Capstan

NEXT-GEN DIFFERENTIATION FOR JULIA

Jarrett Revels, MIT

First Things First

- 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.

- Currently working on extending the Julia compiler to support new AD

- Let’s talk about AD in Julia, where it’s at, where I’d like it to go, and how to get there.

First Things First

- 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.

- Currently working on extending the Julia compiler to support new AD

- Let’s talk about AD in Julia: where it’s at, where I’d like it to go, and how to get there.

- Caveat: I’m going to assume some knowledge of Julia and of AD.

The Present Landscape

- Multiple dispatch + method invocation JIT + metaprogramming = very decent language for AD

- **ForwardDiff** : native OO FM w/ stack-allocated perturbations

- **ReverseDiff** : native OO RM w/ dynamic taping + static compilation to Julia source, array primitives, and mixed-mode operation

- **JuMP** : modeling language w/ a static scalar RM interpreter. Supports hessian sparsity exploitation, and some mixed-mode operation + native Julia injection via ForwardDiff

- **ReverseDiffTape** : experimental edge-pushing RM for JuMP

The Present Landscape (cont'd)

- **Flux** : Functional ML framework with built-in OO RM; heavily PyTorch inspired

- **XGrad** : native source-to-source RM

- **Nabla** : native OO RM, many linear algebraic kernels

- **AutoGrad**: OO RM used by the Knet ML framework, port of Python `autograd` package

The Present Landscape (cont’d)

- **Flux** : Functional ML framework with built-in OO RM; heavily PyTorch inspired

- **XGrad** : native source-to-source RM

- **Nabla** : native OO RM, many linear algebraic kernels

- **AutoGrad**: OO RM used by the Knet ML framework, port of Python `autograd` package

- Okay, but what are we still missing?

Things That Make Me Sad

- feature set + “actual” target language of each AD tool is non-obvious

- multiple dispatch is great for exploiting runtime types, but quickly leads to method ambiguities for OO-based approaches

- few tools officially support generic mode nesting, and only with each other (ForwardDiff + ReverseDiff, JuMP + ForwardDiff)

- only one framework supports higher-order sparsity exploitation (JuMP + ReverseDiffTape)

- perturbation/sensitivity confusion runs rampant (ForwardDiff has compile-time tagging, but still not completely safe)

More Things That Make Me Sad

- Little-to-no work on differentiation for nonsmooth problems

- Little-to-no official complex number support

- Many linear algebraic derivative kernels are not implemented to be AD’able themselves

- Only in early stages of work for memory optimization, most of which is variable buffer pre-allocation (ReverseDiff, XGrad)

- Mostly slow dynamic scalar AD support (exception: ForwardDiff)

- Tools that are fast on the CPU are slow on the GPU and vise versa

Goals for Capstan

- no user-visible cumbersome custom array/number types
- works even with concrete dispatch/structural type constraints
- official support for complex differentiation
- safe nested/higher-order differentiation
- API for custom perturbation/sensitivity seeding
- user-extensible scalar and tensor derivative definitions
- tunable dynamism/mode for computation subgraphs
- live typed variable caching optimizations
- mixed-mode fused broadcast optimizations
- GPU support
- higher-order sparsity exploitation (edge-pushing)

Goals for Capstan

- no user-visible cumbersome custom array/number types
- works even with concrete dispatch/structural type constraints
- official support for complex differentiation
- safe nested/higher-
- API for custom pert
- user-extensible scalar and tensor derivative definitions
- tunable dynamism/mode for computation subgraphs
- live typed variable caching optimizations

Initial Release

- mixed-mode fused
- GPU support
- higher-order sparsit

Future Work

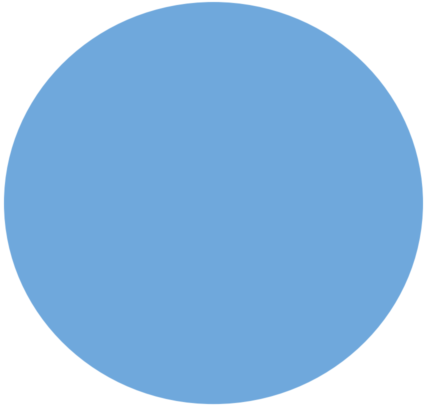
Great!

...But How?

