julia for automatic differentiation

Jarrett Revels (MIT), Miles Lubin (MIT), Theodore Papamarkou (University of Glasgow)

Numerical Derivatives 101

Let's say I give you an implementation of a function $f : \mathbb{R} \to \mathbb{R}$ whose code you **didn't write** and **can't read**. How do you calculate $f'(x)$?

Finite Difference Method

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \boxed{\frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)}$$

Advantages

- works "out of the box"

Disadvantages

- $\mathcal{O}(h^2)$ truncation error
- h too small $\rightarrow$ subtractive cancellation error
- requires 2 evaluations of f

Complex Step Method

$$f(x+ih) = f(x)+f'(x)hi+\frac{f''(x)}{2!}h^2i^2 \ldots = f(x) + f'(x)hi - \frac{f''(x)}{2!}h^2 \ldots$$

$$\boxed{f'(x) = \frac{\mathrm{Im}[f(x+hi)]}{h} + \mathcal{O}(h^2)}$$

Advantages

- no subtractive cancellation error
- Doesn't need 2 calls to f
- A lot of languages offer efficient complex number implementations
- truncation error can be close to machine epsilon in practice

Disadvantages

- complex number operations can be much slower than real number operations
- unsafe for programs which already incorporate complex inputs
- requires operator overloading and/or source code transformation

Dual Number Method

$$f(x+y\epsilon) = f(x)+f'(x)y\epsilon+\frac{f''(x)}{2!}y^2\epsilon^2 \dots \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$= f(x) + f'(x)y\epsilon$$

