DSL Macros

Building diagrams with @functorflow and unicode operators

Simon Frost

Table of contents

Introduction

Julia's macro system allows FunctorFlow to offer a concise, declarative DSL for building categorical diagrams. The `@functorflow` macro uses mathematical notation inspired by the AlgebraicJulia ecosystem — type annotations for objects, arrows (→) for morphisms, and unicode operators Σ (left Kan) and Δ (right Kan) for extensions. The legacy `@diagram` macro with `@object`/`@morphism` sub-macros is still supported for compatibility, but `@functorflow` is the recommended style.

Setup

```
using FunctorFlow
```

The @functorflow Macro

The `@functorflow` macro takes a name and a `begin...end` block using a clean, mathematical syntax:

- Objects: `Name :: kind` (type annotation style)
- Morphisms: `name = Source → Target` (arrow notation)
- Left Kan: `name = Σ(:source; along=:relation, ...)` (sigma for aggregation)
- Right Kan: `name = Δ(:source; along=:relation, ...)` (delta for completion)

```
D = @functorflow SimpleTransform begin
    X :: input
    Y :: output
    transform = X → Y
end

println(D)
```

```
Diagram :SimpleTransform ⟨2 objects, 1 morphisms, 0 Kan, 0 losses⟩
```

Each type annotation declares a typed interface and each arrow declares a typed morphism. Objects referenced by morphisms that haven't been declared are automatically created as placeholders.

```
# Morphisms auto-create placeholder objects for undeclared references
D2 = @functorflow AutoCreate begin
    f = A → B
    g = B → C
end

d = as_dict(to_ir(D2))
println("Objects: ", collect(keys(d["objects"])))
println("Operations: ", collect(keys(d["operations"])))
```

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

Unicode Operators

FunctorFlow provides mathematical unicode operators that work both inside and outside the `@functorflow` macro:

Operator	Unicode	ASCII alias	Meaning
Σ	`\Sigma`	`left_kan`	Left Kan extension (aggregation)
Δ	`\Delta`	`right_kan`	Right Kan extension (completion)
□	`\to`	`→`	Arrow (morphism declaration)
□	`\cdot`	`compose`	Composition
□	`\otimes`	`product`	Product
□	`\oplus`	`coproduct`	Coproduct

Compositions

Use `@compose` inside a `@functorflow` block to chain morphisms sequentially:

```
D3 = @functorflow Pipeline begin
    Raw :: input
    Cleaned :: intermediate
    Embedded :: output

    clean = Raw → Cleaned
    embed = Cleaned → Embedded

    @compose clean embed name=:full_pipeline
end

d3 = as_dict(to_ir(D3))
println("Operations: ", collect(keys(d3["operations"])))
```

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

We can bind implementations and run the composed diagram:

```
bind_morphism!(D3, :clean, x -> lowercase(strip(x)))
bind_morphism!(D3, :embed, x -> collect(Int, x))

compiled = compile_to_callable(D3)
```

```
result = FunctorFlow.run(compiled, Dict(:Raw => "  HELLO  "))
println("Cleaned: ", result.values[:clean])
println("Embedded: ", result.values[:embed])
```

```
Cleaned: hello
Embedded: [104, 101, 108, 108, 111]
```

Kan Extensions with Σ and Δ

Inside `@functorflow`, use `Σ(...)` and `Δ(...)` for left and right Kan extensions:

```
D4 = @functorflow Aggregator begin
    Messages :: messages
    Neighbors :: relation
    Pooled :: output

    pool = Σ(:Messages; along=:Neighbors, target=:Pooled, reducer=:mean)
end

compiled4 = compile_to_callable(D4)
result4 = FunctorFlow.run(compiled4, Dict(
    :Messages => Dict(:a => 10.0, :b => 20.0, :c => 30.0),
    :Neighbors => Dict(:x => [:a, :b], :y => [:b, :c])
))
println("Pooled (mean): ", result4.values[:pool])
```

```
Pooled (mean): Dict(:y => 25.0, :x => 15.0)
```

Right Kan extensions use `Δ(...)` and default to the `:first_non_null` reducer:

```
D5 = @functorflow Completer begin
    Partial :: partial
    Compat :: relation

    repair = Δ(:Partial; along=:Compat, reducer=:first_non_null)
end

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("Repaired: ", result5.values[:repair])
```

