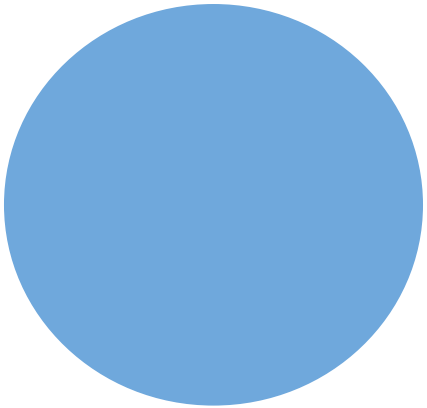
Jarrett Revels, MIT

First Things First

many julia packages try to fit a

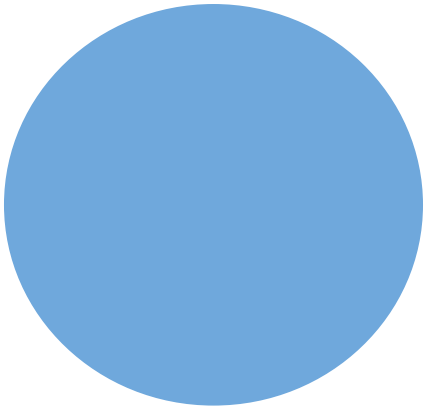
peg into a



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 Is Standard

- Common way to implement these in most languages that support polymorphic/generically overloadable functions

- Often requires additional manual implementation of limited forms of multiple dispatch

- Julia is actually really, really good at this

This Approach Stinks

- Overloading-based approaches **often thwarted by** dispatch and/or structural **type constraints**

- Proper usage requires proper genericity criteria, i.e. "**What weird subset of Julia do I really support?**"

- Many relevant language mechanisms **not interceptable via method overloading** (control flow, literals, bindings, calling scope).

- Composing these abuses of multiple dispatch leads to **crazy method ambiguities**. Known brute-force solutions generate too many methods and/or depend on package load order.

Example: Simple Logging

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

julia> @context PrintCtx

julia> prehook(::PrintCtx, f, args...) = println(f, args)

julia> @overdub(PrintCtx(), 1/2)
/(1, 2)
float(1,)
AbstractFloat(1,)
Float64(1,)
sitofp(Float64, 1)
float(2,)
AbstractFloat(2,)
Float64(2,)
sitofp(Float64, 2)
/(1.0, 2.0)
div_float(1.0, 2.0)
0.5
```

Example: Counting Calls

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

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

julia> @context CountCtx

julia> function prehook(ctx::CountCtx{Count{T}}, ::Any, ::T, ::Any...) where T
          ctx.metadata.count += 1
       end

julia> c = Count{DataType}(0)
Count{DataType}(0)

julia> @overdub(CountCtx(metadata = c), 1/2)
0.5

julia> c
Count{DataType}(2)
```

Example: GPU Primitives

```
import Cassette: @context, @overdub, execute

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.
function execute(::GPUCtx, ::typeof(Base.tanh), x::Number)
    return CUDAnative.tanh(x)
end

function execute(::GPUCtx, ::typeof(Base.exp), x::Number)
    return CUDAnative.exp(x)
end
```

```
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: @pass

fitsin32bit(x) = false
fitsin32bit(x::Integer) = (typemin(Int32) <= x <= typemax(Int32))
fitsin32bit(x::AbstractFloat) = (typemin(Float32) <= x <= typemax(Float32))

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

bit32pass = @pass (ctxtype, method_signature, method_body) -> begin
    # applies the first function to any piece of the
    # IR for which the second function returns `true`
    Cassette.replace_match!(to32bit, fitsin32bit, method_body.code)
    return method_body
end

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

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

Example: Nested Tracing

```
julia> import Cassette: @context, execute, @overdub, overdub, canoverdub, similarcontext, fallback

julia> @context TraceCtx;

julia> function execute(ctx::TraceCtx, args...)
           subtrace = Any[]
           push!(ctx.metadata, args => subtrace)
           if canoverdub(ctx, args...)
               newctx = similarcontext(ctx, metadata = subtrace)
               return overdub(newctx, args...)
           else
               return fallback(ctx, args...)
           end
       end
execute (generic function with 24 methods)

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

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

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
```

Example: Function Slicing (“Julia Continuations”)

