Neurosymbolic Pipelines

BASKET ROCKET, Democritus assembly, TopoCoend, horn filling, and behavioral quotienting

Simon Frost

Table of contents

Introduction

This vignette packages five `papers/md/catagi.md`-style ideas into first-class `FunctorFlow` blocks:

- BASKET ROCKET for draft-then-repair planning
- Democritus assembly for local-to-global causal gluing with regrounding
- TopoCoend for learned neighborhood construction before Kan aggregation
- Horn filling for higher-order simplicial consistency losses
- Bisimulation quotienting for collapsing behaviorally equivalent trajectories

The common pattern is categorical: assemble local evidence, map it through a relation, and use either a Kan extension or a universal construction to force global consistency.

This vignette is the symbolic overview page for the CATAGI block family. The dedicated follow-ups are:

- 07 Lux Neural Backend for trainable versions
- 23 TopoCoend Triage for a focused TopoCoend example
- 24 Bisimulation Quotient for the quotient/coequalizer example

Setup

```
using Pkg
Pkg.activate(joinpath(@__DIR__, ".."))

using FunctorFlow
```

BASKET ROCKET: Draft, Repair, Score

The enriched `basket_rocket_pipeline` now exposes both the draft and the repaired plan, plus a `:draft_repair_consistency` obstruction loss. By default that loss uses a token-overlap (`:jaccard`) comparator, which is friendlier to symbolic plans than pure `L2`.

```
pipeline = basket_rocket_pipeline(;
    config=BasketRocketPipelineConfig(
        basket_config=BASKETWorkflowConfig(),
        rocket_config=ROCKETRepairConfig(reducer=:concat)
    )
)
println(summary(pipeline))
println("Ports: ", collect(keys(pipeline.ports)))
println("Losses: ", collect(keys(pipeline.losses)))
```

```
FunctorFlow.Diagram
Ports: [:input, :draft_relation, :repair_relation, :draft, :consistency_loss, :output]
Losses: [:draft_repair_consistency]
```

```
coverage_score(plan::AbstractString, targets::Set{String}) =
    length(intersect(Set(split(lowercase(plan))), targets)) / length(targets)

target_skills = Set([
    "collect", "validate", "train", "red-team", "deploy", "monitor"
])

result = FunctorFlow.run(pipeline, Dict(
```

```
    pipeline.ports[:input].ref => Dict(
        :collect => "1. collect data\n",
        :validate => "2. validate schema\n",
        :train => "3. train model\n",
        :redteam => "4. red-team prompts\n",
        :deploy => "5. deploy service\n",
        :monitor => "6. monitor drift\n"
    ),
    pipeline.ports[:draft_relation].ref => Dict(
        :base => [:collect, :train, :deploy],
        :safety => [:validate, :redteam],
        :ops => [:monitor]
    ),
    pipeline.ports[:repair_relation].ref => Dict(
        :execution_ready => [:base, :safety, :ops]
    )
))

println("Draft: ", result.values[pipeline.ports[:draft].ref])
println("Repair: ", result.values[pipeline.ports[:output].ref])
println("Consistency loss: ", result.losses[:draft_repair_consistency])
println("Draft coverage: ", coverage_score(result.values[pipeline.ports[:draft].ref][:base], t
println("Repaired coverage: ", coverage_score(result.values[pipeline.ports[:output].ref][:exec
```

```
Draft: Dict{Any, Any}(:safety => "2. validate schema\n4. red-team prompts\n", :base => "1. col
Repair: Dict{Any, Any}(:execution_ready => "1. collect data\n3. train model\n5. deploy service
team prompts\n6. monitor drift\n")
Consistency loss: 0.18181818181818177
Draft coverage: 0.5
Repaired coverage: 1.0
```

This turns the agentic workflow into a diagram with an explicit plan-edit geometry:

1. Left-Kan composes fragments into a global draft.
2. Right-Kan repairs incomplete or inconsistent candidates.
3. An obstruction loss penalizes unnecessary semantic drift during repair.

In this version, the draft stage emits multiple plan views (`:base`, `:safety`, `:ops`), and the repair stage glues them into an execution-ready plan. That makes the pipeline feel much closer to an actual agent stack: draft specialized subplans first, then repair them into a single deployable action sequence.

Democritus Assembly: Glue Then Reground

`democritus_assembly_pipeline` adds a regrounding morphism from the glued global section back to local claims. The induced obstruction loss behaves like a sheaf-style coherence penalty: if the global causal story cannot be restricted back to the original local sections, the loss goes up.

```
demo = democritus_assembly_pipeline(; restrict_impl=identity)
println(summary(demo))
```

```
FunctorFlow.Diagram
```

```
demo_result = FunctorFlow.run(demo, Dict(
    demo.ports[:input].ref => Dict(
        :cell_a => Set(["Rain -> WetGrass"]),
        :cell_b => Set(["WetGrass -> SlipperyRoad"])
    ),
    demo.ports[:relation].ref => Dict(:world => [:cell_a, :cell_b])
))

println("Global state: ", demo_result.values[demo.ports[:global_output].ref])
println("Regrounded claims: ", demo_result.values[demo.ports[:regrounded_claims].ref])
println("Coherence loss: ", demo_result.losses[:section_coherence])
```

```
Global state: Dict{Any, Any}(:world => Set(Any["WetGrass -> SlipperyRoad", "Rain -> WetGrass"]
Regrounded claims: Dict{Any, Any}(:world => Set(Any["WetGrass -> SlipperyRoad", "Rain -> WetGr
Coherence loss: 0.4285714285714286
```

This is a useful template for fragmentary causal discovery, multi-document synthesis, or multi-view world-model assembly.