$$\boxed{f'(x) = \mathrm{Eps}[f(x+\epsilon)]}$$

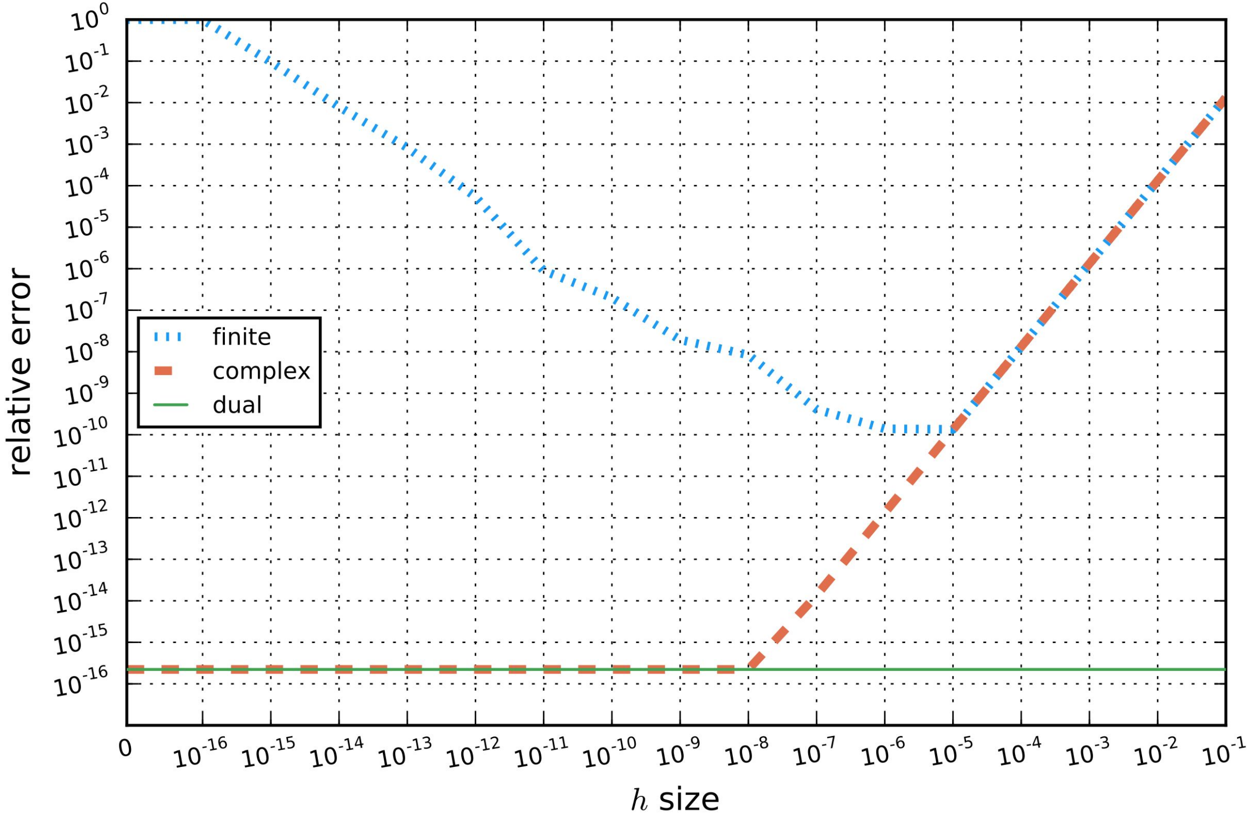
Advantages

- no approximation error

Disadvantages

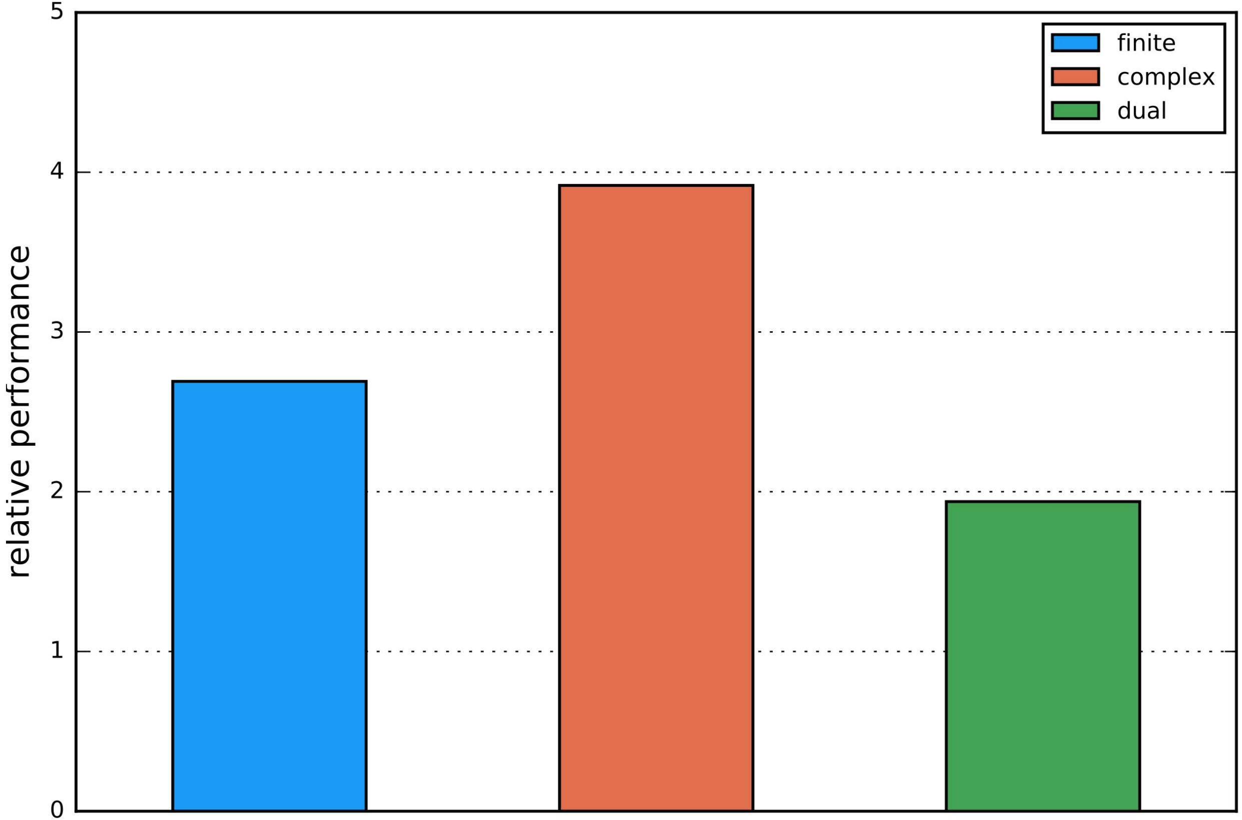
- target program must accept generic number types
- requires operator overloading and/or source code transformation

Error Comparison



Performance Comparison

$$f(x) = \frac{e^x}{\sqrt{\sin^3(x) + \cos^3(x)}}$$



performance is relative to $f(x)$

Going to Higher Dimensions

First-order derivatives of scalar functions are **boring**. What about functions like $g : \mathbb{R}^n \to \mathbb{R}$ or $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$? What about **gradients** and **Jacobians**?

