int,Float64), optimize=false)[]
CodeInfo(:(begin
nothing
SSAValue(0) = Base.Broadcast.materialize
SSAValue(1) = Base.Broadcast.broadcasted
SSAValue(2) = (Core._apply)(Main.hypot, args)::Float64
SSAValue(3) = (Core.typeassert)(SSAValue(2), $(Expr(:static_parameter, 1)))::Float64
SSAValue(4) = (SSAValue(1))(Main.:*, x, SSAValue(3))::Base.Broadcast.Broadcasted…
SSAValue(5) = (SSAValue(0))(SSAValue(4))::Array{Float64,1}
return SSAValue(5)
end)) => Array{Float64,1}
```

```
julia> code_typed(hypotmul, (Vector{Float64},Int,Float64),
                  optimize=true)[]
CodeInfo(:(begin
(getfield)(args, 1)::Int64
(getfield)(args, 2)::Float64
(Base.sitofp)(Float64, SSAValue(1))::Float64
$(Expr(:invoke, MethodInstance for hypot(::Float64…
⋮
$(Expr(:foreigncall, :(:jl_alloc_array_1d), Array{Float64,1},…
(Base.arraysize)(SSAValue(12), 1)::Int64
⋮
(SSAValue(17) === false)::Bool
unless SSAValue(18) goto 21
goto 22
Core.PhiNode(Any[20, 21], Any[false, true])
goto 24
unless SSAValue(22) goto 111
unless false goto 26
Core.PiNode(false, Bool)
unless SSAValue(26) goto 29
nothing
(SSAValue(12) === x)::Bool
unless SSAValue(29) goto 32
goto 45
$(Expr(:foreigncall, :(:jl_array_ptr), Ptr{Float64},
svec(Any),…
⋮
return SSAValue(12)
end)) => Array{Float64,1}
```



RUN TIME

```
julia> @code_llvm hypotmul([3.4], 2, 2.7)

; Function hypotmul
; Location: REPL[1]:1
define nonnull %jl_value_t addrspace(10)* @julia_hypotmul…
top:
  %gcframe = alloca %jl_value_t addrspace(10)*, i32 3
  %3 = bitcast %jl_value_t addrspace(10)** %gcframe to i8*
