`Mooncake.jl`

Will Tebbutt

2025-10-03

Introduction

- Algorithmic Differentiation in Julia
- Who: Me (ATI)
- Who: Hong Ge (University of Cambridge, ATI)
- Who: Guillaume Dalle, Xianda Sun, and many others!

Introduction

```julia
import DifferentiationInterface as DI
import Mooncake

backend = DI.AutoMooncake()
p = DI.prepare_gradient(f, backend, x)
DI.value_and_gradient(f, p, backend, x)
```

Introduction

```julia
function f(x::AbstractArray{<:Real})
    s = zero(eltype(x))
    for n in eachindex(x)
        s += cos(exp(x[n]))
    end
    return s
end

x = randn(1024)
```

Tool	Ratio	Language
`Enzyme.jl`	2.89	LLVM
`Mooncake.jl`	2.15	Julia
`ReverseDiff.jl`	7.41	Julia
`Zygote.jl`	427	Julia

Your numbers *will* differ.

Introduction

```julia
function f(x::AbstractArray{<:Real})
    s = zero(eltype(x))
    for n in eachindex(x)
        s += 5 * x[n]
    end
    return s
end

x = randn(1024)
```

Tool	Ratio	Language
`Enzyme.jl`	6.36	LLVM
`Mooncake.jl`	9.99	Julia
`ReverseDiff.jl`	73.0	Julia
`Zygote.jl`	5370	Julia

Your numbers *will* differ.

Why Bother?

Questions

Could we improve on `ReverseDiff` / `Zygote` while staying within the language?

- Testable?
- Composition "guarantees"?
- Good primal performance $\implies$ good AD performance?
- Wider language support inc. mutation?

AD Recap

What?

- $f : \mathbb{R}^P \to \mathbb{R}^Q$, differentiable at x
- $\mathbf{J} \in \mathbb{R}^{Q \times P}$, $\mathbf{J}_{q,p} := \partial f_q / \partial x_p$
- Forward Mode: $\dot{x} \mapsto \mathbf{J}\dot{x}$, $\dot{x} \in \mathbb{R}^P$
- Reverse Mode: $\bar{y} \mapsto \mathbf{J}^\top \bar{y}$, $\bar{y} \in \mathbb{R}^Q$

How?

- Repeated application of the chain rule
- $f := f_2 \circ f_1 \implies \mathbf{J} = \mathbf{J}_2\mathbf{J}_1$
- Forward Mode: $(\dot{x}_2 \mapsto \mathbf{J}_2\dot{x}_2) \circ (\dot{x}_1 \mapsto \mathbf{J}_1\dot{x}_1)$
- Reverse Mode (Forward-Pass): construct + store $\bar{y}_n \mapsto \mathbf{J}^\top\bar{y}_n$
- Reverse Mode (Reverse-Pass): load and apply: $(\bar{y}_2 \mapsto \mathbf{J}_2\bar{y}_2) \circ (\bar{y}_1 \mapsto \mathbf{J}_1\bar{y}_1)$

Testability and Tangents

Testability and Tangents

- $f : \mathbb{R}^P \to \mathbb{R}^Q$, differentiable
- $\mathbf{J}$: Jacobian of f at some x
- $\langle \bar{y}, \mathbf{J}\dot{x} \rangle = \langle \mathbf{J}^\top \bar{y}, \dot{x} \rangle$, $\dot{x} \in \mathbb{R}^P$, $\bar{y} \in \mathbb{R}^Q$
- `x isa P`, then `dx isa tangent_type(P)`
- Ensure $\langle \cdot, \cdot \rangle$ works for `tangent_type(P)` for all `P`
- Define `function`: `test_rule(rng, f, x...)`