Gradients

$$\boxed{\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}}$$

Gradients

$$\boxed{\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}}$$

$$\boxed{f(x + \epsilon) = f(x) + f'(x)\epsilon}$$

Gradients

$$\boxed{\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}}$$

$$\boxed{f(x+\epsilon) = f(x) + f'(x)\epsilon}$$

Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \boxed{\frac{\partial g(\mathbf{x})}{\partial x_i}} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$

$$g\left(\begin{bmatrix} x_1 \\ \vdots \\ x_i + \epsilon \\ \vdots \\ x_n \end{bmatrix}\right) = g(\mathbf{x}) + \boxed{\frac{\partial g(\mathbf{x})}{\partial x_i}}\epsilon$$

Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \boxed{\frac{\partial g(\mathbf{x})}{\partial x_i}} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$

But do we really need n calls to g?

$$g\left(\begin{bmatrix} x_1 \\ \vdots \\ x_i + \epsilon \\ \vdots \\ x_n \end{bmatrix}\right) = g(\mathbf{x}) + \boxed{\frac{\partial g(\mathbf{x})}{\partial x_i}}\epsilon$$

Extending Dual Numbers

$$\boxed{f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0}$$

Extending Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x+\sum_{i=1}^{n} y_i\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Extending Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x+\sum_{i=1}^{n} y_i\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \rightarrow \mathbf{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}$$

Extending Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x + \sum_{i=1}^{n} y_i\epsilon_i) = f(x) + f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \rightarrow \mathbf{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}$$

$$g(\mathbf{x}_\epsilon) = g(\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial g(\mathbf{x})}{\partial x_i}\epsilon_i$$

Extending Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x + \sum_{i=1}^{n} y_i\epsilon_i) = f(x) + f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \rightarrow \mathbf{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}$$

$$g(\mathbf{x}_\epsilon) = g(\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial g(\mathbf{x})}{\partial x_i}\epsilon_i$$

Extending Dual Numbers

Naive implementation: Heap-allocate an array for y_i values

$$f\left(x+\sum_{i=1}^{n}\boxed{y_i}\epsilon_i\right) = f(x)+f'(x)\sum_{i=1}^{n}y_i\epsilon_i \quad \text{where} \quad \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Extending Dual Numbers

Naive implementation: Heap-allocate an array for y_i values

$$f\left(x+\sum_{i=1}^{n}\boxed{y_i}\epsilon_i\right) = f(x)+f'(x)\sum_{i=1}^{n}y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Problem: y_i loads/stores will be slower than x value load/stores

Extending Dual Numbers

Naive implementation: Heap-allocate an array for y_i values

$$f(x+\sum_{i=1}^{n}\boxed{y_i}\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n}y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Problem: y_i loads/stores will be slower than x value load/stores

Problem: Heap allocation for every scalar will incur GC wrath

Extending Dual Numbers

Naive implementation: Heap-allocate an array for y_i values

$$f(x+\sum_{i=1}^{n}\boxed{y_i}\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n}y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Problem: y_i loads/stores will be slower than x value load/stores

Problem: Heap allocation for every scalar will incur GC wrath

Solution: Allocate y_i values on the stack

The `Dual` Type

```
# stack-allocated vector of partial derivatives
using ForwardDiff.Partials


# N-dimensional dual number type
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end
```

