ChainRules.jl
AD Agnostic Custom Rules for JuliaLang

Lyndon White (Invenia Labs)
EuroAD 2021

Alex Arslan
David Widmann
Jarrett Revels
Lyndon White
Michael Abbott
Miha Zgubic

Matt Brzezinski
Nick Robinson
Seth Axen
Simeon Schaub
Will Tebbutt
Yingbo Ma
What is the ChainRules Project?

Extensible, AD engine-agnostic, set of AD rules (custom primitives).

It is made up of three key packages:

**ChainRulesCore.jl**: system for defining rules (custom primitives).
**ChainRules.jl**: defines all the rules for Julia’s standard libraries.
**ChainRulesTestUtils.jl**: robust system for testing rule definitions.

and 4 other auxiliary packages: ChainRulesOverloadGeneration.jl, ChainRulesDeclarationHelpers.jl, FiniteDifferences.jl, DocThemeIndigo.jl
Why?

AD primitive rules have three keys uses.

**Fundamental language primitives:** At least instructions like addition, but likely also things implemented in another language.

**Domain knowledge:** as experts we can provide extra knowledge that makes differentiation possible/much faster like the *implicit function theorem* for optimizations, or *local sensitivity analysis* methods for differential equations.

**Work around in-perfections in AD engine:** sometimes AD doesn’t generate the optimal code.
How does this hang together?

Operator overloading ADs:
- Nabla.jl
- ReverseDiff.jl

Source-to-source ADs:
- Zygote.jl
- Diffractor.jl

Julia and standard libs:
- Base
- LinearAlgebra

Packages:
- SpecialFunctions.jl
- NNlib.jl
- AbstractFFTs.jl
- DiffEqSensitivity.jl

Note the lack of direct dependencies between AD systems (left) and Packages (lower right).

Instead, e.g. Flux.jl (neural net package) depends on Zygote.jl and NNLib.jl.
Roll-out (as of November 2021)

Version 1.0.0 was released at JuliaCon, in August 2021. ChainRulesCore is used to provide rules by about 70 packages. It is used by 6 AD engines.

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Scalar</th>
<th>Array</th>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zygote.jl</td>
<td>IR SCT</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Diffractor.jl</td>
<td>IR SCT</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Yota.jl</td>
<td>IR Tape</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ReverseDiff.jl</td>
<td>OpOv Tape</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>Nabla.jl</td>
<td>OpOv Tape</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ReversePropagation.jl</td>
<td>OpOv Trace</td>
<td>✔</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Key Requirements
Key Requirements

- Mathematically sensible
- Extensible
- Represent anything any AD can handle
- Able to call back into AD
- Minimally magic
- Able to share work between primal and pullback/pushforward.
Key Requirement: mathematically sensible

- Require tangents to be represented with types that can be added to primal, and basically define vector spaces.
- Don’t allow real numbers to have complex tangents.
- Don’t allow diagonal matrices to have dense tangents.
- Don’t pretend that bools and integers represent continuous quantities.
- Distinguish between: not implemented, zero as in not part of computation, and the tangent space not existing.
How do we define a tangent/cotangent type

Technically, we only really need something that defines addition between other tangent types.

For practicality we have it define to have everything you would expect from a vector space.

More linear operators are useful for defining rules.

We also define addition to the primal value, which allows it to be directly used for optimization; and resolves disputes about if two representations are equal.
Key requirement: Extensible

Predecessor was DiffRules.jl (2017), for scalar functions.

All rules had to be implemented in DiffRules.jl, used metaprogramming in loading of Nabla.jl.

Back in the old-bad-days:
To define a new rule, either Nabla had to load your package (Foo.jl) and define rules, or you had to load Nabla (Bar.jl). (if possible at all)
Key requirement: must be able to represent anything any AD can handle

- Definitely need to be able to represent tangents for scalars and matrices.
- But also need to also represent structures and tuples
- Need to represent rules for callable object (functors/closures).
Key requirement: must be able to call back into AD

- A rule for a higher order function, like map or reduce, needs to use AD inside it for the called function.
- But we can’t hard code which AD to use.
- Further, might want to use reverse-via-forwards if both are available.

So need abstraction over running AD, and different rules need to hit depending what AD modes are available.
Key requirement: must be able to share work between primal and pullback/pushforward

- Must be able to remember arbitrary information from primal computation to use in the pullback.
- Must be able to change primal computation to use different algorithm (e.g. computing eigenvalues + vectors rather than just eigenvalues) to have extra information needed for derivatives.
- In forward mode: must be allowed to compute primal result and tangent as a single operation (fused pushforward and primal).
Key requirement: minimally magic

- Just overloading functions.
- No macro-based DSLs required.
- Always represent tangent for function object itself.
- Keyword arguments are just keyword arguments, not a special object from Julia’s internals.

When magic is required, make the AD author have to engage with it. Not the rule author.
Key challenge: Polymorphism
What is polymorphism?

Polymorphism is the one of key promises of Object Orientation.