```
import Cassette: @context, overdub, execute

@context SliceCtx

mutable struct Callback
    f::Any
end

function execute(ctx::SliceCtx,
                 ::typeof(println),
                 args...)
    previous = ctx.metadata.f
    ctx.metadata.f = () -> begin
        previous()
        println(args...)
    end
    return nothing
end
```

```
julia> begin
           a = rand()
           b = rand()
           function add(a, b)
               println("I'm about to add $a + $b")
               c = a + b
               println("c = $c")
               return c
           end
           add(a, b)
       end
I'm about to add 0.7570148782668673 + 0.28327047810025685
c = 1.0402853563671242
1.0402853563671242

julia> ctx = SliceCtx(metadata = Callback(() -> nothing));

julia> c = overdub(ctx, add, a, b)
1.0402853563671242

julia> ctx.metadata.f()
I'm about to add 0.7570148782668673 + 0.28327047810025685
c = 1.0402853563671242
```

Example: Function Slicing (“Julia Continuations”)

```
function add(a, b)
    println("I'm about to add $a + $b")
    c = a + b
    println("c = $c")
    return c
end
```

```
function overdub(ctx::SliceCtx, add, a, b)
    _callback_ = ctx.metadata
    _, _callback_ = execute(ctx, _callback_, println, "I'm about to add $a + $b")
    c, _callback_ = execute(ctx, _callback_, +, a, b)
    _, _callback_ = execute(ctx, _callback_, println, "c = $c")
    return c, _callback_
end
```

Example: Function Slicing (“Julia Continuations”)

```
import Cassette: @context, @pass, execute, canoverdub, similarcontext, overdub, fallback

@context SliceCtx

function execute(ctx::SliceCtx, callback, f, args...)
   if canoverdub(ctx, f, args...)
       _ctx = similarcontext(ctx, metadata = callback)
       return overdub(_ctx, f, args...) # return result, callback
   else
       return fallback(ctx, f, args...), callback
   end
end

function execute(ctx::SliceCtx, callback, ::typeof(println), args...)
   return nothing, () -> (callback(); println(args...))
end

function sliceprintln(::Type{<:SliceCtx}, signature, ir::CodeInfo)
   # 1 At the beginning of `ir`, insert something like `_callback_ = context.metadata`
   # 2 Change every method invocation of the form `f(args...)` to `_callback_(f, args...)`.
   # 3 Ensure the output of every method invocation is properly destructured into the original
   #   assignment slot/SSAValue and the `_callback_` slot.
   # 4 Change every return statement of the form `return x` to `return (x, _callback_)`
   return ir
end

const sliceprintlnpass = @pass sliceprintln
```

Example: Function Slicing (“Julia Continuations”)

```
julia> begin
           a, b = rand(2)
           function add(a, b)
               println("I'm about to add $a + $b")
               c = a + b
               println("c = $c")
               return c
           end
           add(a, b)
       end
I'm about to add 0.10586214325731103 + 0.02529007116348958
c = 0.1311522144208006
0.1311522144208006

julia> ctx = SliceCtx(pass=sliceprintlnpass, metadata = () -> nothing);

julia> result, callback = Cassette.overdub(ctx, add, a, b)
(0.1311522144208006, getfield(Main, Symbol("##4#5")){...}(...))

julia> callback()
I'm about to add 0.10586214325731103 + 0.02529007116348958
c = 0.1311522144208006
```

A Mental Model For `overdub`

```
f(args...)
```

```
begin
    prehook(context, f, args...)
    tmp = execute(context, f, args...)
    if isa(tmp, Cassette.OverdubInstead)
        tmp = overdub(context, f, args...)
    end
    posthook(context, tmp, f, args...)
    tmp
end
```

A Mental Model For `overdub`

```
julia> @code_lowered 1 / 2
CodeInfo(
59 1 ─ %1 = (Base.float)(x)
   │    %2 = (Base.float)(y)
   │    %3 = %1 / %2
   └──       return %3
)
```

A Mental Model For `overdub`