The `Dual` Type

```julia
# stack-allocated vector of partial derivatives
using ForwardDiff.Partials


# N-dimensional dual number type
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end


# overload various math operations
import Base: sin, cos, -, +, *

sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
cos(d::Dual) = Dual(cos(d.value), -(sin(d.value)) * d.partials)
(-)(d::Dual) = Dual(-(d.value), -(d.partials))
(+)(a::Dual, b::Dual) = Dual(a.value + b.value, a.partials + b.partials)
(*)(a::Dual, b::Dual) = Dual(a.value * b.value,
                             b.value * a.partials + a.value * b.partials)
```

The `Dual` Type

```
# stack-allocated vector of partial derivatives
using ForwardDiff.Partials


# N-dimensional dual number type
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end


# overload various math operations
import Base: sin, cos, -, +, *

sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
cos(d::Dual) = Dual(cos(d.value), -(sin(d.value)) * d.partials)
(-)(d::Dual) = Dual(-(d.value), -(d.partials))
(+)(a::Dual, b::Dual) = Dual(a.value + b.value, a.partials + b.partials)
(*)(a::Dual, b::Dual) = Dual(a.value * b.value,
                             b.value * a.partials + a.value * b.partials)
```

This code enables:

- `sin` and `cos` derivatives to **arbitrary order** (e.g. `Dual{M,Dual{N,T}}`)
- `sin` and `cos` derivatives over **complex number types** (e.g. `Complex{Dual{N,T}}`)
- `sin` and `cos` derivatives over **custom number types** (e.g. `Custom{Dual{N,T}}`)

Taking Jacobians with `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \cdots x_1 \end{bmatrix}$$

```
function cumprod(x)
    y = similar(x)
    if length(x) < 1
        return y
    end
    y[1] = x[1]
    for i in 2:length(y)
        y[i] = y[i-1]*x[i]
    end
    return y
end
```

Taking Jacobians with `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \frac{\partial g_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_j(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \frac{\partial g_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_j(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$\mathbf{g}(\mathbf{x}_\epsilon) = \begin{bmatrix} g_1(\mathbf{x}_\epsilon) \\ \vdots \\ g_j(\mathbf{x}_\epsilon) \\ \vdots \\ g_m(\mathbf{x}_\epsilon) \end{bmatrix} = \begin{bmatrix} g_1(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_1(\mathbf{x})}{\partial x_i}\epsilon_i \\ \vdots \\ g_j(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_j(\mathbf{x})}{\partial x_i}\epsilon_i \\ \vdots \\ g_m(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_m(\mathbf{x})}{\partial x_i}\epsilon_i \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \boxed{\frac{\partial g_1(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_1(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_1(\mathbf{x})}{\partial x_n}} \\ \vdots \quad \ddots \quad \vdots \quad \ddots \\ \boxed{\frac{\partial g_j(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_j(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_j(\mathbf{x})}{\partial x_n}} \\ \vdots \quad \ddots \quad \vdots \quad \ddots \\ \boxed{\frac{\partial g_m(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_m(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_m(\mathbf{x})}{\partial x_n}} \end{bmatrix}$$

$$\mathbf{g}(\mathbf{x}_\epsilon) = \begin{bmatrix} g_1(\mathbf{x}_\epsilon) \\ \vdots \\ g_j(\mathbf{x}_\epsilon) \\ \vdots \\ g_m(\mathbf{x}_\epsilon) \end{bmatrix} = \begin{bmatrix} g_1(\mathbf{x}) + \sum_{i=1}^{n} \boxed{\frac{\partial g_1(\mathbf{x})}{\partial x_i}} \epsilon_i \\ \vdots \\ g_j(\mathbf{x}) + \sum_{i=1}^{n} \boxed{\frac{\partial g_j(\mathbf{x})}{\partial x_i}} \epsilon_i \\ \vdots \\ g_m(\mathbf{x}) + \sum_{i=1}^{n} \boxed{\frac{\partial g_m(\mathbf{x})}{\partial x_i}} \epsilon_i \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\mathrm{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\text{cumprod}(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

Taking Jacobians with `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

$$\downarrow$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\downarrow$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

```julia
julia> using ForwardDiff.Dual

julia> x = [Dual(1, 1, 0, 0),   # 1 + ϵ₁
            Dual(2, 0, 1, 0),   # 2 + ϵ₂
            Dual(3, 0, 0, 1)];  # 3 + ϵ₃
```

