Block Library

Pre-built architectural patterns

Simon Frost

Table of contents

Introduction . . . 1
Setup . . . 1
KET Block . . . 2
DB Square . . . 2
GT Neighborhood . . . 3
Completion Block . . . 3
BASKET Workflow . . . 4
ROCKET Repair . . . 4
Structured LM Duality . . . 5
Democritus Gluing . . . 5
BASKET-ROCKET Pipeline . . . 6
The Macro Registry . . . 7

Introduction

FunctorFlow provides a library of pre-built block builders — parameterized diagram constructors that encode common architectural patterns from categorical AI. Each block returns a `Diagram` with named objects, operations, and ports. You can customize them via configuration structs, bind concrete implementations, and compose them into larger pipelines.

Setup

```
using FunctorFlow
```

KET Block

The KET (Kan Extension Template) block is the most fundamental pattern. It performs left-Kan aggregation over an incidence relation — the universal building block for attention, pooling, and message passing.

```
D_ket = ket_block(; config=KETBlockConfig(reducer=:sum))
println(D_ket)
```

```
Diagram :KETBlock ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled = compile_to_callable(D_ket)
result = FunctorFlow.run(compiled, Dict(
    :Values ⇒ Dict(:a ⇒ 1.0, :b ⇒ 2.0, :c ⇒ 3.0),
    :Incidence ⇒ Dict(:x ⇒ [:a, :b], :y ⇒ [:b, :c])
))
println("KET output: ", result.values[:aggregate])
```

```
KET output: Dict(:y => 5.0, :x => 3.0)
```

DB Square

The DB Square (Diagrammatic Backpropagation Square) measures the obstruction to commutativity of two morphisms. It computes `f∘g` and `g∘f` and reports the distance as a loss.

```
D_db = db_square(;
    first_impl=x -> x + 1.0,
    second_impl=x -> x * 2.0
)
println(D_db)
```

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

```
compiled_db = compile_to_callable(D_db)
result_db = FunctorFlow.run(compiled_db, Dict(:State ⇒ 5.0))
println("f∘g path: ", result_db.values[:p1])
println("g∘f path: ", result_db.values[:p2])
println("Obstruction loss: ", result_db.losses[:obstruction])
```

```
f∘g path: 12.0
g∘f path: 11.0
Obstruction loss: 1.0
```

The loss is zero only when `f` and `g` commute.

GT Neighborhood

The GT Neighborhood (Graph Transformer Neighborhood) block first lifts tokens to edge messages via a morphism, then aggregates them with a left Kan extension. This is the standard two-step pattern in graph neural networks.

```
D_gt = gt_neighborhood_block()
println(D_gt)
```

```
Diagram :GTNeighborhood ⟨4 objects, 1 morphisms, 1 Kan, 0 losses⟩
```

```
bind_morphism!(D_gt, :lift, x -> Dict(k => v * 10 for (k, v) in x))

compiled_gt = compile_to_callable(D_gt)
result_gt = FunctorFlow.run(compiled_gt, Dict(
    :Tokens => Dict(:a => 1, :b => 2, :c => 3),
    :Incidence => Dict(:x => [:a, :b], :y => [:c])
))
println("Lifted then aggregated: ", result_gt.values[:aggregate])
```

```
Lifted then aggregated: Dict(:y => 30, :x => 30)
```

Completion Block

The Completion Block uses a right Kan extension for universal completion — filling in partial or missing data from compatible neighbors.

```
D_comp = completion_block()
println(D_comp)
```

```
Diagram :CompletionBlock ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled_comp = compile_to_callable(D_comp)
result_comp = FunctorFlow.run(compiled_comp, Dict(
    :PartialValues => Dict(:a => nothing, :b => 42, :c => nothing, :d => 7),
    :Compatibility => Dict(:a => [:b, :c], :c => [:d])
))
println("Completed: ", result_comp.values[:complete])
```

```
Completed: Dict{Any, Any}(:a => 42, :c => 7)
```

BASKET Workflow

The BASKET (Bounded Aggregation via Sheaf-theoretic Kan Extension Templates) workflow block composes local plan fragments into a composed plan using left Kan with a `:concat` reducer.

```
D_basket = basket_workflow_block()
println(D_basket)
```

```
Diagram :BASKETWorkflow ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled_basket = compile_to_callable(D_basket)
result_basket = FunctorFlow.run(compiled_basket, Dict(
    :PlanFragments => Dict(:step1 => "fetch data", :step2 => "clean data", :step3 => "train mo
    :WorkflowRelation => Dict(:phase1 => [:step1, :step2], :phase2 => [:step3])
))
println("Composed plan: ", result_basket.values[:compose_fragments])
```

```
Composed plan: Dict{Any, Any}(:phase1 => "fetch dataclean data", :phase2 => "train model")
```