```
julia> @code_lowered overdub(Ctx(), /, 1, 2)
CodeInfo(
59 1 ─       #self# = (Core.getfield)(_args_, 1)
   │         x = (Core.getfield)(_args_, 2)
   │         y = (Core.getfield)(_args_, 3)
   │         (Cassette.prehook)(_context_, Base.float, x)
   │         _tmp_ = (Cassette.execute)(_context_, Base.float, x)
   │    %6  = _tmp_ isa Cassette.OverdubInstead
   └──       goto #3 if not %6
  2 ─       _tmp_ = (Cassette.overdub)(_context_, Base.float, x)
  3 ─       (Cassette.posthook)(_context_, _tmp_, Base.float, x)
   │    %10 = _tmp_
   │         (Cassette.prehook)(_context_, Base.float, y)
   │         _tmp_ = (Cassette.execute)(_context_, Base.float, y)
   │    %13 = _tmp_ isa Cassette.OverdubInstead
   └──       goto #5 if not %13
  4 ─       _tmp_ = (Cassette.overdub)(_context_, Base.float, y)
  5 ─       (Cassette.posthook)(_context_, _tmp_, Base.float, y)
   │    %17 = _tmp_
   │         (Cassette.prehook)(_context_, Base.:/, %10, %17)
   │         _tmp_ = (Cassette.execute)(_context_, Base.:/, %10, %17)
   │    %20 = _tmp_ isa Cassette.OverdubInstead
   └──       goto #7 if not %20
  6 ─       _tmp_ = (Cassette.overdub)(_context_, Base.:/, %10, %17)
  7 ─       (Cassette.posthook)(_context_, _tmp_, Base.:/, %10, %17)
   │    %24 = _tmp_
   └──       return %24
)
```

Cassette's Contextual Tagging System

- On top of the overdubbing mechanism, Cassette supports "tagging" arbitrary Julia values with a context and metadata

- Values tagged w.r.t. a context behave just like their untagged selves when propagating through a program overdubbed with that context

- Tagged values can propagate even through concrete type constraints (dispatch constraints, struct field constraints, etc.)

- Special care is given to the tagging system to allow for safe nested contextual execution - as long as the context author follows the rules, there should be no metadata confusion between contexts!

Example: Weird Identity

```
struct Bar{X,Y,Z}
    x::X
    y::Y
    z::Z
end

mutable struct Foo
    a::Bar{Int}
    b
end

function foo_bar_identity(x)
    bar = Bar(x, x + 1, x + 2)
    foo = Foo(bar, "ha")
    foo.b = bar
    foo.a = Bar(4,5,6)
    foo2 = Foo(foo.a, foo.b)
    foo2.a = foo2.b
    array = Float64[]
    push!(array, foo2.a.x)
    return [array[1]][1]
end

v, m = 1, 2
```

Example: Weird Identity

```
struct Bar{X,Y,Z}
    x::X
    y::Y
    z::Z
end

mutable struct Foo
    a::Bar{Int}
    b
end

function foo_bar_identity(x)
    bar = Bar(x, x + 1, x + 2)
    foo = Foo(bar, "ha")
    foo.b = bar
    foo.a = Bar(4,5,6)
    foo2 = Foo(foo.a, foo.b)
    foo2.a = foo2.b
    array = Float64[]
    push!(array, foo2.a.x)
    return [array[1]][1]
end

v, m = 1, 2
```

```
julia> import Cassette: @context, enabledtagging, overdub, tag,
                        untag, metadata, metadatatype

julia> @context FooBarCtx

julia> metadatatype(::Type{<:FooBarCtx}, ::Type{T}) where T<:Number = T

julia> ctx = enabledtagging(FooBarCtx(), foo_bar_identity);

julia> tagged = tag(v, ctx, m)
Tagged(Tag{nametype(FooBarCtx),2736618262450864357,Nothing}(), 1, Meta(2, _))

julia> result = overdub(ctx, foo_bar_identity, tagged)
Tagged(Tag{nametype(FooBarCtx),2736618262450864357,Nothing}(), 1.0, Meta(2.0, _))

julia> untag(result, ctx) === float(v)
true

julia> metadata(result, ctx) === float(m)
true
```

Example: Forward-Mode AD

```
import Cassette: @context, execute, Tagged, tag, untag, enabletagging,
                 overdub, metadata, hasmetadata, metadatatype

