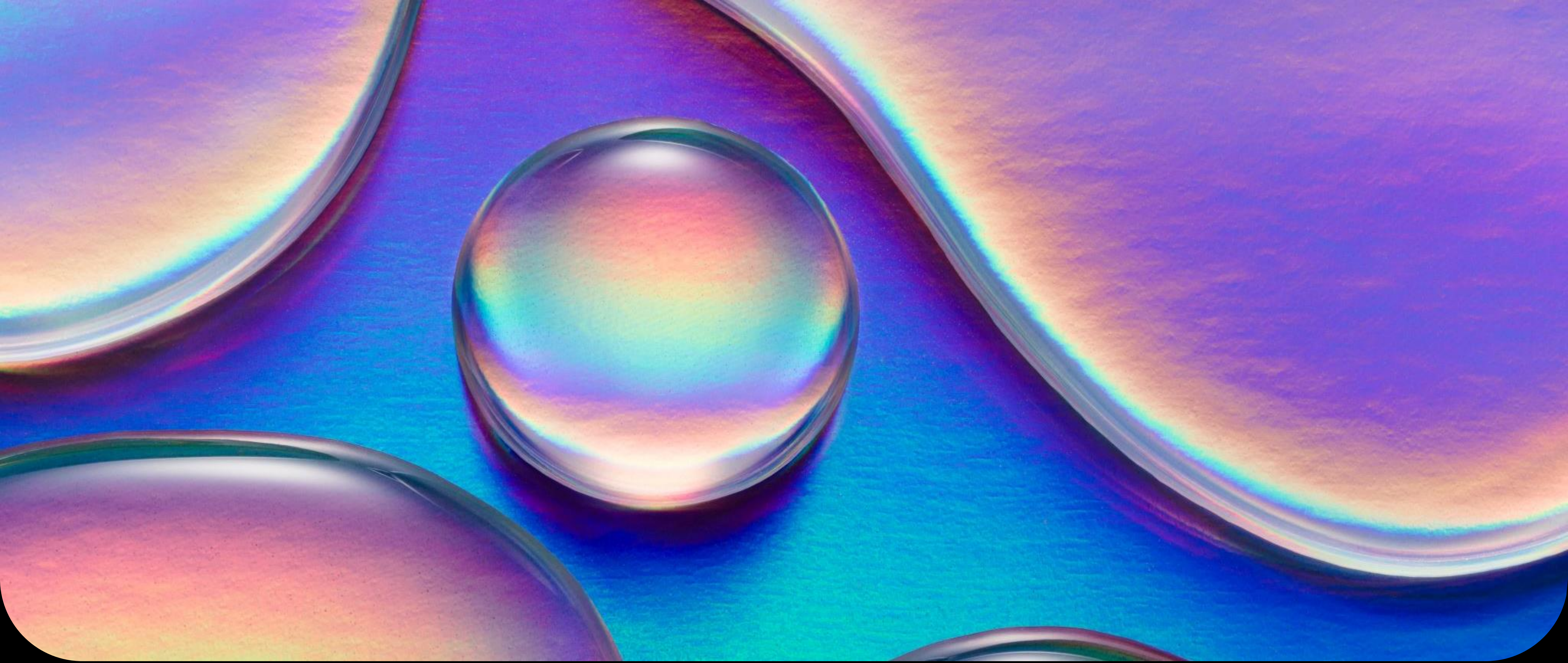


The Decapodes Pipeline

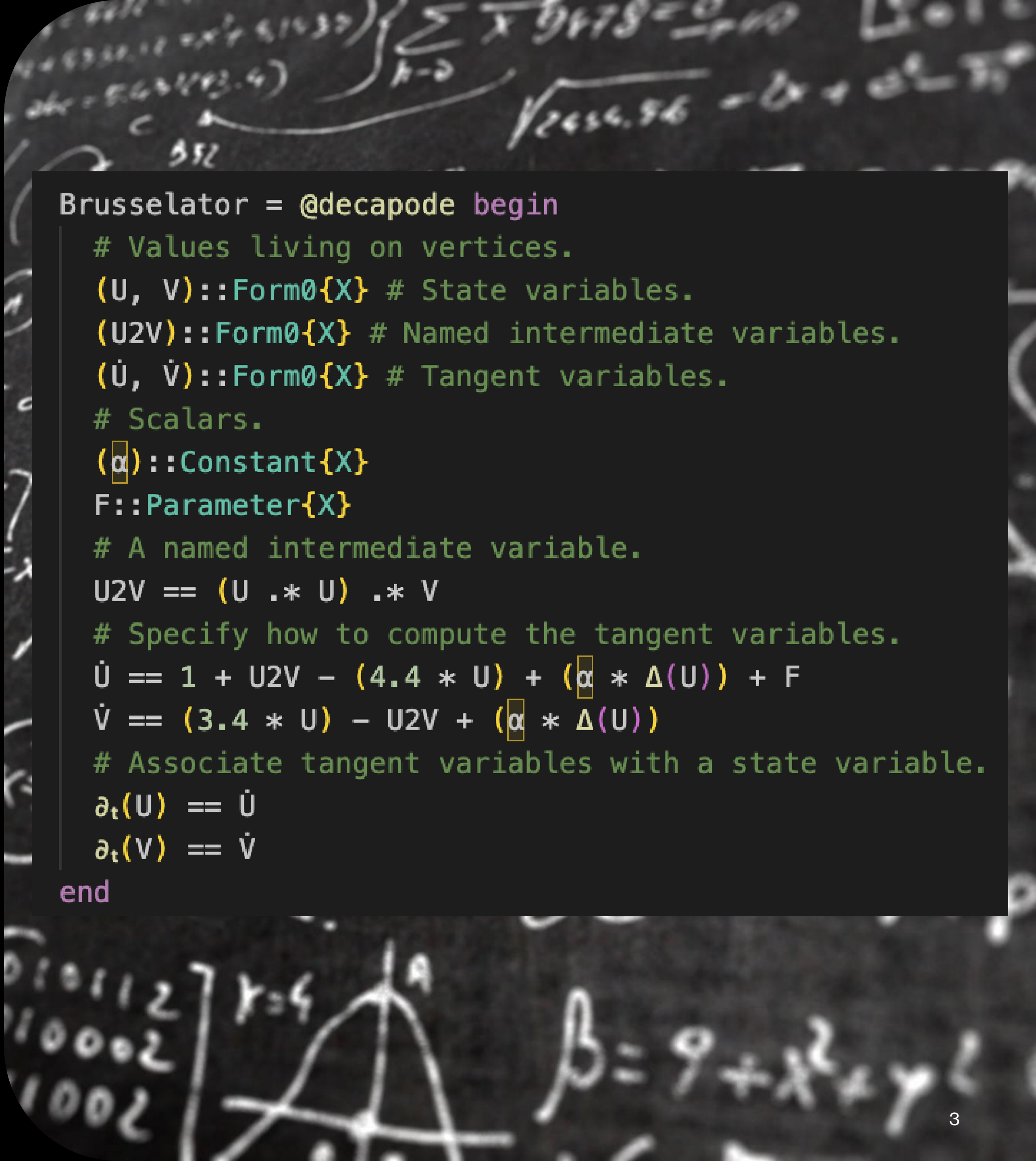
Making Multiphysics Simulations for Fun and Non-Profit

