juliacon 2021

Everything you need to know about ChainRules 1.0

Miha Zgubič, Research Software Engineer @ INVENIA LABS

INVENIA 💙 julia

We use machine learning to optimise the electricity grids.

more Julia, less emissions

Come join us!

ChainRules has many


Alex Arslan


David Widmann


Jarrett Revels


Lyndon White

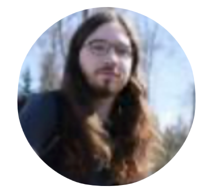

Michael Abbott


Miha Zgubic

Matt Brzezinski


Nick Robinson


Seth Axen


Simeon Schaub


Will Tebbutt


Yingbo Ma


...many wonderful contributors

Shashi Gowda
Pietro Vertechi
Kristoffer Carlsson
Andrew Fitzgibbon
Curtis Vogt
Jerry Ling
Mason Protter
Steven Johnson
Viral Shah
Keno Fischer
Rory Finnegan
Mike Innes
Glenn Moynihan

Dhairya Gandhi
Branwen Snelling
Gaurav Dhingra
Niklas Heim
Andrew Rosemberg
Niklas Schmitz
Chris Rackauckas
Andrei Zhabinski
Mathieu Besançon
Jeffrey Sarnoff
Anton Isopoussu
Antoine Levitt

Fernando Chorney
James Bradbury
Ben Cottier
Oliver Schulz
Alex Robson
cormullion
Roger Luo
Simon Etter
ho-oto
Wessel Bruinsma
Takafumi Arakaki
Carlo Lucibello
Eric Davies

Tour de ChainRules

Introduction

When to write rules

How to write rules

- Basic example
- Gotchas and helpers
- RuleConfig
- Array dilemma

How to test rules

Summary

Lyndon's talk last JuliaCon is a great introduction

But in a nutshell:

- Automatic differentiation (AD) is useful
- We have many AD systems in Julia
- They all need rules that are efficient and correct

ChainRules.jl

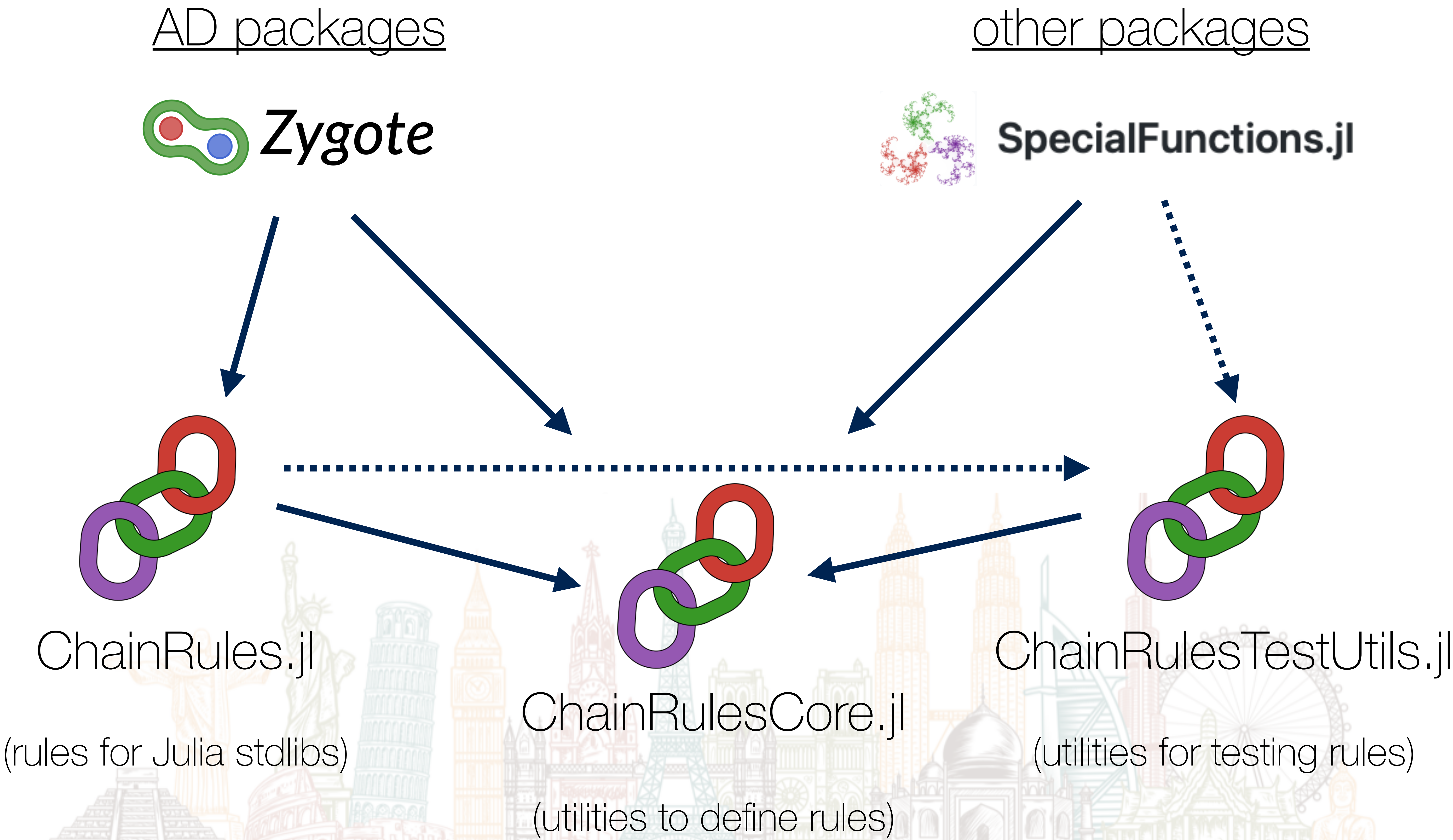
(rules for Julia stdlibs)

