```org
-*- julia-snail-extensions: (ob-julia) -*-
#+TITLE:  Zero to Julia (with AD)
#+AUTHOR: Kiran Shila
#+STARTUP: latexpreview

* Hello

- I'm Kiran
  + EE PhD Student
  + Radio Astronomy
  + Programming Language Enthusiast

* Why learn a new language

- What are the requirements for scientific computing?
  + Fast
  + Iterable
  + Generic

* Intro to Julia

** What is Julia?

Julia is a high-level, high-performance, dynamic programming language.

While it is a general-purpose language and can be used to write any
application, most of its features are well suited for numerical analysis
and computational science.

** The REPL

Open REPL, showcase
- REPL modes, Pkg, Shell, Help
- Primitives
  + numbers, complex, rationals
- Strings, Characters
- Blocks, for, while, if
- Function syntax
- Struct syntax
  + generics
- Macros
  + Show, Benchmark

** Multiple Dispatch and Philosophy

The most powerful tool you have

*All named functions are generic*

Functions are things you *do* to data, the data doesn't define it.

*** Yes, but what is it?

$f = x+y$ is an abstract function of two variables. The behavior of this
function is
dependent of what $x$ and $y$ are.

Multiple dispatch is a run-time feature that specializes a function call
based
on the entire call signature.
```

```org
** Unreasonable Effectiveness, an Example

Multiple dispatch maximizes code sharing

Code reuse comes in two forms:
- Types (Shared across packages)
- Algorithms (Applied across types)

Both stem from MD

** bUt YoURe DeScriBing FunCtIOn OveRLoaDinG

Are you sure?
#+begin_src c++
class Pet {
public:
  string name;
};

string meets(Pet a, Pet b) { return "FALLBACK"; }

void encounter(Pet a, Pet b) {
  string verb = meets(a, b);
  cout << a.name << " meets " << b.name << " and " << verb << endl;
}
#+end_src

** bUt YoURe DeScriBing FunCtIOn OveRLoaDinG

#+begin_src c++
class Dog : public Pet {};
class Cat : public Pet {};

string meets(Dog a, Dog b) { return "sniffs"; }
string meets(Dog a, Cat b) { return "chases"; }
string meets(Cat a, Dog b) { return "hisses"; }
string meets(Cat a, Cat b) { return "slinks"; }
#+end_src

** bUt YoURe DeScriBing FunCtIOn OveRLoaDinG

#+begin_src c++
int main() {
  Dog fido;     fido.name = "Fido";
  Dog rex;      rex.name = "Rex";
  Cat whiskers; whiskers.name = "Whiskers";
  Cat spots;    spots.name = "Spots";

  encounter(fido, rex);
  encounter(fido, whiskers);
  encounter(whisters, rex);
  encounter(whisterks, spots);

  return 0;
}
#+end_src

** Any Guesses?
** Any Guesses?

=clang++ pets.cxx -o pets && ./pets=
#+begin_src
Fido meets Rex and FALLBACK
Fido meets Whiskers and FALLBACK
Whiskers meets Rex and FALLBACK
Whiskers meets Spots and FALLBACK
#+end_src

C++ solves this statically with templates, no RTTI solution

** RGB and Sharing Types

Say you have a type that holds an RBG color value, =RGB=.
The package that includes this type has basic operations that make sense
to the
author. How do you add functionality?

** RGB in Julia

You just add the method.
This works on existing operations

#+begin_src julia
Base.zero(::Type{RGB}) = RGB(0,0,0)
#+end_src

And for writing new ones
#+begin_src julia
dotc(x::RGB, y::RGB) = 0.200*x.r*y.r + 0.771*x.g*y.g + 0.029*x.b*y.b
#+end_src

** RGB in *Other Languages*

How would you do this in other languages?
In class-based OOP, methods are *textually inside* the class definition.
The data container is declaring what that data is able to do.

** RGB in *Other Languages*

To add methods, you can
- Edit the class
- Inherit the class

** Why is this a problem?

To edit the class you must convince the author it's a good idea (do they
want to
maintain your code?).

If everyone does that, the class becomes bloated with everyone's fancy
feature.

You can't change it without breaking changes.

** Why is this a problem?

Inheriting is just as bad.
New name, =MyRGB= is semantically different from =RGB=
```

You *must* have instances of =MyRGB= to use your functionality, not =RGB= from
elsewhere.
- This is kinda solved with dependency injection, but sucks
If you have multiple people's custom types and you want all the features, you
need multiple inheritance