  call void @llvm.memset.p0i8.i32(i8* %3, i8 0, i32 24, i32…
  %4 = alloca i64, align 8
  %5 = call %jl_value_t*** inttoptr (i64 4480712320 to…
⋮
```

cached native code

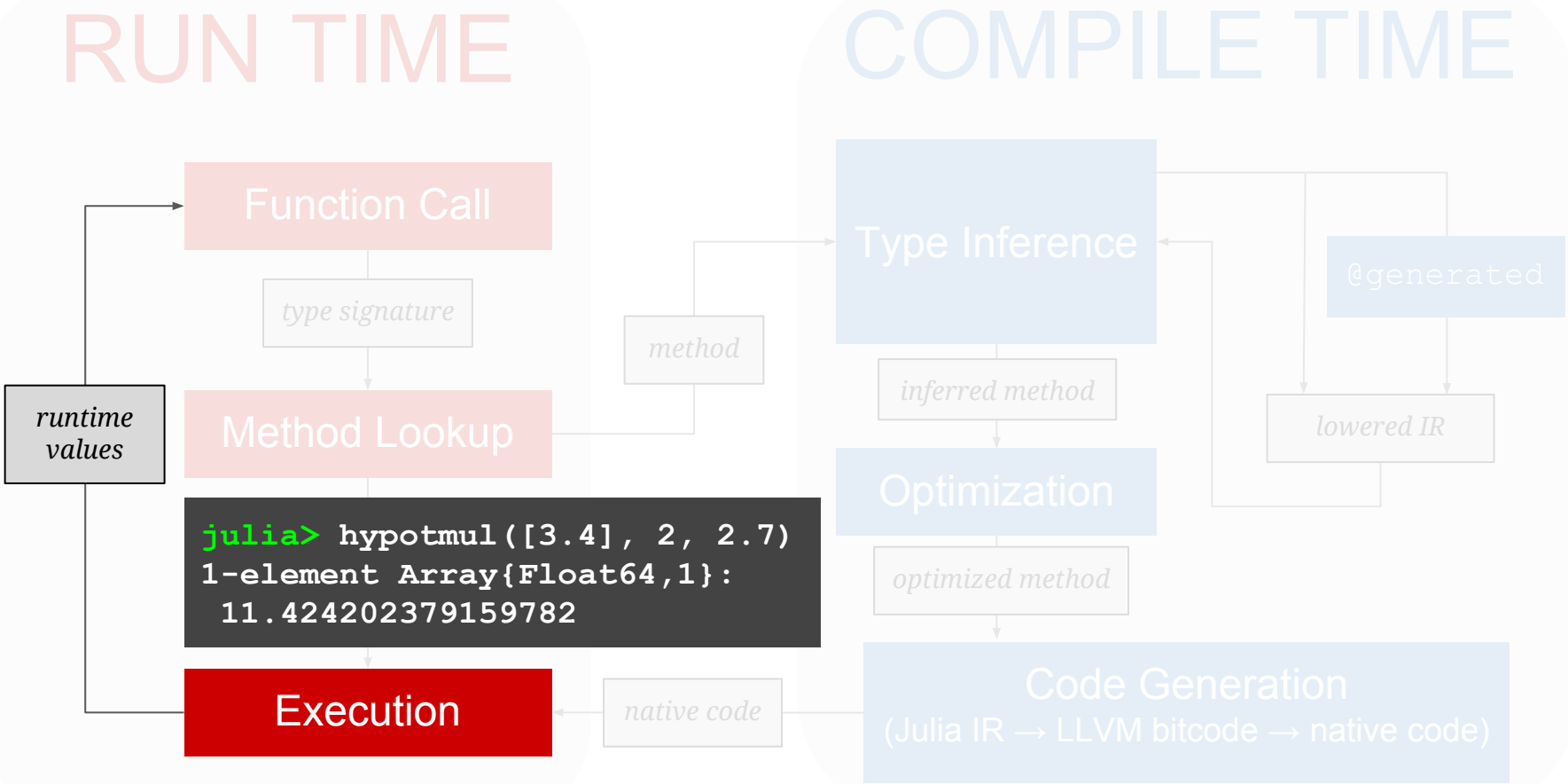
Execution

native code

```
julia> @code_native hypotmul([3.4], 2, 2.7)
        .section        __TEXT,__text,regular …
; Location: REPL[1]:1
        pushq   %rbp
        pushq   %r15
        pushq   %r14
        pushq   %r13
        pushq   %r12
        pushq   %rbx
        subq    $72, %rsp
        vmovsd  %xmm0, 32(%rsp)
        movq    %rsi, %r15
        movq    %rdi, %r12
        movabsq $jl_alloc_array_1d, %r13
        vxorps  %xmm0, %xmm0, %xmm0
        vmovaps %xmm0, (%rsp)
        movq    $0, 16(%rsp)
        leaq    153440(%r13), %rax

⋮
```

optimized method

