Getting Started with FunctorFlow.jl

Building your first categorical diagram

Simon Frost

Table of contents

Introduction

FunctorFlow.jl is a categorical DSL and executable intermediate representation for building diagrammatic AI systems in Julia, built on top of the AlgebraicJulia ecosystem (Catlab.jl, ACSets.jl). It is a port of the Python FunctorFlow package by Sridhar Mahadevan. The library provides a principled way to compose computational operations using the language of category theory — objects represent typed interfaces, morphisms represent transformations, and Kan extensions provide universal aggregation and completion patterns.

The API uses mathematical unicode notation: Σ for left Kan extensions (universal aggregation) and Δ for right Kan extensions (universal completion), with a `@functorflow` macro DSL inspired by algebraic Julia packages like DiagrammaticEquations.jl.

Setup

```
using FunctorFlow
```

Core Concepts

FunctorFlow is built on three core abstractions:

- Objects (`FFObject`): Typed interfaces that represent data flowing through a diagram. Each object has a name, a kind (e.g. `:input`, `:output`, `:hidden_state`), and an optional shape.
- Morphisms (`Morphism`): Typed arrows from a source object to a target object. Each morphism can be bound to a concrete implementation — any Julia callable.
- Diagrams (`Diagram`): Mutable containers that hold objects, morphisms, Kan extensions, and obstruction losses. Diagrams can be compiled and executed.

Let's create each of these individually:

```
# A typed interface with kind and shape metadata
x = FFObject(:X; kind=:input, shape="(n,)", description="Input vector")
println(x)
```

```
X::input [(n,)]
```

```
# A typed transformation from X to Y
f = Morphism(:f, :X, :Y; description="Linear transform")
println(f)
```

```
f: X → Y
```

```
# A mutable diagram container
D = Diagram(:MyFirstDiagram)
println(D)
```

```
Diagram :MyFirstDiagram ⟨0 objects, 0 morphisms, 0 Kan, 0 losses⟩
```

Your First Diagram

Let's build a simple diagram with two objects and one morphism, then compile and execute it.

```
D = Diagram(:DoubleDiagram)

# Add typed objects
add_object!(D, :X; kind=:input, description="Input value")
add_object!(D, :Y; kind=:output, description="Doubled value")

# Add a morphism with an implementation bound inline
add_morphism!(D, :double, :X, :Y;
              implementation=x -> x * 2,
              description="Doubles the input")

# Compile the diagram
compiled = compile_to_callable(D)

# Execute with concrete inputs
result = FunctorFlow.run(compiled, Dict(:X => 5))
println("double(5) = ", result.values[:double])
```

```
double(5) = 10
```

We can also bind implementations after construction:

```
D2 = Diagram(:SquareDiagram)
add_object!(D2, :A; kind=:input)
add_object!(D2, :B; kind=:output)
add_morphism!(D2, :square, :A, :B)

# Bind later
bind_morphism!(D2, :square, x -> x^2)

compiled2 = compile_to_callable(D2)
result2 = FunctorFlow.run(compiled2, Dict(:A => 7))
println("square(7) = ", result2.values[:square])
```

```
square(7) = 49
```

Compositions

Morphisms can be composed sequentially using `compose!()`. FunctorFlow validates that each morphism's target matches the next morphism's source, ensuring type safety in the composition chain.

```
D3 = Diagram(:Pipeline)
add_object!(D3, :Raw; kind=:input, description="Raw text input")
add_object!(D3, :Cleaned; kind=:intermediate, description="Cleaned text")
add_object!(D3, :Embedded; kind=:output, description="Numeric embedding")

add_morphism!(D3, :clean, :Raw, :Cleaned;
              implementation=x -> lowercase(strip(x)),
              description="Normalize whitespace and case")
add_morphism!(D3, :embed, :Cleaned, :Embedded;
              implementation=x -> length(x),
              description="Simple length-based embedding")

# Compose the two morphisms into a named pipeline
compose!(D3, :clean, :embed; name=:pipeline)

compiled3 = compile_to_callable(D3)
result3 = FunctorFlow.run(compiled3, Dict(:Raw => "  HELLO World  "))
println("Cleaned: ", result3.values[:clean])
println("Embedded: ", result3.values[:embed])
println("Pipeline: ", result3.values[:pipeline])
```

```
Cleaned: hello world
Embedded: 11
Pipeline: 11
```

When a composition is created, all intermediate morphisms still execute individually (their results stored under both the morphism name and target object name). The composition provides a named reference to the sequential chain and validates type compatibility at construction time.