Precompile Your Packages

- Decapodes
- CombinatorialSpaces
- OrdinaryDiffEq
- GeometryBasics: Point2D, Point3D
- Catlab.Graphics (Optional)
- GLMakie (Optional)

Writing out the Physics

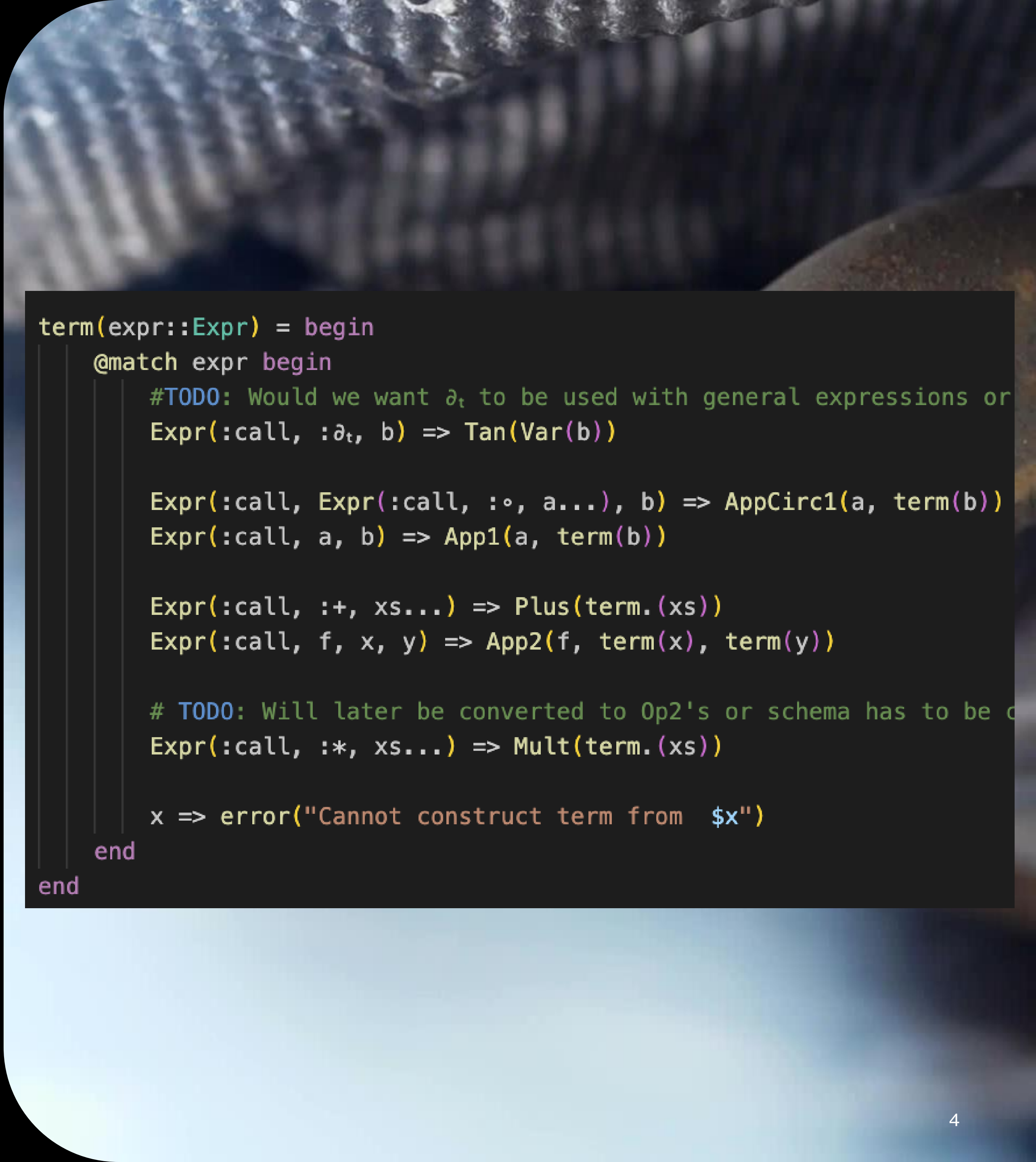
- Decapodes supports parsing a simple grammar that encodes many physics equations
- Allowances for Forms/DualForms, Numeric Literals and user supplied Constants and Parameters
- Time Derivative is a special operator



```julia
Brusselator = @decapode begin
  # Values living on vertices.
  (U, V)::Form0{X} # State variables.
  (U2V)::Form0{X} # Named intermediate variables.
  (U̇, V̇)::Form0{X} # Tangent variables.
  # Scalars.
  (α)::Constant{X}
  F::Parameter{X}
  # A named intermediate variable.
  U2V == (U .* U) .* V
  # Specify how to compute the tangent variables.
  U̇ == 1 + U2V - (4.4 * U) + (α * Δ(U)) + F
  V̇ == (3.4 * U) - U2V + (α * Δ(U))
  # Associate tangent variables with a state variable.
  ∂ₜ(U) == U̇
  ∂ₜ(V) == V̇
end
```

Under the Hood (parse_decapode)

- Takes in an Expr, produced by the quote block and produces a DecaExpr
- DecaExpr consists of main types Judgements and Equations
- Judgements house variable information (name, type, space)
- Equations hold AST representations for each line in the physics



```
term(expr::Expr) = begin
    @match expr begin
        #TODO: Would we want ∂ₜ to be used with general expressions or
        Expr(:call, :∂ₜ, b) => Tan(Var(b))

        Expr(:call, Expr(:call, :∘, a...), b) => AppCirc1(a, term(b))
        Expr(:call, a, b) => App1(a, term(b))

        Expr(:call, :+, xs...) => Plus(term.(xs))
        Expr(:call, f, x, y) => App2(f, term(x), term(y))

        # TODO: Will later be converted to Op2's or schema has to be c
        Expr(:call, :*, xs...) => Mult(term.(xs))

        x => error("Cannot construct term from  $x")
    end
end
```