Composition and Tangents

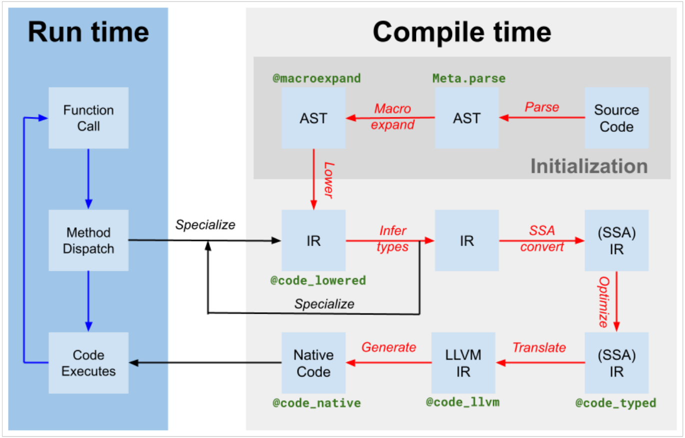
$$f = f_2 \circ f_1 \implies \mathbf{J}\dot{x} = \mathbf{J_2}(\mathbf{J_1}\dot{x})$$

```
function f(x)
    t = f_1(x)
    return f_2(t)
end
function rule(::typeof(f), (x, dx))
    (t, dt) = rule(f_1, (x, dx))
    return rule(f_2, (t, dt))
end
```

- Not formalised in Zygote
- Doable via `typeof(dt) = tangent_type(typeof(t))`

Performance

Transform Julia IR



https://docs.julialang.org/en/v1/devdocs/jit/

Transform Julia IR

```julia
f(x) = sin(cos(x))
Base.code_ircode_by_type(
    Tuple{typeof(f),Float64}
)[1][1]
```

```
julia> Base.code_ircode_by_type(Tuple{typeof(f),Float64})[1][1]
1 1 ─ %1 = invoke Main.cos(_2::Float64)::Float64
  │   %2 = invoke Main.sin(%1::Float64)::Float64
  └──        return %2
```

- Data must be stored on the forward-pass of reverse mode, and recovered on the reverse pass
- Zygote does not have type information
- `Core.OpaqueClosure` unavailable to Zygote

Mutation

```julia
function f(x::Vector{Float64})
    x .= 5 * x
    return sin(x[1])
end
```

- There *must* be a differentiable function associated to `f`
- What is it?
- Central trick: model state transition[1]

$$\Phi(x) = (5x, \sin(5x_1))$$

- Undo changes to state on the reverse-pass

[1] Pearlmutter and Siskind 2008.

Supported Language Features

Supported Language Features

```
f(x::AbstractVector{<:Real}) = sum(5x)
```

`Mooncake` ✓, `Enzyme`, ✓, `ReverseDiff` ✓, `Zygote` ✓

Supported Language Features

```julia
function f(x::AbstractVector{<:Real})
    for n in eachindex(x)
        x[n] = exp(x)
    end
    return x
end
```

Mooncake ✓, Enzyme ✓, ReverseDiff ✓, Zygote ✗

Supported Language Features

```julia
function f(x::Vector{Float64})
    for n in eachindex(x)
        x[n] = exp(x)
    end
    return x
end
```

`Mooncake` ✓, `Enzyme` ✓, `ReverseDiff` ✗, `Zygote` ✗

Supported Language Features

- Complicated Types ✓
- Incomplete Initialisation ✓
- Self-referential data structures ✓
- Dynamic dispatch ✓
- Value-dependent control flow ✓
- BLAS (mostly) / LAPACK (somewhat) ✓
- Structured matrices (`Diagonal`, `Symmetric`...) ✓
- Callable `structs` / closures ✓

Limitations

Basic Limitations
Julia

- Exception handling (`UpsilonNodes` and `PhiCNodes`)
- `foreigncalls` always require rules
- Higher memory usage (in general)
- Self-referential types

```
struct F
    x::Union{Nothing,F}
end
```

Current Limitations: Cheap Scalar Operations

- Vectorisation (SIMD) obstructed
- Sub-optimal storage strategies (loop invariants, induction variables, constants, etc)
- See issue 156 for thoughts.

Conclusion

- Challenges remain
- Some important performance and language feature support limitations of `RD` / `Zygote` addressed my `Mooncake`
- Written in, and operates on, Julia