Taking Jacobians with `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \boxed{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}}$$

```
julia> using ForwardDiff.Dual

julia> x = [Dual(1, 1, 0, 0),  # 1 + ϵ₁
            Dual(2, 0, 1, 0),  # 2 + ϵ₂
            Dual(3, 0, 0, 1)]; # 3 + ϵ₃

julia> cumprod(x)
3-element Array{ForwardDiff.Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)
```

Taking Jacobians with `Duals`

$$\text{cumprod}(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}) = \boxed{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}}$$

```
julia> using ForwardDiff.Dual

julia> x = [Dual(1, 1, 0, 0),  # 1 + ϵ₁
            Dual(2, 0, 1, 0),  # 2 + ϵ₂
            Dual(3, 0, 0, 1)]; # 3 + ϵ₃

julia> cumprod(x)
3-element Array{ForwardDiff.Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)

julia> ForwardDiff.jacobian(cumprod, [1,2,3])
3x3 Array{Int64,2}:
 1   0   0
 2   1   0
 6   3   2
```

Taking Jacobians with `Duals`

$$\mathrm{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\mathrm{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\mathrm{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

```
julia> using ForwardDiff.Dual

julia> x = [Dual(1, 1, 0, 0),   # 1 + ϵ₁
            Dual(2, 0, 1, 0),   # 2 + ϵ₂
            Dual(3, 0, 0, 1)]; # 3 + ϵ₃

julia> cumprod(x)
3-element Array{ForwardDiff.Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)

julia> ForwardDiff.jacobian(cumprod,
                            [1,2,3],
                            Chunk{3}())

3x3 Array{Int64,2}:
 1  0  0
 2  1  0
 6  3  2
```

Let’s talk about performance

Is `Dual` efficient?

```
julia> @code_native 1 * 1
Filename: int.jl
          pushq     %rbp
          movq      %rsp, %rbp
          imulq     %rsi, %rdi
          movq      %rdi, %rax
          popq      %rbp
          retq
          nopl      (%rax)
```

vs.

```
julia> @code_native Dual(1) * Dual(1)
Filename: dual.jl
          pushq     %rbp
          movq      %rsp, %rbp
          movq      (%rsi), %rax
          imulq     (%rdi), %rax
          popq      %rbp
          retq
          nopl      (%rax)
```

Is `Dual` efficient?

```
julia> @code_native 1 * 1
Filename: int.jl
          pushq     %rbp
          movq      %rsp, %rbp
          imulq     %rsi, %rdi
          movq      %rdi, %rax
          popq      %rbp
          retq
          nopl      (%rax)
```

vs.

```
julia> @code_native Dual(1) * Dual(1)
Filename: dual.jl
          pushq     %rbp
          movq      %rsp, %rbp
          movq      (%rsi), %rax
          imulq     (%rdi), %rax
          popq      %rbp
          retq
          nopl      (%rax)
```

Is `Dual` efficient?

```
julia> f(a, b) = (a[1]*b[1], a[1]*b[2] + b[1]*a[2])
f (generic function with 2 methods)

julia> @code_native f((1, 1), (1, 1))
Filename: REPL[17]
            pushq       %rbp
            movq        %rsp, %rbp
            movq        (%rsi), %rax
            movq        (%rdx), %rcx
            movq        %rcx, %r8
            imulq       %rax, %r8
            imulq       8(%rdx), %rax
            imulq       8(%rsi), %rcx
            addq        %rax, %rcx
            movq        %r8, (%rdi)
            movq        %rcx, 8(%rdi)
            movq        %rdi, %rax
            popq        %rbp
            retq
            nopw        (%rax,%rax)
```

vs.

```
julia> @code_native Dual(1, 1) * Dual(1, 1)
Filename: dual.jl
            pushq       %rbp
            movq        %rsp, %rbp
            movq        (%rsi), %rax
            movq        (%rdx), %rcx
            movq        8(%rsi), %rsi
            imulq       %rcx, %rsi
            imulq       %rax, %rcx
            imulq       8(%rdx), %rax
            addq        %rsi, %rax
            movq        %rcx, (%rdi)
            movq        %rax, 8(%rdi)
            movq        %rdi, %rax
            popq        %rbp
            retq
            nopw        (%rax,%rax)
```