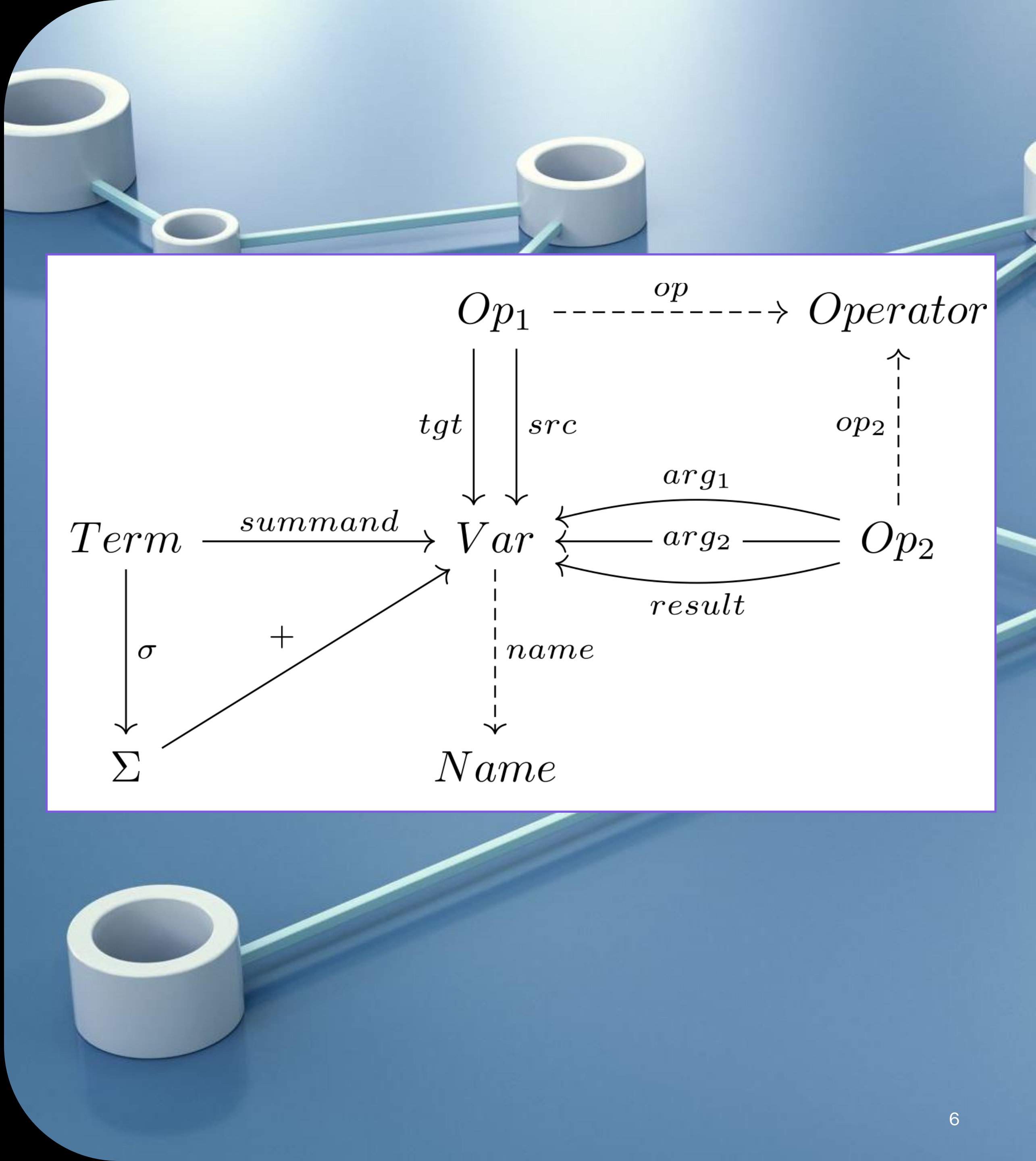
Under the Hood (Constructor)

- Takes in the DecaExpr and produces an ACSet representing the physics
- DecaExpr consists of main types Judgements and Equations
- Judgements house variable information (name, type, space)
- Equations hold AST representations for each line in the physics

```
function SummationDecapode(e::DecaExpr)
    d = SummationDecapode{Any, Any, Symbol}()
    symbol_table = Dict{Symbol, Int}()

    for judgement in e.context
      var_id = add_part!(d, :Var, name=judgement.var.name, type=judgement.dim)
      symbol_table[judgement.var.name] = var_id
    end

    deletions = Vector{Int}()
    for eq in e.equations
      eval_eq!(eq, d, symbol_table, deletions)
    end
    rem_parts!(d, :Var, sort(deletions))

    recognize_types(d)

    fill_names!(d)
    d[:name] = normalize_unicode.(d[:name])
    make_sum_mult_unique!(d)
    return d
end
```

The Decapode ACSet

- Essentially a database with the schema shown here
- Represents all the physics in a dataflow diagram which we can traverse like a DAG
- This allows for easy querying and modification of the data held
- Also allows for easy viewing of the overall physics (to_graphviz)



The Decapode ACSet

- Essentially a database with the schema shown here
- Represents all the physics in a dataflow diagram which we can traverse like a DAG
- This allows for easy querying and modification of the data held
- Also allows for easy viewing of the overall physics (to_graphviz)

SummationDecapode{Any, Any, Symbol} {Var: 21, TVar:2, Op1:4, Op2:8, Σ:3, Summand:7, Type:0, Operator:0, Name:0}

Var	type	name
1	Form0	U
2	Form0	V
3	Form0	U2V
4	Form0	One
5	Form0	U̇
6	Form0	V̇
7	Constant	α
8	Parameter	F
9	infer	•1
10	infer	•2
11	infer	•3
12	Literal	1
13	infer	sum_1
14	infer	•4
15	Literal	4.4
16	infer	•5
17	infer	•6
18	infer	sum_2
19	infer	•7
20	infer	•8
21	Literal	3.4