Code Generation
(Julia IR → LLVM bitcode → native code)


RUN TIME
COMPILE TIME
Function Call
type signature
Method Lookup
runtime values
julia> hypotmul([3.4], 2, 2.7)
1-element Array{Float64,1}:
11.424202379159782
Execution
native code
method
Type Inference
@generated
inferred method
lowered IR
Optimization
optimized method
Code Generation
(Julia IR → LLVM bitcode → native code)

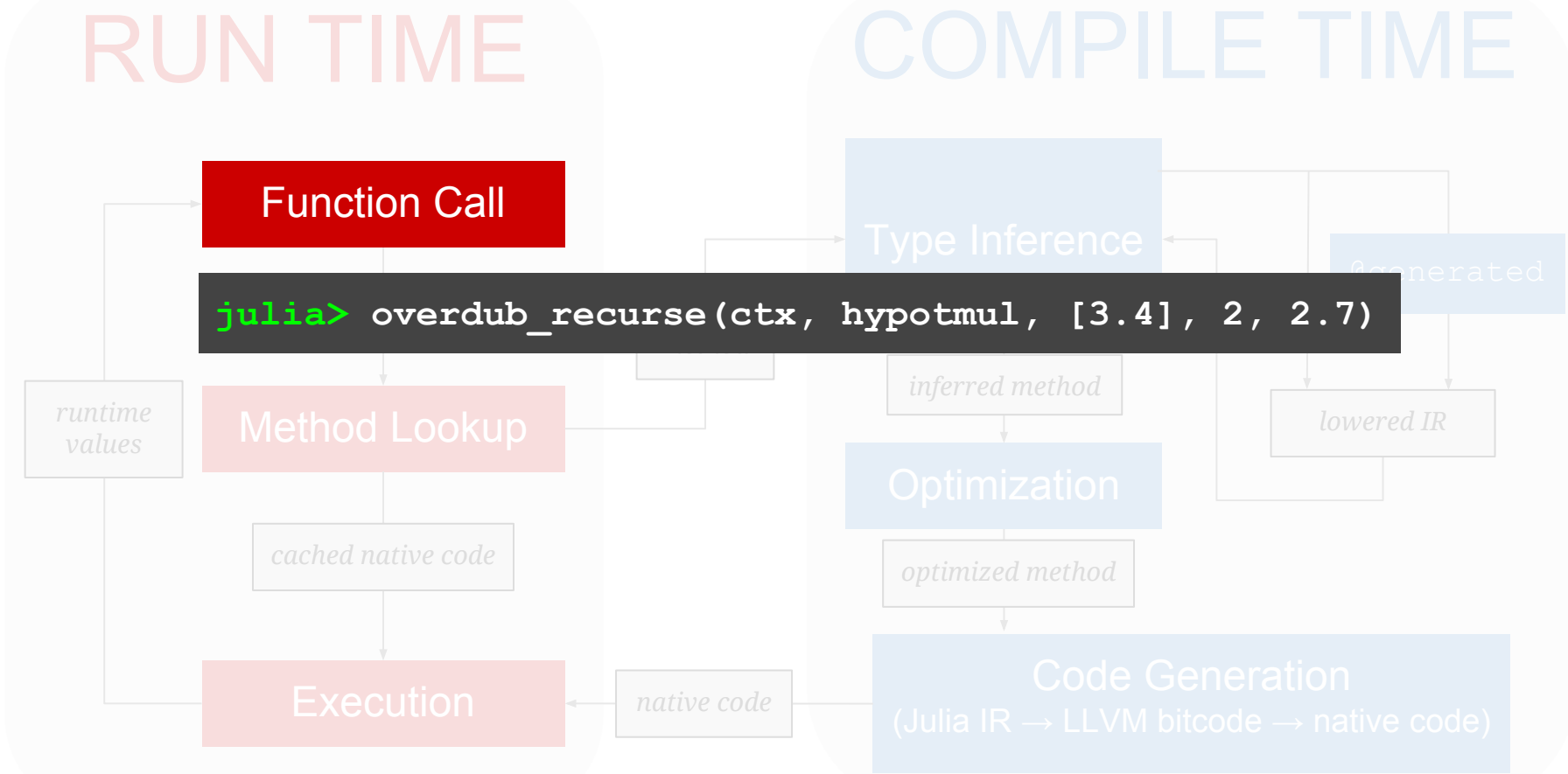
Let's "overdub" our example, and see where the cycle changes:

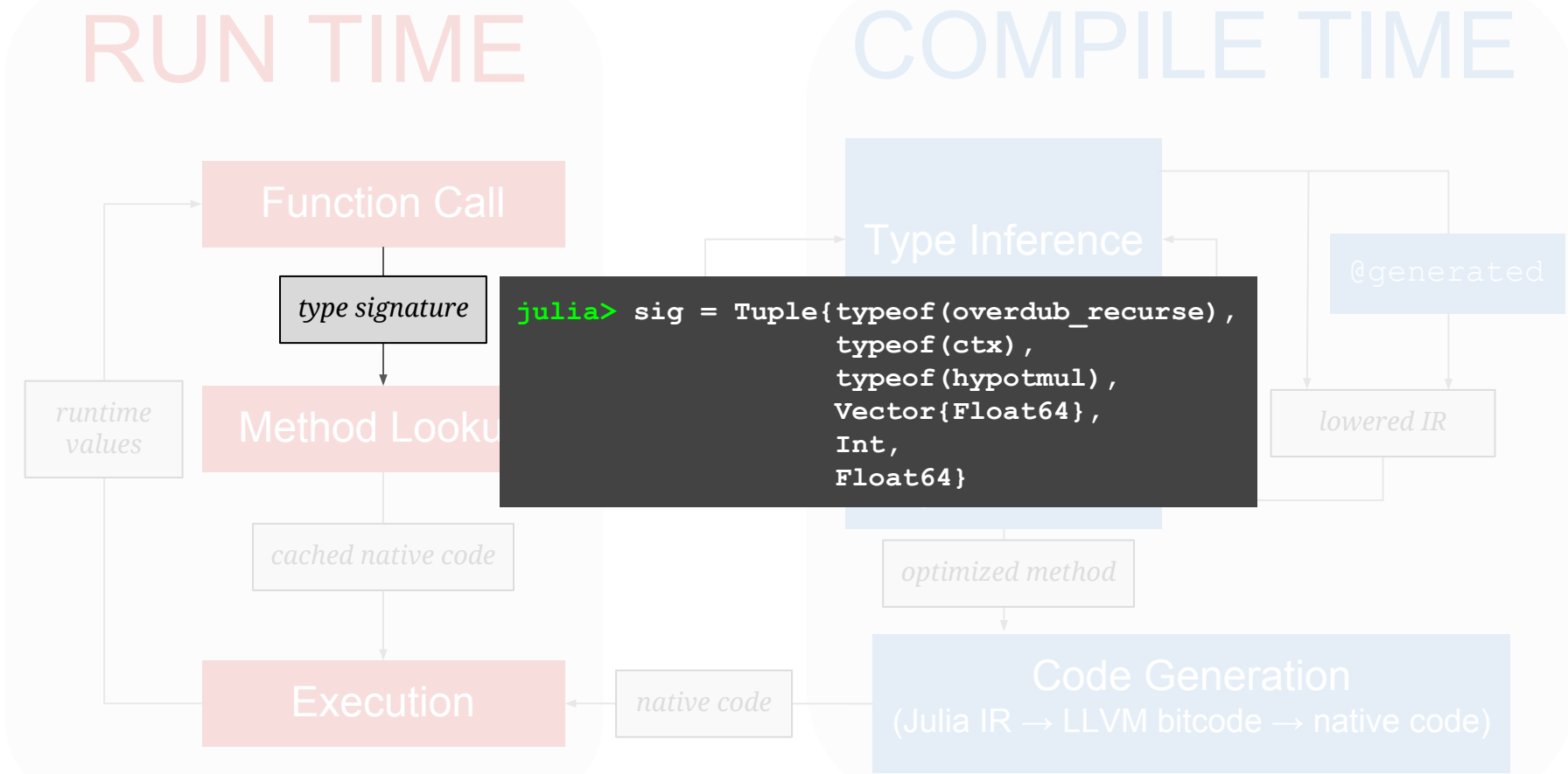
```
julia> using Cassette: @context, overdub_recurse

julia> @context Ctx # define a new context type

julia> ctx = Ctx();

# execute hypotmul([3.4], 2, 2.7) in our new Cassette context
julia> overdub_recurse(ctx, hypotmul, [3.4], 2, 2.7)