Obstruction Loss

FunctorFlow can also measure how much two diagram paths fail to commute. An `ObstructionLoss` compares outputs of different paths and computes a scalar loss — the foundation of Diagrammatic Backpropagation.

```
D_obs = Diagram(:CommutativityCheck)
add_object!(D_obs, :S; kind=:state)
add_morphism!(D_obs, :f, :S, :S; implementation=x -> x + 1.0)
add_morphism!(D_obs, :g, :S, :S; implementation=x -> x * 2.0)
```

```
compose!(D_obs, :f, :g; name=:fg)
compose!(D_obs, :g, :f; name=:gf)

add_obstruction_loss!(D_obs, :comm_loss; paths=[(:fg, :gf)], comparator=:l2)

compiled_obs = compile_to_callable(D_obs)
result_obs = FunctorFlow.run(compiled_obs, Dict(:S ⇒ 3.0))
println("f∘g(3) = ", result_obs.values[:fg], ", g∘f(3) = ", result_obs.values[:gf])
println("Commutativity loss: ", result_obs.losses[:comm_loss])
```

```
f∘g(3) = 8.0, g∘f(3) = 7.0
Commutativity loss: 1.0
```

The loss is zero when the paths produce identical results (i.e. the diagram commutes).

Adding Kan Extensions

Kan extensions are the crown jewel of FunctorFlow. A left Kan extension (denoted Σ) performs universal aggregation — it pushes forward values along a relation and reduces them. This single pattern subsumes attention, pooling, and message passing.

```
D4 = Diagram(:AggregationDemo)
add_object!(D4, :Values; kind=:messages, description="Node values")
add_object!(D4, :Incidence; kind=:relation, description="Edge incidence")
add_object!(D4, :Aggregated; kind=:output)

# Use the Σ operator (unicode left Kan extension)
Σ(D4, :Values; along=:Incidence, target=:Aggregated, reducer=:sum, name=:aggregate)

compiled4 = compile_to_callable(D4)

# Values is a dict mapping source keys to values
# Incidence maps target keys to their source neighborhoods
result4 = FunctorFlow.run(compiled4, Dict(
    :Values ⇒ Dict(:a ⇒ 1, :b ⇒ 2, :c ⇒ 3),
    :Incidence ⇒ Dict(:x ⇒ [:a, :b], :y ⇒ [:b, :c])
))

println("Aggregated: ", result4.values[:aggregate])
```

```
Aggregated: Dict(:y => 5, :x => 3)
```

The `:sum` reducer sums values within each neighborhood. Target `:x` gets `1 + 2 = 3` and target `:y` gets `2 + 3 = 5`.

The dual concept is the right Kan extension (denoted Δ), which performs universal completion — filling in missing values from compatible neighbors:

```
D5 = Diagram(:CompletionDemo)
add_object!(D5, :Partial; kind=:partial)
add_object!(D5, :Compat; kind=:relation)

# Use the Δ operator (unicode right Kan extension)
Δ(D5, :Partial; along=:Compat, name=:complete)

compiled5 = compile_to_callable(D5)
result5 = FunctorFlow.run(compiled5, Dict(
    :Partial ⇒ Dict(:a ⇒ nothing, :b ⇒ 42, :c ⇒ nothing),
    :Compat ⇒ Dict(:a ⇒ [:b, :c], :c ⇒ [:b])
))
println("Completed: ", result5.values[:complete])
```

```
Completed: Dict{Any, Any}(:a => 42, :c => 42)
```

Here, `:a` was `nothing` but got filled with `42` from its compatible neighbor `:b`. Together, left and right Kan extensions form the predict-and-repair duality central to FunctorFlow.

Inspecting Diagrams

FunctorFlow provides several ways to inspect diagram structure.

```
# JSON serialization
json_str = to_json(D4)
println(json_str[1:min(200, length(json_str))], "...")
```

```
{"operations":[{"kind":"kanextension","name":"aggregate","source":"Values","direction":"left",
```

```
# Dictionary representation
d = as_dict(to_ir(D4))
println("Objects: ", collect(keys(d["objects"])))
println("Operations: ", collect(keys(d["operations"])))
```

```
Objects: [1, 2, 3]
Operations: [1]
```

```
# IR representation
ir = to_ir(D4)
println(ir)
```

```
DiagramIR :AggregationDemo ⟨3 objects, 1 operations, 0 losses⟩
```

Catlab Integration

FunctorFlow.jl integrates with the AlgebraicJulia ecosystem. You can convert any diagram to an ACSet (Attributed C-Set) for use with Catlab.jl's categorical algebra tools:

```
# Convert diagram to ACSet representation
acs = to_acset(D4)
println("ACSet nodes: ", nparts(acs, :Node))
println("ACSet Kan extensions: ", nparts(acs, :Kan))
```

```
ACSet nodes: 3
ACSet Kan extensions: 1
```

```
# Convert to a Catlab Presentation for symbolic reasoning
pres = to_presentation(D4)
println("Generators: ", length(FunctorFlow.Catlab.generators(pres)))
```

```
Generators: 3
```

Next Steps

Now that you understand the basics, explore the other vignettes:

- DSL Macros — build diagrams with the concise `@functorflow` macro syntax
- Kan Extensions — deep dive into Σ (aggregation) and Δ (completion) patterns
- Block Library — use pre-built architectural patterns (KET, DB Square, BASKET, ROCKET, etc.)
- Diagram Composition — compose sub-diagrams using ports and adapters