TVar	incl
1	5
2	6

Op1	src	tgt	op1
1	1	17	Δ
2	1	18	Δ
3	1	5	∂_t
4	2	6	∂_t

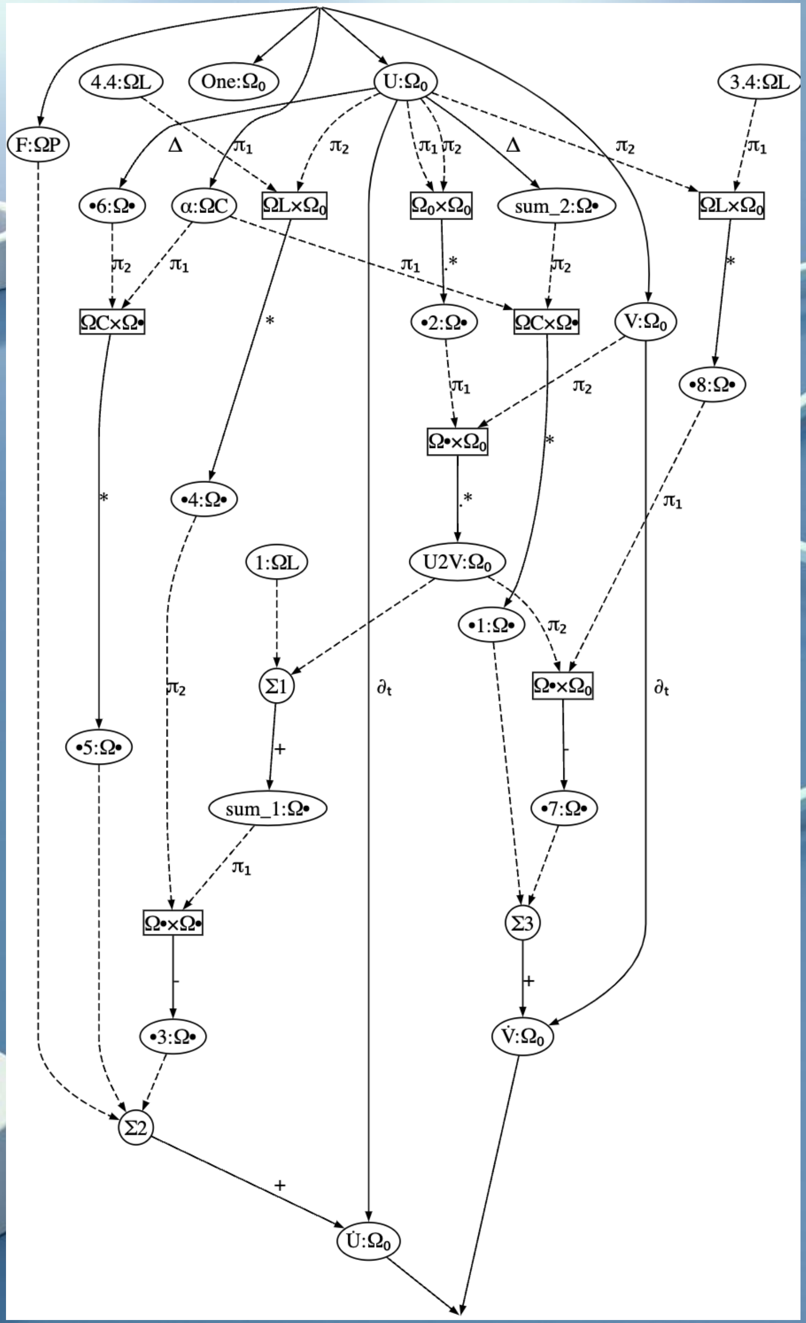
Op2	proj1	proj2	res	op2
1	1	1	10	.*
2	10	2	3	.*
3	15	1	14	*
4	13	14	11	-
5	7	17	16	*
6	21	1	20	*
7	20	3	19	-
8	7	18	9	*

Σ	sum
1	13
2	5
3	6

Summand	summand	summation
1	12	1
2	3	1
3	11	2
4	16	2
5	8	2
6	19	3
7	9	3

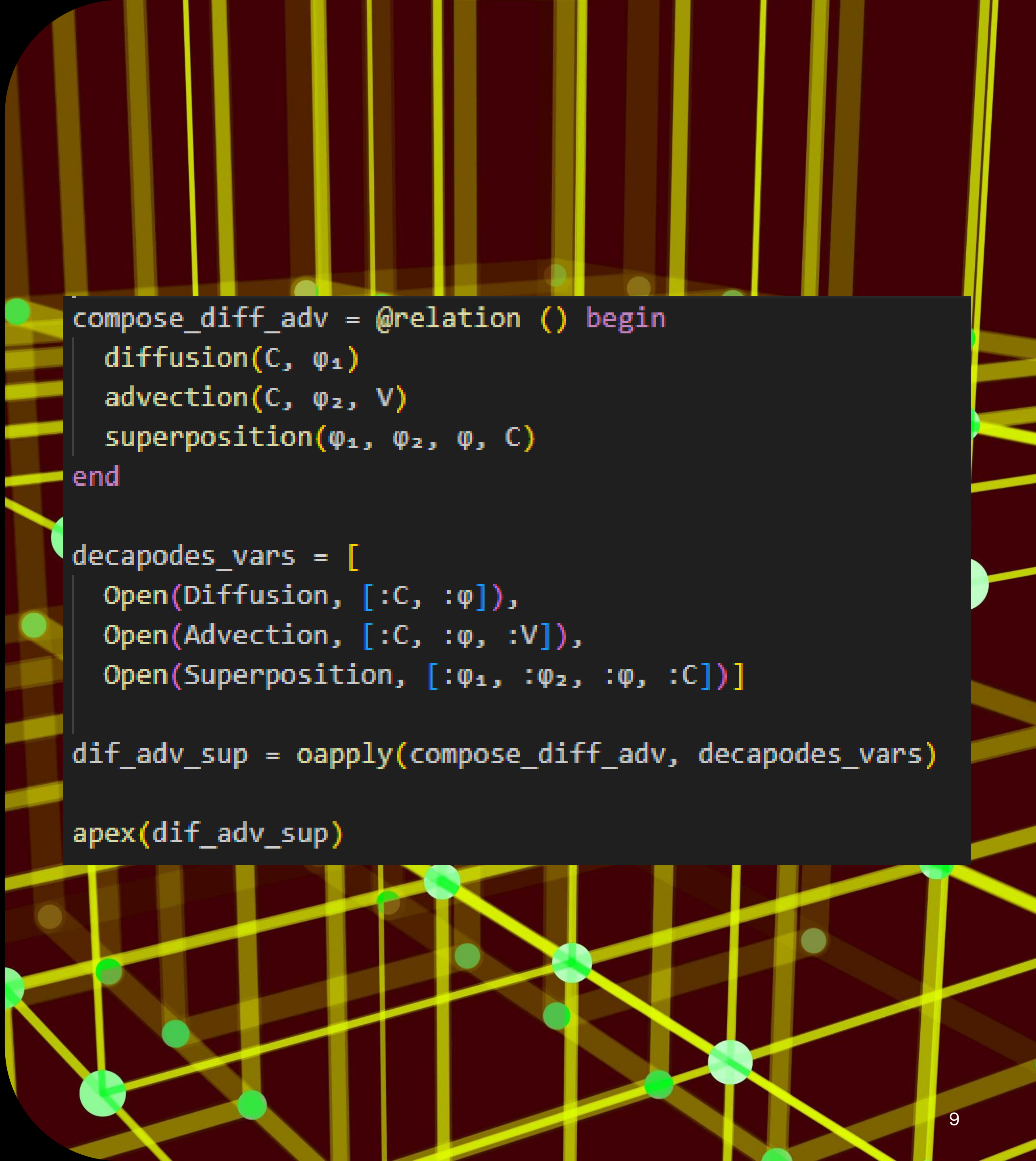
The Decapode ACSet

- Essentially a database with the schema shown here
- Represents all the physics in a dataflow diagram which we can traverse like a DAG
- This allows for easy querying and modification of the data held
- Also allows for easy viewing of the overall physics (to_graphviz)