** Two bad choices

What do people actually do?
- Give up
  + Use =f(x,y)= instead of =x.f(y)=, ruining reuse as this can't be generic
    - No, not even templates fix this as those are static
- Don't share
  + Write your own RGB
  + The most popular option in OOP

** Julia Perspective

In Julia, you can define methods on types whenever you want
Generic functions are namespaced (unlike methods in OOP)

** The "Expression Problem" solved with MD

Can you easily and "correctly" do both

1. Define *new types* to which *existing operations* apply
   a. Easy in OOP, hard in FP
2. Define *new operations* which apply to *existing types*
   a. Easy in FP, hard in OOP

** Not Unique to Julia

- =multipledispatch= in python (opt-in)
- CLOS in CL (opt-in, inspiration for Julia, relativley slow)
- ML (started the idea in 1973), C++ didn't invent generics
- Dylan and C# have native support

** The Julia Compiler

How does this work?

[[file:compiler.png]]

** Not Perfect

- Lack of traits
  + What do traits mean if *everything* is generic
- Compiler latency, TTFP, Warm up, etc.
- Memory consumption
  + Perfectly acceptable for PC or cluster applications
  + Not for mobile, embedded, daemons, etc.
- Julia FFI
  + Massive runtime -> Hard to embed
- Other smaller issues
  + Static analysis

```
  + Ecosystem
  + Stability

* Automatic Differentiation

*** Forward Mode (Dual Numbers)

- Hypercomplex number system from the 19th century
- $a+b\varepsilon$ where $a$ and $b$ are real
- $\varepsilon$ is a (nonzero) symbol that satisfies $\varepsilon ^{2} =
0$

*** Where do derivatives fit in?

Consider an arbitrary nth order polynomial

$P(x) = p_0 + p_1x + p_2x^{2} + \dots + p_nx^{n}$

We can extend the domain of this polynomial from the reals to the duals by
just

\begin{align*}
P\left(a + b\varepsilon\right) &= p_0 + p_1(\left a+b\varepsilon\right) +
\dots + p_n\left(a+b\varepsilon\right)^{n} \\
&= p_{0} + p_{1}a + p_{2}a^{2} + \dots + p_{n}a^{n} + p_{1}b\varepsilon +
2p_{2}ab\varepsilon + \dots + np_{n}a^{n-1}b\varepsilon
\end{align*}

Which is exactly equivalent to

$P\left(a\right) + bP'\left(a\right)\varepsilon$

*** Limit Perspective

\begin{equation*}
f'\left(a\right) = \lim_{\varepsilon \rightarrow 0} \frac{f\left(a +
\varepsilon\right) - f\left(a\right)}{\varepsilon}
\end{equation*}

Dual numbers are an algebraic way of expressing the same thing
Assume $\varepsilon$ is *just* an infinitesimal, just from algebra

*** Dual Numbers (Taylor's Version)

\begin{equation*}
f(a+b\varepsilon) = \sum_{n=0}^{\infty}
\frac{f^{(n)}(a)b^{n}\varepsilon^{n}}{n!} = f(a) + bf'(a)\varepsilon
\end{equation*}

*** Free Chain Rule

Again:

\begin{equation*}
f(a+b\varepsilon) = f(a) + bf'(a)\varepsilon
\end{equation*}

So if we want $g(f(x))$, we just do the same thing:

\begin{align*}
```

```
g\left(f(a+b\varepsilon)\right) &= g\left(f(a) + bf'(a)\varepsilon\right)
\\
&= g(f(a)) + bf'(a)g'(f(a))\varepsilon
\end{align*}

Tada! The same algebra on the duals induces the chain rule from function
composition.

*** What about non-polynomials?

We just say the rules for transcendental functions. i.e.

$\sin(a+b\varepsilon) = \sin(a) + b\cos(a)\varepsilon$

*** Math Nerd Zone
- The set of Duals ($\mathds{D}$) are a topological vector space
- The duals are basically $\mathds{R}^{2}$ made into an algebra by
  $(a,b)\cdot(c,d) = (ac,ad+bc)$
-
  $\mathds{R}^{2}$ is a subset of $\mathds{D}$ by $a$ with $a +
0\varepsilon$

*** More Math Nerd Zone

- You've got involution (like the complexes)
  $\overline{a+b\varepsilon} = a - b\varepsilon$
- Magnitudes
$|a+b\varepsilon| = |a|$
But this gets weird as $|z|^{2} = z\overline{z}$. BUT, $|z| = 0$ does not
imply $z=0$
This has some kinda scary consequences as the order matters in certain
calculations. Unlike the reals or complexes, the duals don't form a field.

*** Quick Summary

- Dual numbers provide an algebraic way of expressing the exact derivative
of a
  function at a point
- The algebra perspective yields a first-order model
  - Not a showstopper
- Easy to implement

** Implementation

*** Simple in Julia with MD

Tons of ways to do this

#+begin_src julia
struct Dual{T <: Number} <: Number
    x::T # Primal
    ϵ::T # Tangent
end

Base.show(io::IO, a::Dual) = print(io, "$(a.x) + $(a.ϵ)ϵ")

Dual(x::T) where T<:Number = Dual(x,one(T))
#+end_src

*** Working with other numbers
```

We need to setup the rules for how Julia knows when to convert/promote
between
numerical types.