CASSETTE

many julia packages try to fit a

peg into a

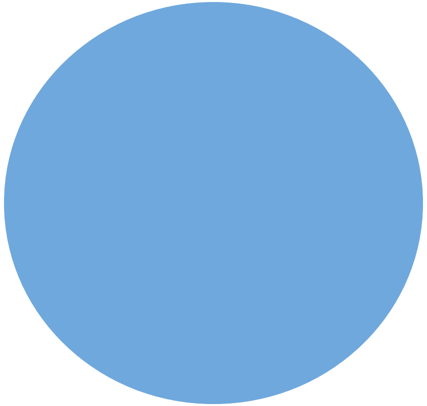


hole

many julia packages try to fit a

method overloading

peg into a



hole

many julia packages try to fit a

method overloading

peg into a

nonstandard interpretation

hole

method overloading

```julia
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

- Has many of the problems we discussed at the beginning of the talk

- 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)

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.

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, 0.0043752081491697 46)
add_float(-5.551115123125783e-17, 0.004375208149169746)
+(0.8370957766587268, 0.004375208149169691)
add_float(0.8370957766587268, 0.004375208149169691)
0.8414709848078965
```

Example: Counting Calls

```julia
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
```

```julia
# 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

```julia
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)
```

```julia
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

```julia
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, sigtype, codeinfo) -> begin
   # applies the first function to any piece of the
   # IR for which the second function returns `true`
   Cassette.replace_match!(to32bit, fitsin32bit, codeinfo.code)
   return codeinfo
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> 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.canrecurse(__context__, f, args...)
               newctx = Cassette.similarcontext(__context__, metadata = subtrace)
               return Cassette.recurse(newctx, f, args...)
           else
               return 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
```

A Mental Model For Overdubbing

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

$$\downarrow$$

```
overdub(Ctx(), () -> f(x))
```

A Mental Model For Overdubbing

```
function overdub(ctx, args...)
    prehook(ctx, args...)
    if isprimitive(ctx, args...)
        output = execute(ctx, args...)
    else
        output = recurse(ctx, args...)
    end
    posthook(ctx, output, args...)
    return output
end
```

A Mental Model For Overdubbing

```
@generated function recurse(ctx::ContextwithPass{pass}, args...) where {pass}
   lowered_ir = reflect(args)
   lowered_ir = pass(ctx, lowered_ir)
   return recurse_pass(lowered_ir)
end
```

```
quote # body of 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
```

recurse_pass

```
quote
   x = args[1]
   T = overdub(ctx, eltype, x)
   n = overdub(ctx, length, x)
   result = overdub(ctx, zero, T)
   oneT = overdub(ctx, one, T)
   k = overdub(ctx, *, 100, oneT)
   for i in overdub(ctx, :, 1, n)
       tmp1 = overdub(ctx, -, oneT, i)
       tmp2 = overdub(ctx, *, k, tmp1)
       result = overdub(ctx, +, result, tmp2)
   end
   return result
end
```

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

```julia
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

```julia
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
julia> using Cassette

julia> using Cassette: @context, withtagfor, overdub, tag, untag, metadata

julia> @context FooBarCtx

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

julia> ctx = withtagfor(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

```julia
import Cassette: @context, @primitive, Tagged, tag, untag, withtagfor, 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 = withtagfor(DiffCtx(), f)
   result = overdub(ctx, f, tag(x, ctx, oftype(x, 1.0)))
   return tangent(result)
end
```

Example: Forward-Mode AD

```
@primitive function Base.sin(x::Tagged{T,<:Real}) where {T,__CONTEXT__<:DiffCtxWithTag{T}}
    vx, dx = untag(x, __context__), tangent(x, __context__)
    return tag(sin(vx), __context__, cos(vx) * dx)
end

@primitive function Base.cos(x::Tagged{T,<:Real}) where {T,__CONTEXT__<:DiffCtxWithTag{T}}
    vx, dx = untag(x, __context__), tangent(x, __context__)
    return tag(cos(vx), __context__, -sin(vx) * dx)
end

@primitive function Base.:*(x::Tagged{T,<:Real}, y::Tagged{T,<:Real}) where {T,__CONTEXT__<:DiffCtxWithTag{T}}
    vx, dx = untag(x, __context__), tangent(x, __context__)
    vy, dy = untag(y, __context__), tangent(y, __context__)
    return tag(vx * vy, __context__, vy * dx + vx * dy)
end

@primitive function Base.:*(x::Tagged{T,<:Real}, y::Real) where {T,__CONTEXT__<:DiffCtxWithTag{T}}
    vx, dx = untag(x, __context__), tangent(x, __context__)
    return tag(vx * y, __context__, y * dx)
end

@primitive function Base.:*(x::Real, y::Tagged{T,<:Real}) where {T,__CONTEXT__<:DiffCtxWithTag{T}}
    vy, dy = untag(y, __context__), tangent(y, __context__)
    return tag(x * vy, __context__, x * dy)
end
```

Example: Forward Mode AD

```
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
```

Looking Forward

- If you want to how all of that really works, come talk to me

- Capstan isn’t just an AD package, it’s a proof-of-concept for the new techniques enabled via Cassette

- Cassette release - fall of this year?

- Capstan release - early 2019?

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)
- Tim Besard (GPU expert and Cassette bug-hunter)
- All you beautiful folks and the (even more!) beautiful organizers