Composition of Decapodes

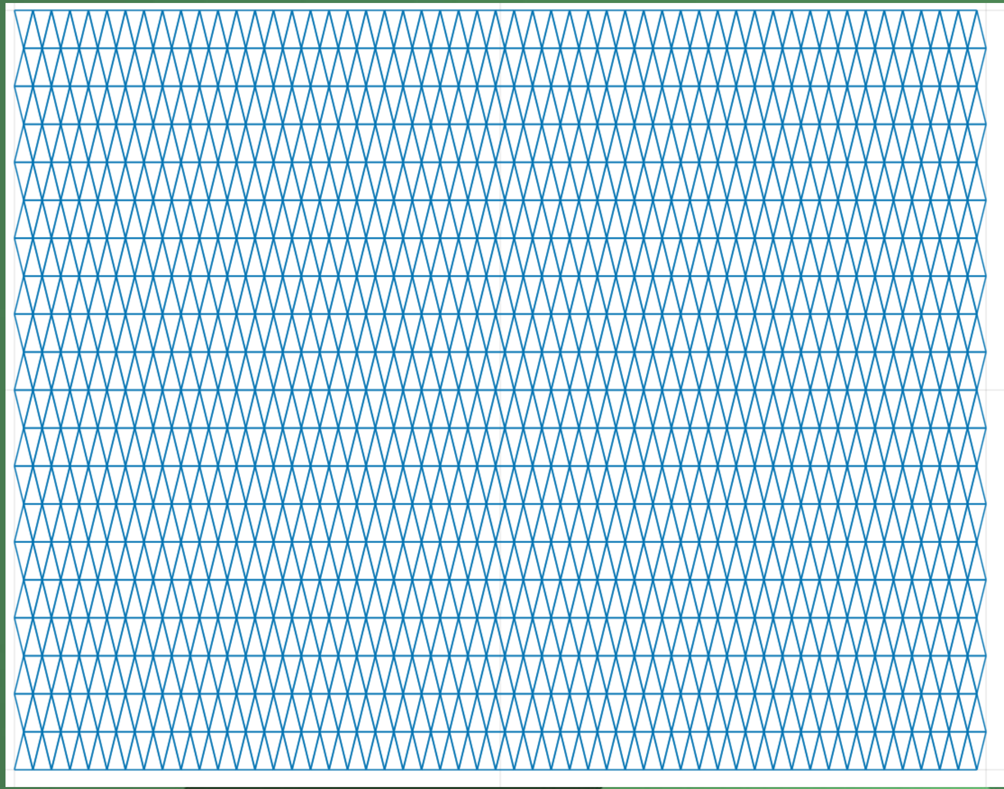
- Composition is used for combining multiple physics one one larger physical system
- The user chooses variables in each Decapode and gives them a global name
- Unselected variables are renamed to avoid naming collisions
- Use user-provided names for glued variables

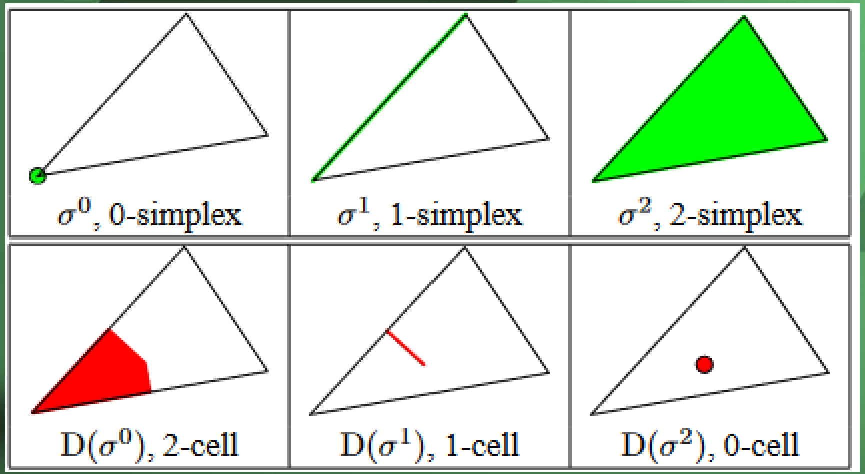


```julia
compose_diff_adv = @relation () begin
  diffusion(C, φ₁)
  advection(C, φ₂, V)
  superposition(φ₁, φ₂, φ, C)
end

decapodes_vars = [
  Open(Diffusion, [:C, :φ]),
  Open(Advection, [:C, :φ, :V]),
  Open(Superposition, [:φ₁, :φ₂, :φ, :C])]

dif_adv_sup = oapply(compose_diff_adv, decapodes_vars)

apex(dif_adv_sup)
```

Loading the Mesh

- Decapodes package already comes preloaded with certain meshes (loadmesh)
- Users can create .obj files and load in their own meshes.
- The subdivided dual of the mesh is then generated and this will be what is used for the simulation





Giving the Math (generate)

- All the Decapode ACSet stores is a table of symbols
- We need the user to define any custom functions
- Done in the generate function, using the MLStyle match on the operator's name

```julia
# Define how operations map to Julia functions.
hodge = GeometricHodge()
Δ₀ = δ(1, sd, hodge=hodge) * d(0, sd)
function generate(sd, my_symbol; hodge=GeometricHodge())
  op = @match my_symbol begin
    # The Laplacian operator on 0-Forms is the codifferential of
    # the exterior derivative. i.e. dδ
    :Δ₀ => x -> Δ₀ * x
    :.* => (x,y) -> x .* y
    x => error("Unmatched operator $my_symbol")
  end
  return (args...) -> op(args...)
end
```

Creating the Code (evalsim,
or eval(gensim))