Is `Dual` efficient?

```
julia> f(a, b) = (a[1]*b[1], a[1]*b[2] + b[1]*a[2])
f (generic function with 2 methods)

julia> @code_native f((1, 1), (1, 1))
Filename: REPL[17]
            pushq       %rbp
            movq        %rsp, %rbp
            movq        (%rsi), %rax
            movq        (%rdx), %rcx
            movq        %rcx, %r8
            imulq       %rax, %r8
            imulq       8(%rdx), %rax
            imulq       8(%rsi), %rcx
            addq        %rax, %rcx
            movq        %r8, (%rdi)
            movq        %rcx, 8(%rdi)
            movq        %rdi, %rax
            popq        %rbp
            retq
            nopw        (%rax,%rax)
```

vs.

```
julia> @code_native Dual(1, 1) * Dual(1, 1)
Filename: dual.jl
            pushq       %rbp
            movq        %rsp, %rbp
            movq        (%rsi), %rax
            movq        (%rdx), %rcx
            movq        8(%rsi), %rsi
            imulq       %rcx, %rsi
            imulq       %rax, %rcx
            imulq       8(%rdx), %rax
            addq        %rsi, %rax
            movq        %rcx, (%rdi)
            movq        %rax, 8(%rdi)
            movq        %rdi, %rax
            popq        %rbp
            retq
            nopw        (%rax,%rax)
```

Benchmarking vs. `autograd`

The Python package **autograd** is a popular **reverse-mode** automatic differentiation tool. Reverse-mode AD is algorithmically more efficient than forward-mode AD for taking gradients of functions $g : \mathbb{R}^n \to \mathbb{R}$.

`ForwardDiff` v.s. `autograd`

$$\text{Rosenbrock}(\vec{x}) = \sum_{i=1}^{k-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

Input Size	**`autograd`** Time (ms)	**`ForwardDiff`** Time (ms)	Ratio
10	0.762710 (105x)	0.000581 (32x)	1312.75
100	4.268964 (284x)	0.027806 (169x)	153.52
1000	39.14904 (424x)	2.653071 (1693x)	14.76
10000	398.4010 (467x)	263.6982 (16927x)	1.51
100000	4086.092 (473x)	26654.46 (171271x)	0.15

`ForwardDiff` v.s. `autograd`

$$\text{Ackley}(\vec{x}) = -a\exp\left(-b\sqrt{\tfrac{1}{k}\sum_{i=1}^{k} x_i^2}\right) - \exp\left(\tfrac{1}{k}\sum_{i=1}^{k}\cos(cx_i)\right) + a + \exp(1)$$

Input Size	**`autograd`** Time (ms)	**`ForwardDiff`** Time (ms)	Ratio
10	1.174616 (73x)	0.000940 (6x)	1249.59
100	8.257150 (257x)	0.053958 (38x)	153.02
1000	79.81085 (419x)	5.480441 (318x)	14.56
10000	809.0989 (452x)	561.3962 (2716x)	1.44
100000	8209.631 (463x)	56838.42 (27308x)	0.14

`ForwardDiff` Downstream

- ~40 unique projects on GitHub depend on **`ForwardDiff`**
- **`Celeste.jl`**, **`SloanDigitalSkySurvey.jl`** (Astronomy)
- **`RigidBodyDynamics.jl`** (Robotics)
- **`Klara.jl`**, **`VinDsl.jl`**, **`MADS.jl`** (Statistics)
- **`ValidatedNumerics.jl`** (Interval Arithmetic)
- **`JuliaFEM.jl`**, **`SurfaceGeometry.jl`** (Finite Element Analysis)
- **`NLSolve.jl`**, **`Roots.jl`**, **`DifferentialEquations.jl`** (Equation Solving)
- **`Optim.jl`**, **`JuMP.jl`** (Optimization)

`ForwardDiff` Downstream

- ~40 unique projects on GitHub depend on **`ForwardDiff`**
- **`Celeste.jl`**, **`SloanDigitalSkySurvey.jl`** (Astronomy)
- **`RigidBodyDynamics.jl`** (Robotics)
- **`Klara.jl`**, **`VinDsl.jl`**, **`MADS.jl`** (Statistics)
- **`ValidatedNumerics.jl`** (Interval Arithmetic)
- **`JuliaFEM.jl`**, **`SurfaceGeometry.jl`** (Finite Element Analysis)
- **`NLSolve.jl`**, **`Roots.jl`**, **`DifferentialEquations.jl`** (Equation Solving)
- **`Optim.jl`**, **`JuMP.jl`** (Optimization)

We’re also working on reverse-mode AD!