```

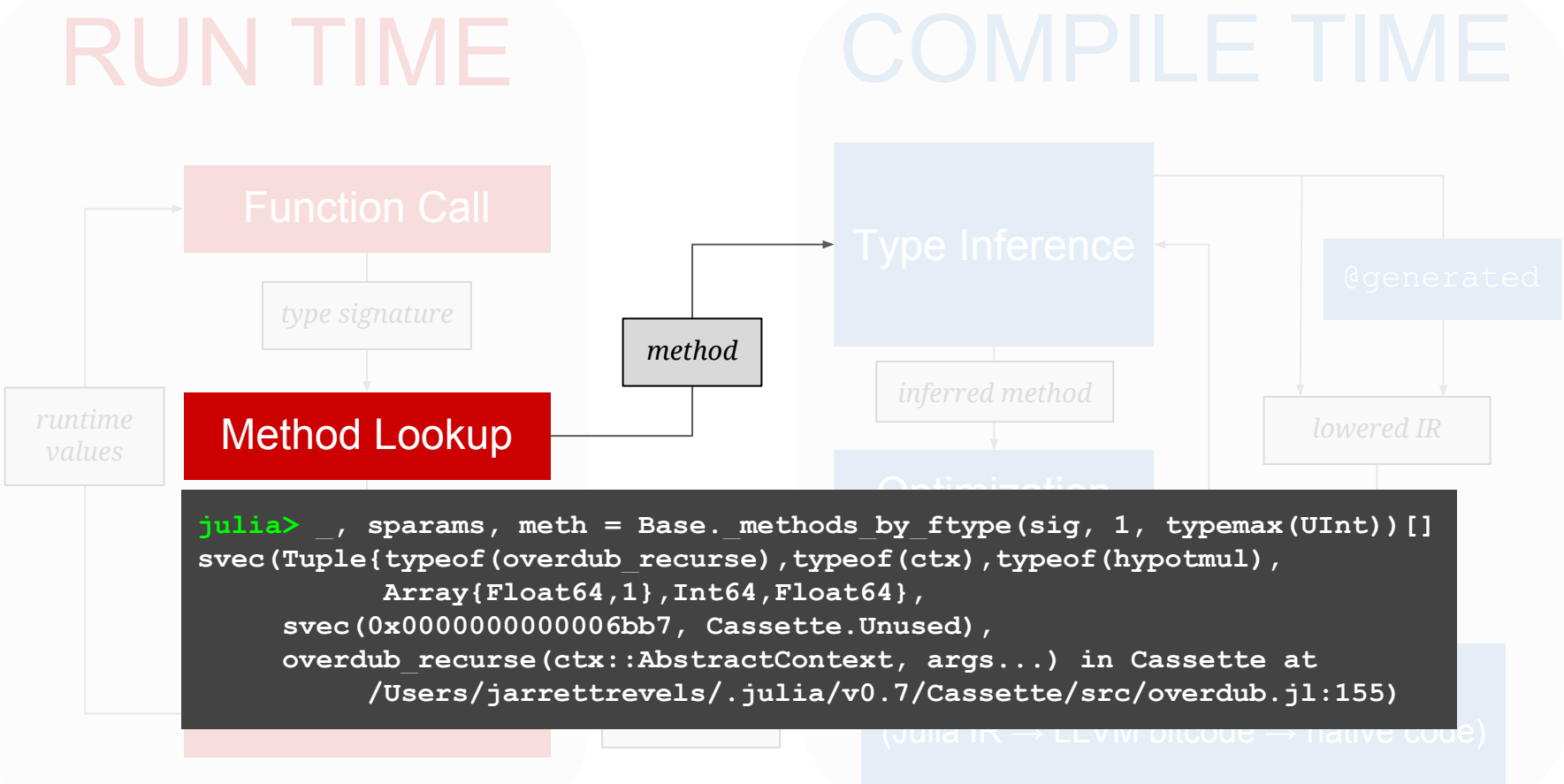
Recall our mental model for `overdub_recurse`

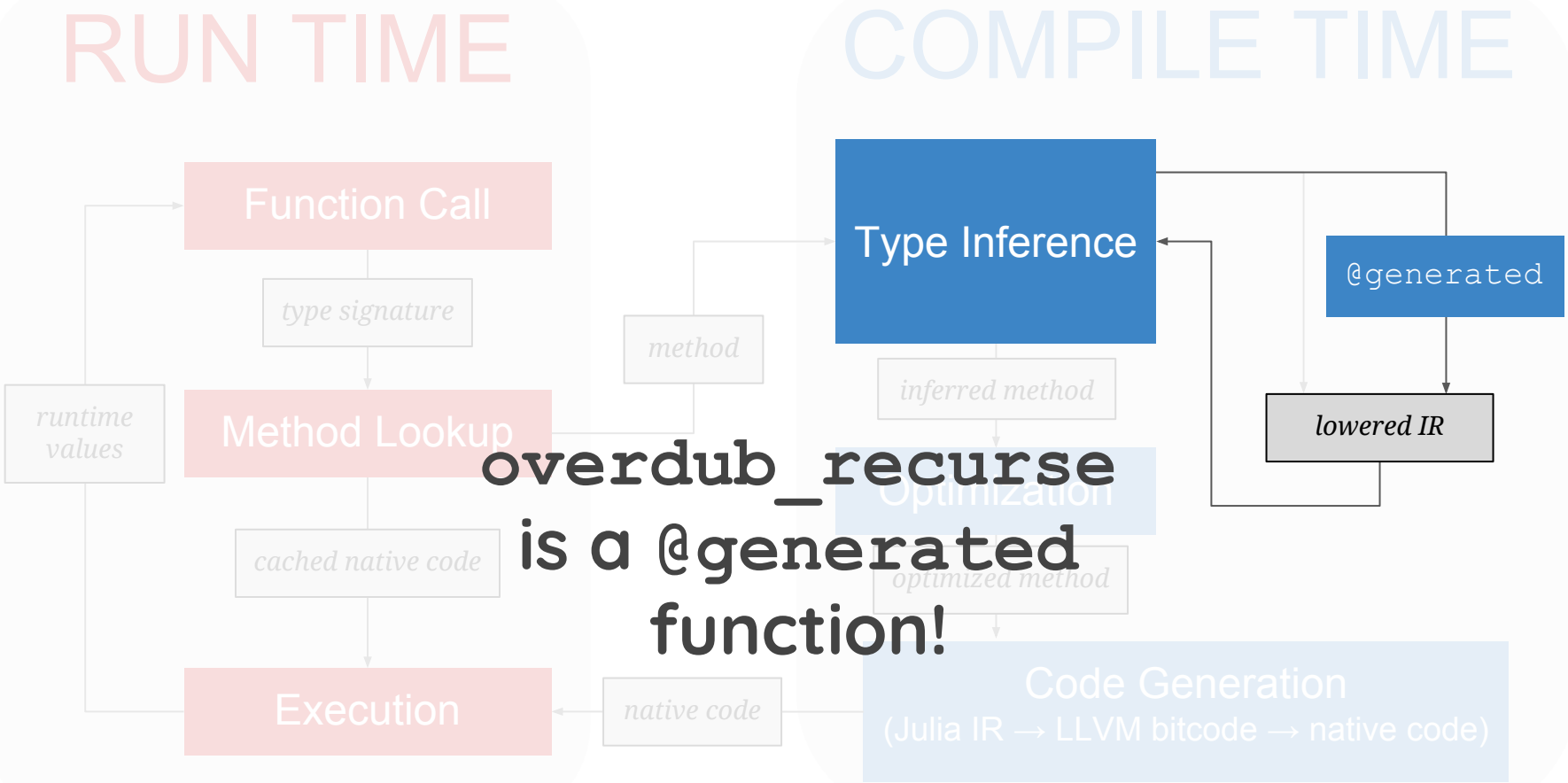
```
function f(x)
    T = eltype(x)
    n = length(x)
    result = zero(T)
    oneT = one(T)
    k = 100 * oneT
    for i in 1:n
        tmp1 = oneT - i
        tmp2 = k * tmp1
        result += tmp2
    end
    return result
end
```

```
function overdub_recurse(ctx, ::typeof(f), x)
    T = overdub_execute(ctx, eltype, x)
    n = overdub_execute(ctx, length, x)
    result = overdub_execute(ctx, zero, T)
    oneT = overdub_execute(ctx, one, T)
    k = overdub_execute(ctx, *, 100, oneT)
    for i in overdub_execute(ctx, :, 1, n)
        tmp1 = overdub_execute(ctx, -, oneT, i)
        tmp2 = overdub_execute(ctx, *, k, tmp1)