@context DiffCtx

const DiffCtxWithTag{T} = DiffCtx{Nothing,T}

metadatatype(::Type{<:DiffCtx}, ::Type{T}) where {T<:Real} = T

tangent(x, context) = hasmetadata(x, context) ? metadata(x, context) : zero(untag(x, context))

function D(f, x)
   ctx = enabletagging(DiffCtx(), f)
   result = overdub(ctx, f, tag(x, ctx, oftype(x, 1.0)))
   return tangent(result, ctx)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(sin), x::Tagged{T,<:Real}) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   return tag(sin(vx), ctx, cos(vx) * dx)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(cos), x::Tagged{T,<:Real}) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   return tag(cos(vx), ctx, -sin(vx) * dx)
end
```

```
function execute(ctx::DiffCtxWithTag{T}, ::typeof(*), x::Tagged{T,<:Real}, y::Tagged{T,<:Real}) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   vy, dy = untag(y, ctx), tangent(y, ctx)
   return tag(vx * vy, ctx, vy * dx + vx * dy)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(*), x::Tagged{T,<:Real}, y::Real) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   return tag(vx * y, ctx, y * dx)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(*), x::Real, y::Tagged{T,<:Real}) where {T}
   vy, dy = untag(y, ctx), tangent(y, ctx)
   return tag(x * vy, ctx, x * dy)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(+), x::Tagged{T,<:Real}, y::Tagged{T,<:Real}) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   vy, dy = untag(y, ctx), tangent(y, ctx)
   return tag(vx + vy, ctx, dx + dy)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(+), x::Tagged{T,<:Real}, y::Real) where {T}
   vx, dx = untag(x, ctx), tangent(x, ctx)
   return tag(vx + y, ctx, dx)
end

function execute(ctx::DiffCtxWithTag{T}, ::typeof(+), x::Real, y::Tagged{T,<:Real}) where {T}
   vy, dy = untag(y, ctx), tangent(y, ctx)
   return tag(x + vy, ctx, dy)
end
```

```
julia> D(sin, 1)
0.5403023058681398

julia> D(x -> sin(x) * cos(x), 1)
-0.4161468365471423

julia> D(x -> x * D(y -> x * y, 3), 5) # no confusion!
10

julia> D(x -> x * foo_bar_identity(x), 1)
2.0

julia> x = rand()
0.9667041764115833

julia> D(x -> CrazyModule.crazy_sum_mul([x, 2], [3, x]), x)
6.933408352823166

julia> 2x + 5
6.933408352823166
```

CrazyModule

```
const CONST_BINDING = Float64[]

global GLOBAL_BINDING = 0.0

struct Foo
    vector::Vector{Float64}
end

mutable struct FooContainer
    foo::Foo
end

mutable struct PlusFunc
    x::Float64
end

(f::PlusFunc)(x) = f.x + x

const PLUSFUNC = PlusFunc(0.0)
```

```
# implements a very convoluted `sum(x) * sum(y)`
function crazy_sum_mul(x::Vector{Float64}, y::Vector{Float64})
    @assert length(x) === length(y)
    fooc = FooContainer(Foo(x))
    tmp = y
    for i in 1:length(y)
        if iseven(i) # `fooc.foo.vector === y && tmp === x`
            v = fooc.foo.vector[i]
            push!(CONST_BINDING, tmp[i])
            global GLOBAL_BINDING = PLUSFUNC(v)
            PLUSFUNC.x = GLOBAL_BINDING
            fooc.foo = Foo(x)
            tmp = y
        else # `fooc.foo.vector === x && tmp === y`
            v = fooc.foo.vector[i]
            push!(CONST_BINDING, v)
            global GLOBAL_BINDING = PLUSFUNC(tmp[i])
            PLUSFUNC.x = GLOBAL_BINDING
            fooc.foo = Foo(y)
            tmp = x
        end
    end
    z = sum(CONST_BINDING) * GLOBAL_BINDING
    empty!(CONST_BINDING)
    PLUSFUNC.x = 0.0
    global GLOBAL_BINDING = 0.0
    return z
end
```

Thanks to EVERYBODY

- Prof. Juan Pablo Vielma and the MIT ORC
- Peter Ahrens and Valentin Churavy (labmates)
- Jeff Bezanson, Keno Fischer, and **Jameson Nash** (Julia Computing compiler team)
- Early Adopters: Tim Besard, Mike Innes, Curtis Vogt, Keno Fischer, James Bradbury, and more
- All you beautiful folks and the (even more!) beautiful organizers