For instance

```
#+begin_src julia
promote_type(Int32,Float64)
#+end_src
```

*** Promotion Rules

Multiple dispatch handles the combinations

```
#+begin_src julia
import Base: promote_rule

# Promotion between Duals of different inner types
function promote_rule(::Type{Dual{T}}, ::Type{Dual{R}}) where {T,R}
  Dual{promote_type(T,R)}
end

# Promotion up to dual
function promote_rule(::Type{Dual{T}}, ::Type{R}) where {T,R}
    Dual{promote_type(T,R)}
end
#+end_src
```

*** Conversions

The fact that the promotions exist doesn't imply that we can readily
convert, so
we need functions for that too

```
#+begin_src julia
import Base: convert

function convert(::Type{Dual{T}}, x::Dual) where {T}
    Dual(convert(T, x.x), convert(T, x.ϵ))
end

function convert(::Type{Dual{T}}, x::Real) where {T}
    Dual(convert(T, x), zero(T))
end
#+end_src
```

*** Rules

```
#+begin_src julia
import Base: +, -, *, /
# Sum Rule
a::Dual + b::Dual = Dual(a.x + b.x, a.ϵ + b.ϵ)
# Subtraction Rule
a::Dual - b::Dual = Dual(a.x - b.x, a.ϵ - b.ϵ)
# Product Rule
a::Dual * b::Dual = Dual(a.x * b.x, a.ϵ * b.x + b.ϵ * a.x)
# Quotient Rule
a::Dual / b::Dual = Dual(a.x / b.x, (b.x * a.ϵ - a.x * b.ϵ) / b.x^2)
#+end_src
```

```
*** Are we being consistent?

#+begin_src julia
Dual(0,1)
#+end_src

We didn't even define what =^= is

#+begin_src julia
Dual(0,1)^2
#+end_src

#+begin_src julia
h(x) = 3*x + 5*x^2
#+end_src

#+begin_src julia
h(10)
#+end_src

#+begin_src julia
h(Dual(10,1))
#+end_src

*** More rules

#+begin_src julia
import Base: sin,cos
sin(d::Dual) = Dual(sin(d.x),  d.ϵ * cos(d.x))
cos(d::Dual) = Dual(cos(d.x), -d.ϵ * sin(d.x))
#+end_src

#+begin_src julia
g(x) = sin(x)^2 + 10cos(3x^5)
#+end_src

#+begin_src julia
g(Dual(3.13159))
#+end_src

*** High Order Functions

#+begin_src julia
derivative(f, x::T) where {T<:Number} = f(Dual(x, one(T))).ϵ
#+end_src

#+begin_src julia
derivative(sin,3.14159)
#+end_src

*** Crazy composability (Party Trick)

Let's bring in the Python FFI layer

#+begin_src julia
using SymPy
@syms x
#+end_src
```

```
#+begin_src julia
g(x)
#+end_src

#+begin_src julia
derivative(g,x)
#+end_src

*** Upsides and Downsides

- Fast
- Callgraph complexity

#+begin_src julia
big_fn(x) = sin(x^2) / x
#@code_typed big_fn(1.0)
@code_typed big_fn(Dual(1.0,1.0))
#+end_src

** Reverse Mode (Zygote, SSA, Metaprogramming)

- Forward Mode is not the only option
- Reverse Mode introduces another trade-space (Time for Memory)

*** How Reverse Mode Works

- First published in 1976 by Seppo Linnainmaa
- Function is derived in two passes
  + First pass calculates all the intermediate values and stores the
    instructions that produced them (Gradient Tape)
  + Second pass applies the chain rule in reverse using the adjoints of
the
    operators on the tape and the intermediate values

*** Why would you want RM

If your function is $\mathds{R}^{N} \rightarrow \mathds{R}^{M}$ where $M
\ll N$,
you only need $M$ "sweeps" to "backpropagate" gradients versus $N$
"sweeps" in
forward mode

Think: Big ML Models

*** Another win for Julia

- Julia uses LLVM for it's "second layer IR"
- LLVM is Static Single Assignment (SSA)
- Gradient tape for free
- IR -> IR transformation to get easy access to tape for reverse passes

*** Packages

- Zygote (Backend for Flux)
- Enzyme (LLVM IR to IR)

** Hybrid Modes

- Both forward and reverse mode are "extremes"
- Finding the minimum number of operations to compute the gradient is the
```

  Optimal Jacobian Accumulation problem and is NP-Complete
- New methods everyday (Lots in Julia)

* Optimization

** Gradients with Duals

The algebra perspective doesn't let you get gradients as tangents aren't unique.