        result = overdub_execute(ctx, +, result, tmp2)
    end
    return result
end
```


RUN TIME
COMPILE TIME
Function Call
julia> overdub_recurse(ctx, hypotmul, [3.4], 2, 2.7)
Type Inference
generated
inferred method
lowered IR
runtime values
Method Lookup
Optimization
cached native code
optimized method
Execution
native code
Code Generation
(Julia IR → LLVM bitcode → native code)


RUN TIME
COMPILE TIME
Function Call
type signature
Method Looku
runtime values
cached native code
Execution
native code
Type Inference
@generated
lowered IR
optimized method
Code Generation
(Julia IR → LLVM bitcode → native code)
julia> sig = Tuple{typeof(overdub_recurse),
typeof(ctx),
typeof(hypotmul),
Vector{Float64},
Int,
Float64}

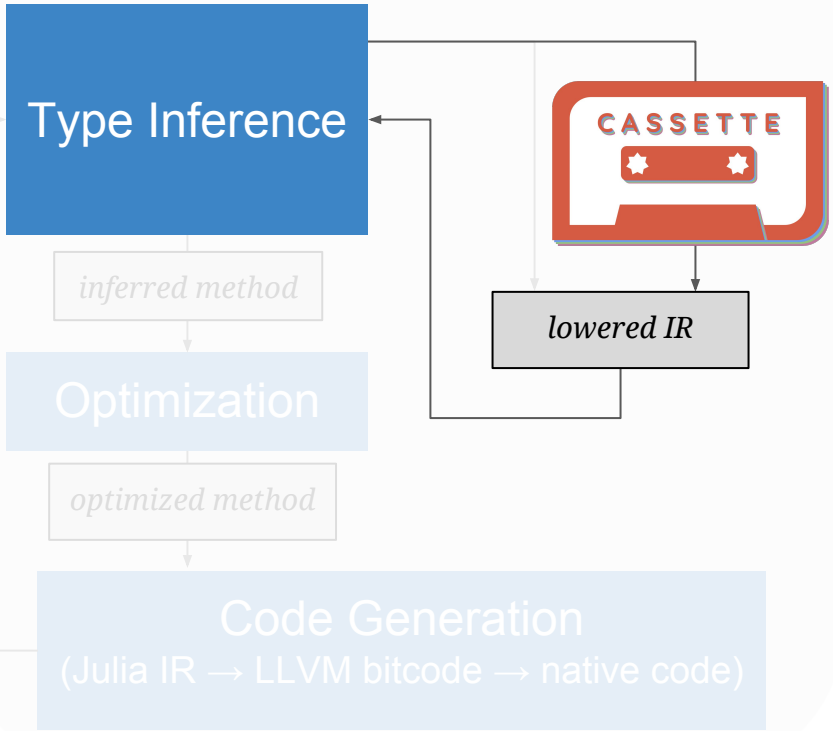

RUN TIME
COMPILE TIME
Function Call
type signature
runtime values
Method Lookup
method
Type Inference
inferred method
@generated
lowered IR
julia> _, sparams, meth = Base._methods_by_ftype(sig, 1, typemax(UInt))[]
svec(Tuple{typeof(overdub_recurse),typeof(ctx),typeof(hypotmul),
Array{Float64,1},Int64,Float64},
svec(0x0000000000006bb7, Cassette.Unused),
overdub_recurse(ctx::AbstractContext, args...) in Cassette at
/Users/jarrettrevels/.julia/v0.7/Cassette/src/overdub.jl:155)


RUN TIME
COMPILE TIME
Function Call
type signature
Method Lookup
cached native code
Execution
runtime values
method
native code
Type Inference
inferred method
Optimization
optimized method
Code Generation
(Julia IR → LLVM bitcode → native code)
@generated
lowered IR
overdub_recurse is a @generated function!