ChainRulesCore.jl

(utilities to define rules)

ChainRulesTestUtils.jl

(utilities for testing rules)


AD packages
Zygote
other packages
SpecialFunctions.jl
ChainRules.jl
(rules for Julia stdlibs)
ChainRulesCore.jl
(utilities to define rules)
ChainRulesTestUtils.jl
(utilities for testing rules)

Kinds of rules

`frule`

(Forwards mode AD)

```
function frule((ḟ, ȧrgs...), f, args...)
    y = f(args)
    ...
    return y, ẏ
end
```

$$\dot{x} = \frac{\partial x}{\partial (\text{something})}$$

`rrule`

(Backwards mode AD)

```
function rrule(f, args...)
    y = f(args...)
    function f_pullback(ȳ)
        ...
        return (f̄, ārgs...)
    end
    return y, f_pullback
end
```

$$\bar{x} = \frac{\partial (\text{something})}{\partial x}$$

Kinds of tangents

`ZeroTangent()`	can be perturbed, but no change in output
`NoTangent()`	can not be perturbed
`Tangent{Foo}()`	tangent of `struct`s, `Tuple`s, `NamedTuple`s, and `Dict`s
`Thunk()`	delayed computation
`InplaceableThunk()`	delayed computation, can accumulate gradients inplace

Tour de ChainRules

Introduction

When to write rules

How to write rules

Basic example

Gotchas and helpers

RuleConfig

Array dilemma

How to test rules

Summary

perfect AD system needs rules

to work at all: basic operations (+, *, …)

to be efficient: e.g. AD through numerical integration, AD through DiffOpt

imperfect AD systems need more rules

to work at all: unsupported features (e.g. mutation)

to be efficient: e.g. differentiating through loops

When to write rules for Zygote

To work around unsupported features

```
function mutation(n::Integer)
    array = zeros(n)
    array[1] = 1
    return sum(array)
end
```

```
function exception_handling(x)
    try
        return x^2
    catch e
        println("can't square $x")
        throw(e)
    end
end
```

To improve efficiency

```
function no_inplace_accum(array)
    x = array[1]
    y = array[2]
    z = array[3]
    return x+y+z
end
```

```
function for_loops(array)
    s = 0
    for a in array
        s += a
    end
    return s
end
```

When to write rules for Zygote

To work around unsupported features

```
function mutation(n::Integer)
    array = zeros(n)
    array[1] = 1
    return sum(array)
end
```

```
function exception_handling(x)
    try
        return x^2
    catch e
        println("can't square $x")
        throw(e)
    end
end
```