ROCKET Repair

The ROCKET (Robust Obstruction-Corrected Kan Extension Transform) repair block uses a right Kan extension to repair candidates using edit neighborhoods.

```
D_rocket = rocket_repair_block()
println(D_rocket)
```

```
Diagram :ROCKETRepair ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled_rocket = compile_to_callable(D_rocket)
result_rocket = FunctorFlow.run(compiled_rocket, Dict(
    :Candidates => Dict(:a => nothing, :b => "valid plan", :c => nothing),
    :EditNeighborhood => Dict(:a => [:b], :c => [:b])
))
println("Repaired: ", result_rocket.values[:repair])
```

```
Repaired: Dict{Any, Any}(:a => "valid plan", :c => "valid plan")
```

Structured LM Duality

The Structured LM Duality block runs parallel left-Kan (prediction) and right-Kan (completion/repair) branches from a shared input. This captures the predict-then-repair duality central to structured language modeling.

```
D_lm = structured_lm_duality()
println(D_lm)
```

```
Diagram :StructuredLMDuality ⟨5 objects, 0 morphisms, 2 Kan, 0 losses⟩
```

```
d_lm = as_dict(to_ir(D_lm))
println("Objects: ", collect(keys(d_lm["objects"])))
println("Operations: ", collect(keys(d_lm["operations"])))
println("Ports: ", collect(keys(d_lm["ports"])))
```

```
Objects: [1, 2, 3, 4, 5]
Operations: [1, 2]
Ports: [1, 2, 3]
```

Democritus Gluing

The Democritus Gluing block implements sheaf-theoretic local-to-global assembly. It uses a right Kan with `:set_union` reducer to glue local causal claims over overlap regions into a global relational state.

```
D_demo = democritus_gluing_block()
println(D_demo)
```

```
Diagram :DemocritusGluing ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled_demo = compile_to_callable(D_demo)
result_demo = FunctorFlow.run(compiled_demo, Dict(
    :LocalClaims => Dict(:region1 => Set(["A causes B"]), :region2 => Set(["B causes C"]), :re
    :OverlapRegion => Dict(:global => [:region1, :region2, :region3])
))
println("Glued global state: ", result_demo.values[:glue])
```

```
Glued global state: Dict{Any, Any}(:global => Set(Any["A causes B", "B causes C", "C causes D"
```

BASKET-ROCKET Pipeline

The BASKET-ROCKET Pipeline composes two stages: a BASKET draft phase (left Kan with `:concat`) followed by a ROCKET repair phase (right Kan with `:first_non_null`). This is a complete draft-then-repair workflow.

```
D_br = basket_rocket_pipeline(;
    config=BasketRocketPipelineConfig(
        rocket_config=ROCKETRepairConfig(reducer=:concat)
    )
)
println(D_br)
```

```
Diagram :BasketRocketPipeline ⟨5 objects, 0 morphisms, 2 Kan, 1 losses⟩
```

```
result_br = FunctorFlow.run(D_br, Dict(
    D_br.ports[:input].ref => Dict(
        :collect => "collect data\n",
        :validate => "validate schema\n",
        :train => "train model\n",
        :deploy => "deploy service\n",
        :monitor => "monitor drift\n"
    ),
    D_br.ports[:draft_relation].ref => Dict(
        :base => [:collect, :train, :deploy],
        :guardrails => [:validate],
        :ops => [:monitor]
    ),
    D_br.ports[:repair_relation].ref => Dict(
```

```
        :execution_ready ⇒ [:base, :guardrails, :ops]
    )
))

println("Draft candidates: ", result_br.values[D_br.ports[:draft].ref])
println("Repaired plan: ", result_br.values[D_br.ports[:output].ref])
println("Consistency loss: ", result_br.losses[:draft_repair_consistency])
```

```
Draft candidates: Dict{Any, Any}(:guardrails => "validate schema\n", :base => "collect data\nt
Repaired plan: Dict{Any, Any}(:execution_ready => "collect data\ntrain model\ndeploy service\n
Consistency loss: 0.2857142857142857
```

The Macro Registry

All blocks are registered in the `MACRO_LIBRARY` and can also be built by name using `build_macro`:

```
println("Available macros: ", collect(keys(MACRO_LIBRARY)))
```

```
Available macros: [:democritus_assembly, :ket, :topocoend, :basket_workflow, :db_square, :kan_
```

```
# Build a block by name
D_from_registry = build_macro(:ket; reducer=:mean)
println(D_from_registry)
```

```
Diagram :KETBlock ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

```
compiled_reg = compile_to_callable(D_from_registry)
result_reg = FunctorFlow.run(compiled_reg, Dict(
    :Values ⇒ Dict(:a ⇒ 10.0, :b ⇒ 20.0),
    :Incidence ⇒ Dict(:x ⇒ [:a, :b])
))
println("Built from registry (mean): ", result_reg.values[:aggregate])
```

```
Built from registry (mean): Dict(:x => 15.0)
```

The macro registry provides a uniform interface for programmatic block construction — useful for meta-learning, architecture search, and pipeline generators.