- Takes a Decapode and creates a function call for a time step in the simulation
- Runs the provided Decapode through multiple steps for ease of compilation, code generation and code optimization
- Process is hidden away from the user

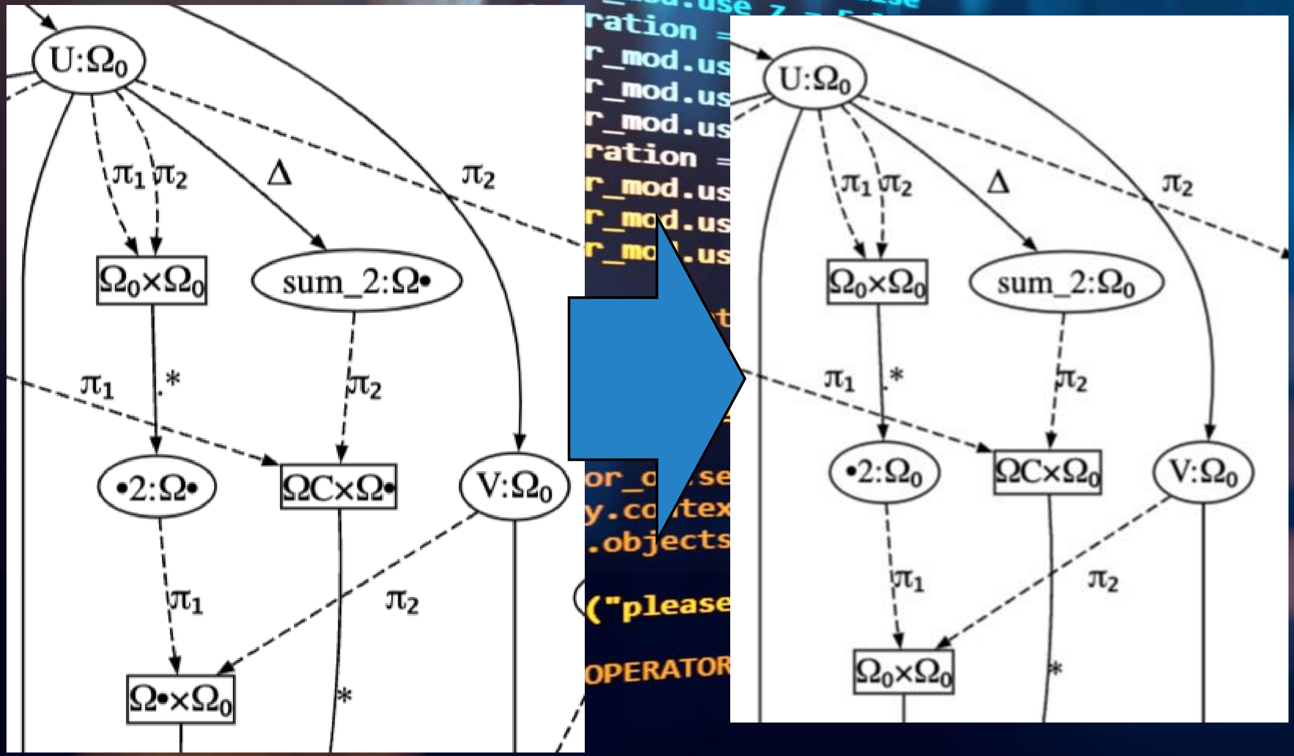
Initial Compiler Steps

- Checks all user provided types are Decapodes recognized types
- Expands out composition operators to make optimization easier
- Gets and saves user provided values and functions (get_vars_code)
- Sets up partial time steps (set_tanvars_code)

```
quote
    #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:427 =#
    function simulate(mesh, operators, hodge = GeometricHodge())
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:427 =#
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:428 =#
        begin
            #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:155 =#
            (M_Δ₀, Δ₀) = default_dec_matrix_generate(mesh, :Δ₀, hodge)
        end
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:429 =#
        begin
            #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:214 =#
            var"•6" = Vector{Float64}(undef, nparts(mesh, :V))
            sum_2 = Vector{Float64}(undef, nparts(mesh, :V))
            var"•2" = Vector{Float64}(undef, nparts(mesh, :V))
            U2V = Vector{Float64}(undef, nparts(mesh, :V))
            var"•4" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•5" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•8" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•7" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•1" = Vector{Float64}(undef, nparts(mesh, :V))
            sum_1 = Vector{Float64}(undef, nparts(mesh, :V))
            V̇ = Vector{Float64}(undef, nparts(mesh, :V))
            var"•3" = Vector{Float64}(undef, nparts(mesh, :V))
            U̇ = Vector{Float64}(undef, nparts(mesh, :V))
        end
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:430 =#
        f(du, u, p, t) = begin
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:430 =#
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:431 =#
                begin
                    #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:236 =#
                    U = (findnode(u, :U)).values
                    V = (findnode(u, :V)).values
                    α = p.α
                    F = p.F(t)
                    var"1" = 1.0
                    var"4.4" = 4.4
                    var"3.4" = 3.4
                end
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:432 =#
                mul!(var"•6", M_Δ₀, U)
                mul!(sum_2, M_Δ₀, U)
                var"•2" .= U .* U
                U2V .= var"•2" .* V
                var"•4" .= var"4.4" .* U
                var"•5" .= α .* var"•6"
                var"•8" .= var"3.4" .* U
                var"•7" .= var"•8" .- U2V
                var"•1" .= α .* sum_2
                sum_1 .= (.+)(var"1", U2V)
                V̇ .= (.+)(var"•7", var"•1")
                var"•3" .= sum_1 .- var"•4"
                U̇ .= (.+)(var"•3", var"•5", F)
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:433 =#
                (findnode(du, :U)).values .= U̇
                (findnode(du, :V)).values .= V̇
            end
    end
end
```