One way to consider polymorphism is as the ability to provide a common interface over multiple different implementations.

For example: a diagonal matrix could be represented just by a vector, and have efficient implementation of matmul as column-wise scaling. But still be indexable as if a dense array.
Polymorphic Matrixes are surprisingly rare

Even though BLAS has special cases for dense, symmetric, trigangular, hermitian, banded, and triangular banded matrixes, plus transposed and conjugate transposed variants of the above, we do not often see that reflected in typed libraries.

- Numpy (python) has 3 matrix types.
- Eigen (C++) has 3 matrix types (though technically is extensible).
- Blaze (C++) has 9 matrix types, though they can be combined.
- Julia has 29 matrix types in the standard library alone, which can be combined.
Once you have polymorphism you have a world of new problems for AD rules

- Should rules be written for functions accepting abstract types?
- Escaping the manifold: a sparse matrix must not have a dense derivative.
- Converting between structural and natural tangent representations.

I am honestly not sure why this is not run into more often. I guess because most technical code is very procedural and is unable to use object orientated features, especially if the ”type-bullet” has already been fired to start using a operator overloading AD.

My bold prediction is that many AD systems are going to be running into these in the next 2 years.
Should rules be written for functions accepting abstract types? 1

The general Julia principle is to have generic fallbacks functions, and introduce specialized overloads when domain knowledge is useful.

However, for AD rules there is a counter argument: there is always a specialized fallback that is available by actually running AD, rather than hitting the rule.
Should rules be written for functions accepting abstract types? 2

But sometimes what AD will generate is much slower than hitting a abstract rule, because domain knowledge was doing something useful.

But sometimes the abstract rule is much slower than letting AD do its thing, because the specialized primal function also contains useful domain knowledge.

Our solution has been to allow abstractly typed rules, but make it easy to opt-out of them on a case-by-case basis.
Consider a rule for the sum operation. Its pullback should be an array matching the primal with copies of the cotangent. Care must be taken when we say matching the primal. If you summed a diagonal matrix (where it is encoded into the type that it is definitely diagonal) then you must not end-up with nonzero off the diagonal. Similar things happen for Real primals, getting Complex cotangents.
We resolve this with a ProjectTo operation. It is defined based on the primal value. It remembers just enough things to drag any tangent type back down to the manifold where the primal lives. For example:

- Dropping off-diagonals on Diagonal
- Enforcing symmetry on Symmetric

As a bonus it also lets us:

- Fix the precision of any floating point types that might have been promoted up the 64 bit from 32 bit primals.
- Reattach named dimensions, etc to named tensorish types.
Core problem is that we can represent (e.g.) a tangent to a structured matrix as either being struct-like or matrix-like.

For constructors, we really want it represented like a struct, so we can attribute the tangents back to the fields initialisers.

For most mathematical operations we want it represented like a matrix, so we can run linear algebra operations on it.
Trivial case: natural tangent has the same type as primal. You thus know all the fields and so can just reinterpret it as a structural tangent.

Not all cases are like this. Sometimes you need a different matrix type (though often can use projection (c.f. Escaping the Manifold) to get back to one that is same as primal).

For the general case however the answer is that the pullback of the collect (densify) operation is exactly what is needed to do this conversion.

See Will Tebbutt’s WIP PR about this: https://github.com/JuliaDiff/ChainRulesCore.jl/pull/449
Summary

- ChainRules project is AD-engine agnostic custom rules in JuliaLang.
- It is now fairly widely used.
- Getting the right abstractions took some care.
- Polymorphism makes for hard problems.
Appendix
Consider a function $f(a, b, c)$. If you only want $\bar{a}$ and $\bar{b}$, then there may be a more efficient rule than if you wanted $\bar{a}$, $\bar{b}$ and $\bar{c}$.

We currently have limited support for that via thunking work for each output.

This is useful for Enzyme and operator overloading ADs.
This is a classic array of structs vs struct of arrays question. We make the call that derivative information (tangents/cotangents) should always remain at the outer-most level. This is the way that allows libraries that assume contiguous arrays of IEEE floats (i.e. BLAS) to be used. It also avoids needing to have the layout of objects changed from the primitive original, which may not always be possible.

It is more inconvenient for use in libraries that follow the opposite convention. But almost by construction such libraries operate at the scalar level and are unable to act upon rules defined for array/struct type objects. And for scalar typed objects the question is moot, the representation is identical.
We thus allow multiple tangent types

This leads to suffering, but it seems to be the only sensible option in the face of dynamic language with extensive polymorphism. As long as the type meets requirements above.

**Natural Tangent** : anything that would represent a difference of primal values. E.g. Number, Matrix, Millisecond.

**Structural Tangent** : for representing structure tangents as a collection of the tangents of all fields.

**Thunk, and InplacableThunk** : for deferred computation

**ZeroTangent** : for not occurring in computation

**NoTangent** : for the tangent space not existing

**NotImplemented** : for if one tangent hasn’t been implemented.