To improve efficiency

```
function no_inplace_accum(array)
    x = array[1]
    y = array[2]
    z = array[3]
    return x+y+z
end
```

```
function for_loops(array)
    s = 0
    for a in array
        s += a
    end
    return s
end
```

When to write rules for Zygote

To work around unsupported features

```julia
function mutation(n::Integer)
    array = zeros(n)
    array[1] = 1
    return sum(array)
end
```

```julia
function exception_handling(x)
    try
        return x^2
    catch e
        println("can't square $x")
        throw(e)
    end
end
```

To improve efficiency

```julia
function no_inplace_accum(array)
    x = array[1]
    y = array[2]
    z = array[3]
    return x+y+z
end
```

```julia
function for_loops(array)
    s = 0
    for a in array
        s += a
    end
    return s
end
```

Tour de ChainRules

Introduction

When to write rules

How to write rules

- Basic example
- Gotchas and helpers
- RuleConfig
- Array dilemma

How to test rules

Summary

Primal program

just a matrix multiplication
(note that Foo.c is not involved)

```
struct Foo
    A::Matrix
    c::Float64
end
```

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

Code to write the rrule

```julia
struct Foo
    A::Matrix
    c::Float64
end
```

```julia
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```julia
import ChainRulesCore: rrule
```

need to extend the function

Code to write the rrule

```
struct Foo
    A::Matrix
    c::Float64
end
```

Reminder of the rrule signature

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Code to write the rrule

```
struct Foo
    A::Matrix
    c::Float64
end
```

It is possible to change the primal computation
(might want for efficiency if work shared in primal and pullback)

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Code to write the rrule

```
struct Foo
    A::Matrix
    c::Float64
end
```

pullback signature:
primal output tangent -> primal input tangents

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Code to write the rrule

```
struct Foo
    A::Matrix
    c::Float64
end
```

`foo_mul` does not have fields, use `NoTangent()`

NB: functors (callable structs) can have fields

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Code to write the rrule

```
struct Foo
    A::Matrix
    c::Float64
end
```

tangent of `foo::Foo` is `Tangent{Foo}(; …)`
tangent of the `c` field is `ZeroTangent()`

```
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Non-specified `Tangent` fields are implicitly `ZeroTangent()`

Code to write the rrule

```julia
struct Foo
    A::Matrix
    c::Float64
end
```

use Thunks (to delay and potentially avoid computation)
or InplaceableThunks (to accumulate gradients inplace)
e.g. `B̄ = @thunk foo.A' * Ȳ`

```julia
function foo_mul(foo::Foo, B::AbstractArray)
    return foo.A * B
end
```

```julia
import ChainRulesCore: rrule

function rrule(::typeof(foo_mul), foo::Foo, B::AbstractArray)
    Y = foo_mul(foo, B)
    function foo_mul_pullback(Ȳ)
        f̄ = NoTangent()
        f̄oo = Tangent{Foo}(; A=Ȳ * B', c=ZeroTangent())
        B̄ = foo.A' * Ȳ
        return f̄, f̄oo, B̄
    end
    return Y, foo_mul_pullback
end
```

Tour de ChainRules

Introduction

When to write rules

How to write rules

Basic example

Gotchas and helpers

RuleConfig

Array dilemma

How to test rules

Summary

```julia
struct Foo
    A::Matrix
    c::Float64
end
```

Struct gotchas

function

```julia
func_foo(foo::Foo, B) = 77
```

```julia
rrule(::typeof(func_foo), f::Foo, B)
```

callable struct

```julia
(foo::Foo)(B) = 77
```

```julia
rrule(foo::Foo, B)
```

constructor

```julia
Foo(A) = Foo(A, 0.0)
```

```julia
rrule(::Type{Foo}, A)
```

note that
`typeof(Foo) == DataType`
so `::typeof(Foo)` defines an `rrule` for all constructors!

If a function has multiple inputs or outputs, and only some of the derivatives have been worked out analytically, `@not_implemented` macro can be used

```julia
B̄ = @not_implemented(
    """
    The derivative w.r.t. B is not implemented:
    https://github.com/User/Package.jl/issues/77
    """
)
```

If all of the arguments of the function are non-perturbable,
`@non_differentiable` macro can be used
to automatically generate both the `frule` and the `rrule`