We’re also working on reverse-mode AD!

…but…what *is* reverse-mode AD?

$f(a,b) = ?$

$\partial f(a,b)/\partial a = ?$

$\partial f(a,b)/\partial b = ?$

$f(a,b) = ?$

$\partial f(a,b)/\partial a = ?$

$\partial f(a,b)/\partial b = ?$

$x_1 = \sin(a)$

f(a,b) = ?

∂f(a,b)/∂a = ?

∂f(a,b)/∂b = ?

x_1 = sin(a)

x_2 = cos(b)

$f(a,b) = ?$

$\partial f(a,b)/\partial a = ?$

$\partial f(a,b)/\partial b = ?$

$x_1 = \sin(a)$

$x_2 = \cos(b)$

$x_3 = x_1 * x_2$

$$f(a,b) = x_3$$
$$= x_1 * x_2$$
$$= \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$x_2 = \cos(b)$$

$$x_3 = x_1 * x_2$$

$$f(a,b) = x_3 = x_1 * x_2 = \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$x_2 = \cos(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3$$
$$= x_1 * x_2$$
$$= \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = \partial x_3/\partial x_2 * \partial x_2/\partial b$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3 = x_1 * x_2 = \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3$$
$$= x_1 * x_2$$
$$= \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$\partial x_3/\partial a = \partial x_3/\partial x_1 * \partial x_1/\partial b$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$
$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3 = x_1 * x_2 = \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$\partial x_3/\partial a = x_2 * \cos(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3 = x_1 * x_2 = \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a = x_2 * \cos(a) = \cos(b) * \cos(a)$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b$$

$$x_1 = \sin(a)$$

$$\partial x_3/\partial a = x_2 * \cos(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$f(a,b) = x_3 = x_1 * x_2 = \sin(a) * \cos(b)$$

$$\partial f(a,b)/\partial a = \partial x_3/\partial a = x_2 * \cos(a) = \cos(b) * \cos(a)$$

$$\partial f(a,b)/\partial b = \partial x_3/\partial b = x_1 * -\sin(b) = \sin(a) * -\sin(b)$$

$$x_1 = \sin(a)$$

$$\partial x_3/\partial a = x_2 * \cos(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

$$
\begin{aligned}
f(a,b) &= x_3 \\
&= x_1 * x_2 \\
&= \sin(a) * \cos(b)
\end{aligned}
$$

$$
\begin{aligned}
\partial f(a,b)/\partial a &= \partial x_3/\partial a \\
&= x_2 * \cos(a) \\
&= \cos(b) * \cos(a)
\end{aligned}
$$

$$
\begin{aligned}
\partial f(a,b)/\partial b &= \partial x_3/\partial b \\
&= x_1 * -\sin(b) \\
&= \sin(a) * -\sin(b)
\end{aligned}
$$

$$x_1 = \sin(a)$$

$$\partial x_3/\partial a = x_2 * \cos(a)$$

$$x_2 = \cos(b)$$

$$\partial x_3/\partial b = x_1 * -\sin(b)$$

$$x_3 = x_1 * x_2$$

$$\partial x_3/\partial x_1 = x_2$$

$$\partial x_3/\partial x_2 = x_1$$

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

- Forward pass records program trace via operator overloading, backward pass is interpreted

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

- Forward pass records program trace via operator overloading, backward pass is interpreted

- Trace nodes can cache arbitrary storage for intermediary derivatives and work buffers. (Goal: non-allocating backward pass.)

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

- Forward pass records program trace via operator overloading, backward pass is interpreted

- Trace nodes can cache arbitrary storage for intermediary derivatives and work buffers. (Goal: non-allocating backward pass.)