There are ways to do it with flags, but the simplest solution here is just to do all of
them one at a time in a loop (i.e. not as fast as it could be)

For every parameter of f, we set one to be derived, the rest are constants.
```
#+begin_src julia
function gradient(f, xs...)
  gs = []
  for (i, x) in enumerate(xs)
      if i == 1
        push!(gs, f(Dual(x), xs[2:end]...))
      else
        push!(gs, f(xs[1:(i-1)]..., Dual(x), xs[(i+1):end]...))
     end
  end
  [g.ϵ for g in gs]
end

gradient(f,xs::AbstractVector) = gradient(f,xs...)
#+end_src
```

** Does it work?

Yes
```
#+begin_src julia
f(x,y) = 3x + 10y^2
gradient(f,1,2)
#+end_src
```

** Gradient Descent

Given some cost function f paramaterized by θ

$\theta_{n+1} = \theta_{n} - \gamma \del f'(\theta_{n})$

Our simple "update rule" updates the parameters with the negative gradient times
some "learning rate"

Considerations for
- Convergence speed
- Global minima
- etc.

This is the most simple update rule, there are tons more

```
#+begin_src julia
gd(θ,grads,γ) = @. θ - γ * grads
#+end_src
```

```
** Modeling
#+begin_src julia
struct Polynomial
    θ
end

(m::Polynomial)(x) = sum(x^(i-1)*coeff for (i,coeff) ∈ enumerate(m.θ))

linear = Polynomial([1.0,2.0])
#+end_src

Broadcasting works
#+begin_src julia
linear.(1:10)
#+end_src

Derivatives work
#+begin_src julia
derivative(linear,10)
#+end_src

** Training

If we have some data and we want to fit a model, we want to *update* the
model
itself based on some fit error.

Mean squared error is typical

#+begin_src julia
mse(y,ŷ) = sum((y .- ŷ).^2) / length(y)
#+end_src

Let's generate some fake data to train on and make sure =mse= works

#+begin_src julia
xs = collect(-10:0.1:10)
ys = linear.(xs) .+ 5 .* rand(length(xs))
mse(ys,linear.(xs))
#+end_src

Now, the goal is to minimize this error w.r.t. a model

#+begin_src julia
model = Polynomial(zeros(2))

update(θ,grads) = gd(θ,grads,0.001)

function train!(model::T,xs,ys,update,cost) where {T}
    θ_old = model.θ
    grads = gradient((θ...)->cost(ys,T(θ).(xs)),θ_old)
    model.θ .= update(θ_old,grads)
    return cost(ys,model.(xs))
end
#+end_src

#+begin_src julia
train!(model,xs,ys,update,mse)
```

```
#+end_src

*** Generic on Polynomial

#+begin_src julia
model_cubic = Polynomial(zeros(4))
ys_cubic = @. 3xs^3 + 10xs^2 + 7xs + 1 + 500rand()
# Smaller learning rate
update_cubic(θ,grads) = gd(θ,grads,6e-6)
#+end_src

#+begin_src julia
train!(model_cubic,xs,ys_cubic,update_cubic,mse)
#+end_src

*** Generic on Model

This =train!= function is entirley generic, so as long as we have rules
defined
for the eventual gradient calculation, anything works!

#+begin_src julia
struct StrangeModel
    θ
end

(m::StrangeModel)(x) = m.θ[1]*sin(x) + x*m.θ[2] / cos(x)

model_strange = StrangeModel(zeros(2))
ys_strange = @. 0.8*sin(xs) + 0.2*xs/cos(xs) + rand()
# Even smaller learning rate
update_strange(θ,grads) = gd(θ,grads,4e-4)
#+end_src

#+begin_src julia
train!(model_strange,xs,ys_strange,update_strange,mse)
#+end_src

* Wrap Up

- Learn Julia!
  + Accelerate your scientific computing with generic code
  + Compose with other *fast* libraries
  + Differentiate all the things
```