RUN TIME

COMPILE TIME

```
julia> mi = Core.Compiler.code_for_method(meth,sig,sparams,…
MethodInstance for overdub_recurse(::typeof(ctx),…

julia> mi.def.generator(typemax(UInt), …
CodeInfo(:(begin
nothing
#self#=(Core.getfield)(args, 1)
x=(Core.getfield)(args, 2)
SSAValue(6)=(Core.getfield)(args, 3)
SSAValue(7)=(Core.getfield)(args, 4)
args=(Core.tuple)(SSAValue(6), SSAValue(7))
SSAValue(0)=Base.Broadcast.materialize
SSAValue(1)=Base.Broadcast.broadcasted
SSAValue(2)=(overdub_execute)(ctx, _apply, Main.hypot, args)
SSAValue(3)=(overdub_execute)(ctx, typeassert, SSAValue(2),…
SSAValue(4)=(overdub_execute)(ctx, SSAValue(1), Main.:*, x,…
SSAValue(5)=(overdub_execute)(ctx, SSAValue(0), SSAValue(4))
return SSAValue(5)
end))
```



Defining `overdub_recurse`

```
mutable struct Reflection
    signature::DataType
    method::Method
    static_params::Vector{Any}
    code_info::CodeInfo
end


function reflect(signature::Tuple)::Union{Reflection,Nothing}
    if method_exists(signature)
        method, sparams, cinfo = get_method_and_info(signature)
        return Reflection(signature, method, sparams, cinfo)
    end
    return nothing
end
```

Defining overdub_recurse

```
@generated function overdub_recurse(ctx, args...)
    ref = reflect(args)
    if isa(ref, Reflection) # if we find a method
        pass = pass_type_from_context_type(ctx)
        if pass <: Cassette.AbstractPass # if the user gave us a valid pass
            ref.code_info = pass(ref.signature, ref.code_info)
        end
        replace_calls_with_overdub_execute!(ref)
        body = ref.code_info
    else # if we didn't find a method
        body = quote
            Cassette.execution(ctx, args...)
        end
    end
    return body
end
```

Some details I conveniently omitted

- propagating world ages as type parameters
- pass type definition actually requires overloading the generator
- spoofing inference recursion limiting heuristics

The Future

- expect a Cassette release in the Julia 1.x timeframe; hopefully before JuliaCon 2018.

- Metadata for trace values; I didn't show it off because it's not ready yet, but cool stuff is planned

- Cassette-based tools for AD, GPU transpilation, interval constraint programming

- Julia IR/compiler stdlib?

Thanks to EVERYBODY

- Prof. Juan Pablo Vielma, Sloan (my PI)
- Peter Ahrens and Valentin Churavy (officemates)
- Jeff Bezanson, Keno Fischer, and Jameson Nash (Julia Computing compiler team)
- Tim Besard (GPU expert and Cassette bug-hunter)
- All you beautiful folks and the (even more!) beautiful organizers