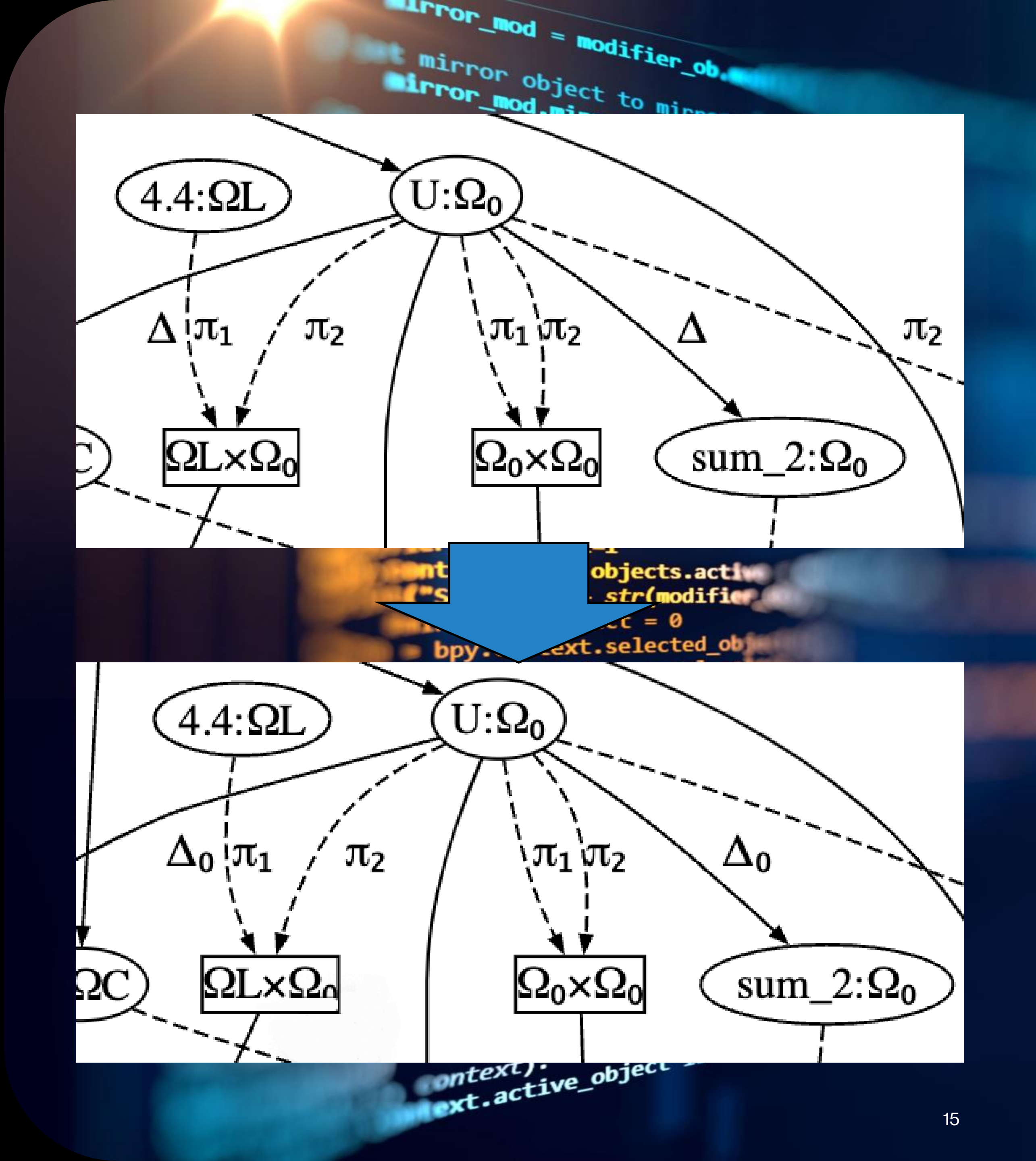
Compiler Type Inference (infer_types!)

- Compiler sets all Constants and Parameters to be inferred.
- Then infer_types! uses a rule book to attempt to determine the types of inferable variables
- Rules are based on the specific functions and what types of input/output they allow
- Knowing types ahead of time allows for optimization
- Also allows the user the freedom to not have to type every single variable



Compiler Function Overloading (resolve_overloads!)

- Some functions the user may give are generic (d, dual_d, ⋆…)
- These can not be matched to a specific function since these symbols are really classes of functions
- Functions in these classes are different based on their input types.
- But if we know the types, we know the functions
- This allows the user to not have to exactly specify every operator



Simulate Function Creation (compile*)

- Rest of the simulate function is put together
- Topological sort through the Decapode ACSet to write out the list of computations (compile)
- Function symbols are told where to get their computational implementations (compile_env)
- Any typed variables are pre-allocated (compile_var)
- This pre-allocation allows the use of in-place operations which improves the memory footprint
- Entire quote block is then returned and evaluated to a real function