- Supports multivariate optimizations (e.g. linear algebra functions) over generic array types

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

- Forward pass records program trace via operator overloading, backward pass is interpreted

- Trace nodes can cache arbitrary storage for intermediary derivatives and work buffers. (Goal: non-allocating backward pass.)

- Supports multivariate optimizations (e.g. linear algebra functions) over generic array types

- Call-site/definition-site user annotations for enabling optimizations

ReverseDiffPrototype.jl

- Reverse-mode AD written in native Julia, for native Julia

- Forward pass records program trace via operator overloading, backward pass is interpreted

- Trace nodes can cache arbitrary storage for intermediary derivatives and work buffers. (Goal: non-allocating backward pass.)

- Supports multivariate optimizations (e.g. linear algebra functions) over generic array types

- Call-site/definition-site user annotations for enabling optimizations

- “Orphan”/constant node elision

`ReverseDiffPrototype` v.s. `autograd`

$$\text{Rosenbrock}(\vec{x}) = \sum_{i=1}^{k-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

Input Size	**`autograd`** Time (ms)	**`ReverseDiffProtoype`** Time (ms)	Ratio
10	0.762710 (105x)	0.005820 (34x)	131.05
100	4.268964 (284x)	0.011051 (17x)	386.29
1000	39.14904 (424x)	0.065846 (13x)	594.55
10000	398.4010 (467x)	0.608596 (12x)	654.62
100000	4086.092 (473x)	6.582295 (13x)	620.77

Limitations of AD

Target Function Restrictions

1. **The function must only be composed of generic Julia code** (e.g. no LAPACK)

2. **The function must accept** `Real` **or** `AbstractArray{T<:Real}` **inputs**

3. **The function should be type-stable** (okay, not strictly, but it really should be)

Perturbation Confusion

```
D = (f, x_0) -> df/dx evaluated at x_0

# nested, closed over differentiation
D(x -> x * D(y -> x + y, 1), 1)

# correct answer
d1 = D(x -> x * D(y -> x + y, 1), 1)
d1 = D(x -> x * (y -> 1)(1), 1)
d1 = D(x -> x, 1)
d1 = (x -> 1)(1)
d1 = 1
```

Perturbation Confusion

```
D = (f, x_0) -> df/dx evaluated at x_0

# nested, closed over differentiation
D(x -> x * D(y -> x + y, 1), 1)

# correct answer
d1 = D(x -> x * D(y -> x + y, 1), 1)
d1 = D(x -> x * (y -> 1)(1), 1)
d1 = D(x -> x, 1)
d1 = (x -> 1)(1)
d1 = 1
```

```
const D = ForwardDiff.derivative

# nested, closed over differentiation
D(x -> x * D(y -> x + y, 1), 1)

# what ForwardDiff will compute
d2 = D(x -> x * D(y -> x + y, 1), 1)
d2 = D(x -> x * epsilon(x + (1 + ϵ)), 1)
d2 = epsilon((1 + ϵ) * epsilon((1 + ϵ) + (1 + ϵ)))
d2 = epsilon((1 + ϵ) * epsilon(2 + 2ϵ))
d2 = epsilon((1 + ϵ) * 2)
d2 = epsilon(2 + 2ϵ)
d2 = 2 # != d1
```

Future Work

- SIMD for `Partials` type
- Perturbation Confusion?
- Better parallel support
- Document and release `ReverseDiffPrototype` as `ReverseDiff`
- Subgraph compilation and reuse
- GPU support

Resources

- Presentation repository: https://github.com/jrevels/ForwardDiffPresentation
- ForwardDiff: https://github.com/JuliaDiff/ForwardDiff.jl
- ReverseDiffPrototype: https://github.com/jrevels/ReverseDiffPrototype.jl
- JuMP: https://github.com/JuliaOpt/JuMP.jl

Acknowledgements

- **The Julia Group:** Alan Edelman, Jiahao Chen, Andreas Noack, and more
- **Steven G. Johnson and US Army Research Office** (Contract No. W911NF-07-D0004)
- **Collaborators:** Miles Lubin, John Pearson
- **Contributors to ForwardDiff:** Kristoffer Carlsson, Tim Holy, and others
- **Workshop Organizers**