```
@non_differentiable joinpath(::AbstractString, ::AbstractString...)
```

If the function is defined on scalars*, i.e. `<:Number`, `@scalar_rule` macro can be used to automatically generate both the `frule` and the `rrule`

```
@scalar_rule sinh(x) cosh(x)
```

*multiple inputs and outputs are allowed

Tour de ChainRules

Introduction

When to write rules

How to write rules

Basic example

Gotchas and helpers

RuleConfig

Array dilemma

How to test rules

Summary

RuleConfig and calling back into AD

RuleConfig is a way to make rules conditionally defined, depending on the properties of the AD system.

This allows us to define a rule which calls forward mode AD inside the rule definition. This rule will only be used by AD systems which support forward mode.

Implemented with a trait-like system (not Holy Traits)

AD systems declare their properties

```
struct MyADRuleConfig <: RuleConfig{Union{Feature1, Feature2}} end
```

While rule authors can specify which properties are needed for the rule to be defined

```
# rrule that is only defined for ADs with `Feature1`
rrule(::RuleConfig{>:Feature1, }, f, args...) = ...

# frule that is only defined for ADs with both `Feature1` and `Feature2`
frule(::RuleConfig{>:Union{Feature1,Feature2}}, f, args...) = ...
```

Calling back into AD

Complementary properties: `HasReverseMode`, `NoReverseMode`
and similarly for forwards mode

AD that `HasReverseMode` needs to define `rrule_via_ad` for its RuleConfig subtype:

```
struct MyReverseOnlyADRuleConfig <: RuleConfig{Union{HasReverseMode, NoForwardsMode}} end

function ChainRulesCore.rrule_via_ad(::MyReverseOnlyADRuleConfig, f, args...)
    ...
    return y, pullback
end
```

Rules for higher order functions

```
function rrule(
    config::RuleConfig{>:HasForwardsMode},
    ::typeof(map),
    f::Function,
    x::Array{<:Real}
)

    y_and_ẏ = map(x) do xi
        frule_via_ad(config, (NoTangent(), one(xi)), f, xi)
    end
    y = first.(y_and_ẏ)
    ẏ = last.(y_and_ẏ)

    pullback_map(ȳ) = NoTangent(), NoTangent(), ȳ .* ẏ
    return y, pullback_map
end
```

Writing rules only for your own AD

Just dispatch on the RuleConfig (do not define a new feature)

```julia
struct MyADConfig <: RuleConfig{Union{Feature1, Feature2}} end
```

```julia
rrule(::MyADConfig, typeof(some_func), args...) = ...
```

Tour de ChainRules

Introduction

When to write rules

How to write rules

Basic example

Gotchas and helpers

RuleConfig

Array dilemma

How to test rules

Summary

Array dilemma

More formally: taking types that represent embedded subspaces seriously

This is complicated.

If you don’t understand, it is my fault, not yours.

I will make sure to wake you up when we move on.

Primal computation

```julia
function sum_array(A::AbstractArray)
    s = 0
    for i in eachindex(A)
        s += A[i]
    end
    return s
end

function sum_array(A::Diagonal)
    return sum_array(diag(A))
end
```

Summing an AbstractArray

Dispatch allows us to specialise

Pullback

For `A::Matrix`, which uses the fallback primal, the pullback is just a matrix of ȳ:

```julia
function rrule(::typeof(sum_array), A::AbstractArray)
    y = sum_array(A)
    sizeA = size(A)
    function sum_array_pullback(ȳ)
        return NoTangent(), fill(ȳ, sizeA)
    end
    return y, sum_array_pullback
end
```

Pullback

What do we want for `A::Diagonal`, which uses the specialised primal?

<u>Option 0:</u>

We can define a custom rrule.

But this approach means we have to write *very* many rules.

(and add new rules as new types appear)

Pullback

What do we want for `A::Diagonal`, which uses the specialised primal, if we have not defined a custom `rrule`?

Option 1:
Fallback to abstractly typed `rrule` with `A::AbstractArray`.

Option 2:
Let AD have a go at differentiating the specialised primal computation.

Pullback

What do we want for `A::Diagonal`, which uses the specialised primal, if we have not defined a custom `rrule`?

Option 1:

Fallback to abstractly typed `rrule` with `A::AbstractArray`.

⚠️ can return the wrong answer: e.g. a dense matrix tangent for `A::Diagonal`

⚠️ can be slower than Option 2

Option 2:

Let AD have a go at differentiating the specialised primal computation.

⚠️ can error if the specialised primal computation uses unsupported features (e.g. mutation)

⚠️ can be slower Option 1 (e.g. for loops are slow)

Pullback

What do we want for `A::Diagonal`, which uses the specialised primal, if we have not defined a custom `rrule`?

<u>Option 1:</u>

Fallback to abstractly typed `rrule` with `A::AbstractArray`.

⚠️ can return the wrong answer: e.g. a dense matrix tangent for `A::Diagonal`

⚠️ can be slower than Option 2

~~Option 2:~~

~~Let AD have a go at differentiating the specialised primal computation.~~

⚠️ ~~can error if the specialised primal computation uses unsupported features (e.g. mutation)~~

⚠️ ~~can be slower Option 1 (e.g. for loops are slow)~~

Pullback

What do we want for `A::Diagonal`, which uses the specialised primal, if we have not defined a custom `rrule`?

<u>Option 1:</u>

Fallback to abstractly typed `rrule` with `A::AbstractArray`.

⚠️ can return the wrong answer: e.g. a dense matrix tangent for `A::Diagonal`

⚠️ can be slower than Option 2

To fix these problems:

`ProjectTo` and `@opt_out`

⚠️ can return the wrong answer: e.g. a dense matrix tangent for `A::Diagonal`

Solve by making sure the tangent remains in the subspace of the primal:

create an object that knows how to project on the right tangent type
(i.e. knows the type, but also the size etc.)

```
project_A = ProjectTo(A)
```

and then project the tangent

```
project_A(tangent)
```

```
function rrule(::typeof(sum_array), A::Matrix)
    y = sum_array(A)
    sizeA = size(A)
    project_A = ProjectTo(A)
    function sum_array_pullback(ȳ)
        return NoTangent(), project_A(fill(ȳ, sizeA))
    end
    return y, sum_array_pullback
end
```

⚠️ can be slower than AD through specialised primal computation

Solve by providing a way to opt out of using the fallback rule:

```
@opt_out rrule(::typeof(sum_array), A::Diagonal)
```

for a particular function signature.

This makes it easy to check whether AD or fallback rule is faster.

Tour de ChainRules

Introduction

When to write rules

How to write rules

- Basic example
- Gotchas and helpers
- RuleConfig
- Array dilemma

How to test rules

Summary

Testing rules

Need to test rules with finite differencing methods.

`frule`

```
frule((ḟ, ȧrgs...), f, args...)
test_frule(f, args...)
```

`rrule`

```
rrule(f, args...)
test_rrule(f, args...)
```

Testing rules

Powered by automatic tangent generation

```
test_rrule(f, x)
```

Can* specify the tangent explicitly: `primal ⊢ tangent`

```
test_rrule(f ⊢ f̄, x ⊢ x̄; output_tangent=ȳ)
```

*sometimes *have to*

Testing AD gradients

Specify the rrule-like function

```
test_rrule(f, args...; rrule_f=rrule_via_ad)
```

Tour de ChainRules

Introduction

When to write rules

How to write rules

Basic example

Gotchas and helpers

RuleConfig

Array dilemma

How to test rules

Summary

ChainRules 1.0 is out! 🤞

new features

calling back into AD

`ProjectTo`

`@opt_out`

pullbacks handle Thunks

testing rules is easier

can test AD gradients

`@non_differentiable`

`@not_implemented`

used by AD systems

Diffractor

Nabla

ReversePropagation

Yötä

Zygote

many rules

100s of rules in ChainRules.jl

AbstractGPs.jl

BlockDiagonals.jl

DiffEqBase.jl

Hankel.jl

PDMatsExtras.jl

SpecialFunctions.jl

...

ChainRules 1.x/2.0 wish list

Higher order rules (2nd derivative)

Rules for mutating functions

Better solution for the array dilemma

Better solution for inplace accumulation

Wirtinger derivatives for complex numbers

Rules for jacobians

Ability to get a basis for anything

Tooling to support systems that can't augment the primal

Summary

🔗 ChainRules project is at 1.0 🎉

Integrated in many AD systems

100s of correct and efficient rules

Utilities to easily write more rules

Utilities to test rules (and AD gradients) using finite differences