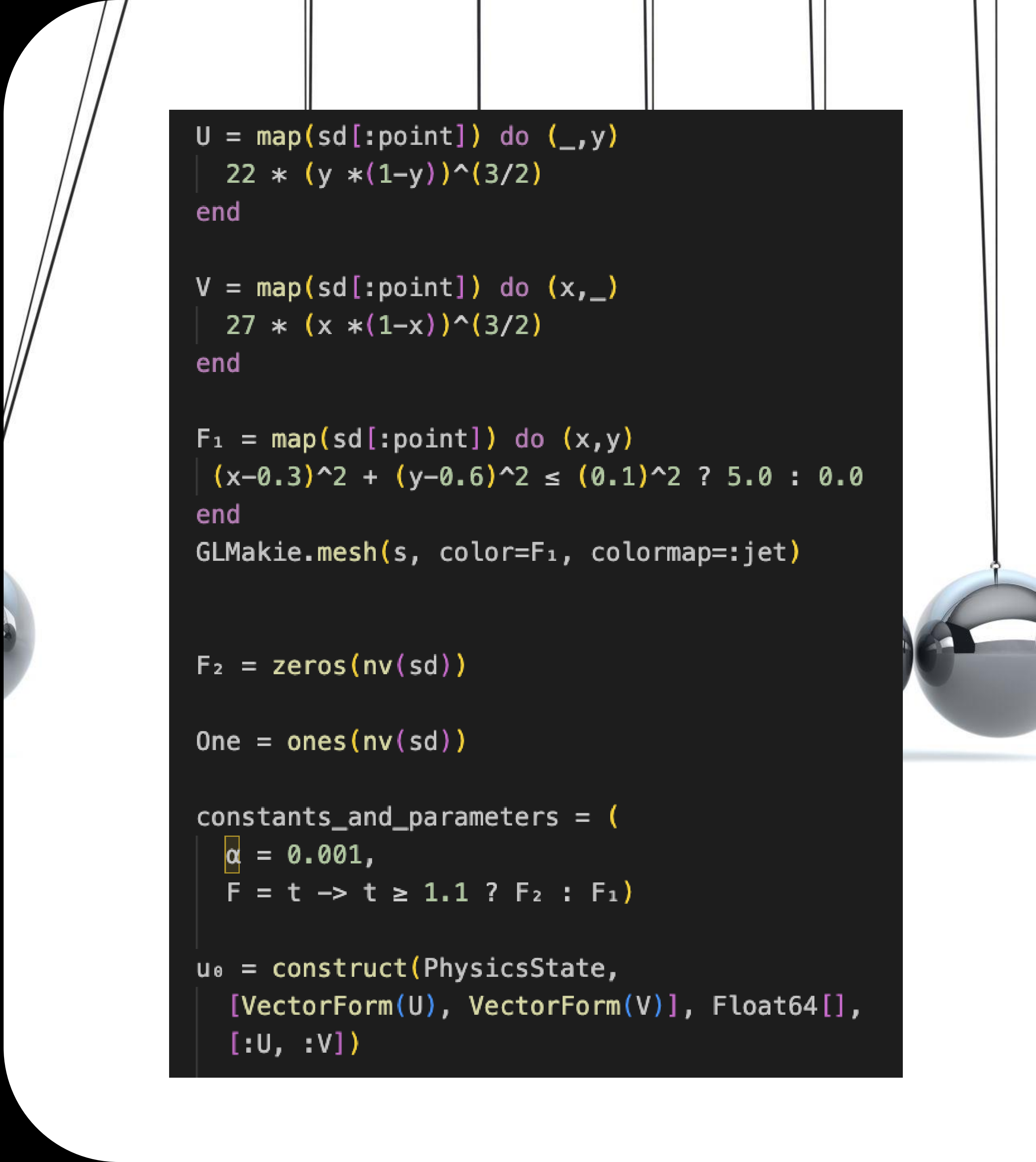
```
quote
    #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:427 =#
    function simulate(mesh, operators, hodge = GeometricHodge())
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:427 =#
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:428 =#
        begin
            #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:155 =#
            (M_Δ₀, Δ₀) = default_dec_matrix_generate(mesh, :Δ₀, hodge)
        end
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:429 =#
        begin
            #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:214 =#
            var"•6" = Vector{Float64}(undef, nparts(mesh, :V))
            sum_2 = Vector{Float64}(undef, nparts(mesh, :V))
            var"•2" = Vector{Float64}(undef, nparts(mesh, :V))
            U2V = Vector{Float64}(undef, nparts(mesh, :V))
            var"•4" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•5" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•8" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•7" = Vector{Float64}(undef, nparts(mesh, :V))
            var"•1" = Vector{Float64}(undef, nparts(mesh, :V))
            sum_1 = Vector{Float64}(undef, nparts(mesh, :V))
            V̇ = Vector{Float64}(undef, nparts(mesh, :V))
            var"•3" = Vector{Float64}(undef, nparts(mesh, :V))
            U̇ = Vector{Float64}(undef, nparts(mesh, :V))
        end
        #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:430 =#
        f(du, u, p, t) = begin
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:430 =#
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:431 =#
                begin
                    #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:236 =#
                    U = (findnode(u, :U)).values
                    V = (findnode(u, :V)).values
                    α = p.α
                    F = p.F(t)
                    var"1" = 1.0
                    var"4.4" = 4.4
                    var"3.4" = 3.4
                end
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:432 =#
                mul!(var"•6", M_Δ₀, U)
                mul!(sum_2, M_Δ₀, U)
                var"•2" .= U .* U
                U2V .= var"•2" .* V
                var"•4" .= var"4.4" .* U
                var"•5" .= α .* var"•6"
                var"•8" .= var"3.4" .* U
                var"•7" .= var"•8" .- U2V
                var"•1" .= α .* sum_2
                sum_1 .= (.+)(var"1", U2V)
                V̇ .= (.+)(var"•7", var"•1")
                var"•3" .= sum_1 .- var"•4"
                U̇ .= (.+)(var"•3", var"•5", F)
                #= /Users/grauta/Documents/GitHub/Decapodes.jl/src/simulation.jl:433 =#
                (findnode(du, :U)).values .= U̇
                (findnode(du, :V)).values .= V̇
            end
    end
end
```

Simulate

- The generate function along with the mesh is passed to the simulate function created by evalsim
- Creates the actual function implementation the ODE solver will use
- Lots of pre-allocation occurring here for functions to get their computational implementations

Initial Conditions

- Need the initial vector form information, created using a PhysicsState
- Collection of initial vector values and associated names
- Materialization of any constants and parameters as a named tuple



```julia
U = map(sd[:point]) do (_,y)
  22 * (y *(1-y))^(3/2)
end

V = map(sd[:point]) do (x,_)
  27 * (x *(1-x))^(3/2)
end

F₁ = map(sd[:point]) do (x,y)
 (x-0.3)^2 + (y-0.6)^2 ≤ (0.1)^2 ? 5.0 : 0.0
end
GLMakie.mesh(s, color=F₁, colormap=:jet)


F₂ = zeros(nv(sd))

One = ones(nv(sd))

constants_and_parameters = (
  α = 0.001,
  F = t -> t ≥ 1.1 ? F₂ : F₁)

u₀ = construct(PhysicsState,
  [VectorForm(U), VectorForm(V)], Float64[],
  [:U, :V])
```

Solve the ODE Problem

- Decapodes uses the OrdinaryDiffEq.jl package to solve ODE problems
- Create an ODEProblem by passing the created function from simulate, the PhysicsState, a time range and the constants and parameters tuple
- Run solve using the ODEProblem and a solver (usually Tsit5)
- Once solved, the returned solution can be checked for statistics (number of steps taken, saved time steps…)

```
tₑ = 11.5

@info("Precompiling Solver")
prob = ODEProblem(fₘ, u₀, (0, 1e-4), constants_and_parameters)
soln = solve(prob, Tsit5())
soln.retcode != :Unstable || error("Solver was not stable")
@info("Solving")
prob = ODEProblem(fₘ, u₀, (0, tₑ), constants_and_parameters)
soln = solve(prob, Tsit5())
@info("Done")
```

```
julia> soln.t
104-element Vector{Float64}:
  0.0
  0.0018406977555249192
  0.013796676061461724
  0.040558444159737306
  0.08124163068340635
  0.12721789253162627
  0.18082091833646546
  0.23625485397765014
  0.29476252278720744
  0.3648421803983336
  0.4469570533214677
  0.549513914124302
  0.6779819145173636
  0.8435672880891704
  1.044920032606646
  1.1541984861431622
  ⋮
```

```
julia> soln.stats
DiffEqBase.Stats
Number of function 1 evaluations:
          663
Number of function 2 evaluations:
          0
Number of W matrix evaluations:
          0
Number of linear solves:
          0
Number of Jacobians created:
          0
Number of nonlinear solver iterations:
          0
Number of nonlinear solver convergence fa
ilures:   0
Number of rootfind condition calls:
          0
Number of accepted steps:
          103
Number of rejected steps:
          7
```

Some GLMakie Magic

And you’re done!