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

Obstruction Loss

Use `@obstruction_loss` to measure non-commutativity between diagram paths:

```
D6 = @functorflow DBSquareDemo begin
    S :: state

    f = S → S
    g = S → S

    @compose f g name=:fg
    @compose g f name=:gf

    @obstruction_loss consistency paths=[(:fg, :gf)] comparator=:l2 weight=1.0
end

bind_morphism!(D6, :f, x -> x + 1)
bind_morphism!(D6, :g, x -> x * 2)

compiled6 = compile_to_callable(D6)
result6 = FunctorFlow.run(compiled6, Dict(:S => 3.0))
println("f∘g(3) = ", result6.values[:fg])
println("g∘f(3) = ", result6.values[:gf])
println("Obstruction loss: ", result6.losses[:consistency])
```

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

Ports

Ports expose semantic interfaces on a diagram for composition. Use `@port` inside `@functorflow`:

```
D7 = @functorflow PortedModel begin
    Tokens :: messages
    Neighbors :: relation
    Output :: contextualized_messages

    aggregate = Σ(:Tokens; along=:Neighbors, target=:Output, reducer=:sum)

    @port input Tokens direction=:input type=:messages
    @port relation Neighbors direction=:input type=:relation
    @port output Output direction=:output type=:contextualized_messages
end

d7 = as_dict(to_ir(D7))
println("Ports: ", collect(keys(d7["ports"])))
```

```
Ports: [1, 2, 3]
```

ACSet Schema Integration

Diagrams built with `@functorflow` can be converted to ACSet representations for use with Catlab.jl:

```
acs = to_acset(D4)
println("Nodes in ACSet: ", nparts(acs, :Node))
println("Kan extensions in ACSet: ", nparts(acs, :Kan))
```

```
Nodes in ACSet: 3
Kan extensions in ACSet: 1
```

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

```
Generators: 3
```

Comparing Three Styles

FunctorFlow offers three ways to build diagrams. Here is the same diagram in each style:

1. Builder API (imperative, fine-grained control):

```
D_api = Diagram(:KETManual)
add_object!(D_api, :Values; kind=:messages)
add_object!(D_api, :Incidence; kind=:relation)
add_object!(D_api, :Aggregated; kind=:contextualized_messages)
Σ(D_api, :Values; along=:Incidence, target=:Aggregated, reducer=:sum, name=:aggregate)
expose_port!(D_api, :input, :Values; direction=INPUT, port_type=:messages)
expose_port!(D_api, :output, :Aggregated; direction=OUTPUT, port_type=:contextualized_messages
println("API diagram: ", D_api.name, " — ", length(D_api.objects), " objects, ", length(D_api.
```

```
API diagram: KETManual — 3 objects, 1 operations
```

2. @functorflow macro (mathematical, recommended):

```
D_ff = @functorflow KETFunctorFlow begin
    Values :: messages
    Incidence :: relation
    Aggregated :: contextualized_messages

    aggregate = Σ(:Values; along=:Incidence, target=:Aggregated, reducer=:sum)

    @port input Values direction=:input type=:messages
    @port output Aggregated direction=:output type=:contextualized_messages
end
println("@functorflow diagram: ", D_ff.name, " — ", length(D_ff.objects), " objects, ", length
```

```
@functorflow diagram: KETFunctorFlow — 3 objects, 1 operations
```

3. Legacy @diagram macro (still supported):

```
D_dsl = @diagram KETMacro begin
    @object Values kind=:messages
    @object Incidence kind=:relation
    @object Aggregated kind=:contextualized_messages

    @left_kan aggregate source=Values along=Incidence target=Aggregated reducer=:sum
```

```
    @port input Values direction=:input type=:messages
    @port output Aggregated direction=:output type=:contextualized_messages
end
println("@diagram (legacy): ", D_dsl.name, " — ", length(D_dsl.objects), " objects, ", length(
```

```
@diagram (legacy): KETMacro — 3 objects, 1 operations
```

All three produce identical diagram structures.

When to Use Which

Scenario	Recommended Style
Quick prototyping	`@functorflow` macro
Mathematical documentation	`@functorflow` macro (unicode operators)
Dynamic construction (loops, conditionals)	Builder API with Σ/Δ operators
Block library integration	Builder API (blocks return `Diagram` objects)
Catlab interop / ACSet conversion	Any style (all produce the same `Diagram`)
Complex wiring with adapters	Builder API (finer control)

The `@functorflow` macro is syntactic sugar over the builder API. Use whichever is clearest for your use case — you can always mix styles by starting with `@functorflow` and then calling builder methods on the resulting `Diagram` object.