TopoCoend: Learn the Neighborhood Before Aggregating

`topocoend_block` explicitly models the step that many neural architectures leave implicit: infer the neighborhood relation first, then aggregate over it.

```
topo = topocoend_block(;
    infer_neighborhood_impl=x -> Dict(:scene => collect(keys(x))),
    lift_impl=identity
)
println(summary(topo))
```

```
FunctorFlow.Diagram
```

```
topo_result = FunctorFlow.run(topo, Dict(
    :Tokens => Dict(:obj1 => 1.0, :obj2 => 2.0, :obj3 => 4.0)
))

println("Learned relation: ", topo_result.values[:infer_neighborhood])
println("Global context: ", topo_result.values[:coend_aggregate])
```

```
Learned relation: Dict(:scene => [:obj1, :obj3, :obj2])
Global context: Dict(:scene => 2.3333333333333335)
```

The categorical point is that a coend-like aggregation should depend on an explicitly constructed cover, not just a fixed adjacency matrix. This is a natural bridge between attention, sheaves, and relation learning.

For a concrete single-block walkthrough, see 23 TopoCoend Triage. For the trainable version, see `build_topocoend_lux_model` in 07 Lux Neural Backend.

Lux-Backed Trainable Planner

The same `BASKET → ROCKET` geometry can be compiled into a differentiable Lux model. In this version both Kan reducers are learnable attention layers, and the `draft_repair_consistency` loss becomes a neural `L2` objective over plan embeddings.

```
using Lux, Random

d_model = 16
rng = Random.MersenneTwister(7)
planner_diagram = basket_rocket_pipeline(;
    config=BasketRocketPipelineConfig(
        basket_config=BASKETWorkflowConfig(reducer=:draft_attention),
        rocket_config=ROCKETRepairConfig(reducer=:repair_attention),
        consistency_comparator=:l2
    )
)
planner = compile_to_lux(planner_diagram;
    reducer_layers=Dict(
        :draft_attention => KETAttentionLayer(d_model; n_heads=2, name=:draft_attention),
        :repair_attention => KETAttentionLayer(d_model; n_heads=2, name=:repair_attention)
    )
```

```
)
ps, st = Lux.setup(rng, planner)

seq_len = 5
mask = Float32.(ones(seq_len, seq_len))
planner_inputs = Dict(
    planner_diagram.ports[:input].ref => randn(rng, Float32, d_model, seq_len),
    planner_diagram.ports[:draft_relation].ref => mask,
    planner_diagram.ports[:repair_relation].ref => mask
)

planner_result, st = planner(planner_inputs, ps, st)
println("Draft embedding shape: ", size(planner_result[:values][planner_diagram.ports[:draft].
println("Repair embedding shape: ", size(planner_result[:values][planner_diagram.ports[:output
println("Trainable consistency loss: ", planner_result[:losses][:draft_repair_consistency])
```

```
Draft embedding shape: (16, 5)
Repair embedding shape: (16, 5)
Trainable consistency loss: 25.501907348632812
```

This gives a direct path from the symbolic planner story to a trainable agent architecture: use the diagram to specify what should compose and repair, then let Lux learn how those maps are realized.

We can even optimize the planner directly against its own consistency signal:

```
using Optimisers, Zygote

opt = Optimisers.setup(Optimisers.Adam(5f-3), ps)
loss_trace = Float32[]
draft_ref = planner_diagram.ports[:draft].ref
repair_ref = planner_diagram.ports[:output].ref

for step in 1:5
    ((loss_val, st_new), grads) = Zygote.withgradient(ps) do p
        out, st_step = planner(planner_inputs, p, st)
        draft = out[:values][draft_ref]
        repair = out[:values][repair_ref]
        delta = repair .- draft
        (sum(abs2, delta) / length(delta), st_step)
    end
```

```
    push!(loss_trace, Float32(loss_val))
    opt, ps = Optimisers.update(opt, ps, grads[1])
    st = st_new
end

println("Consistency loss trace: ", round.(loss_trace; digits=4))
```

```
Consistency loss trace: Float32[8.1293, 4.498, 2.7115, 1.6874, 1.2151]
```

This is intentionally tiny, but it shows the full loop: the diagram determines which embeddings should agree, and Lux/Zygote optimize that consistency objective directly.

Horn Filling as an Obstruction Loss

`horn_fill_block` gives a compact 2-simplex model of higher-order consistency. The boundary path `d12 ∘ d01` should agree with the direct filler `d02`. Any disagreement becomes a first-class loss.

```
horn = horn_fill_block(;
    first_face_impl=x -> x + 1,
    second_face_impl=x -> x * 2,
    filler_impl=x -> (x + 1) * 2
)
println(summary(horn))
```

```
FunctorFlow.Diagram
```

```
horn_result = FunctorFlow.run(horn, Dict(:Vertex0 => 3))
println("Boundary path: ", horn_result.values[:horn_boundary])
println("Direct filler: ", horn_result.values[:d02])
println("Horn obstruction: ", horn_result.losses[:horn_obstruction])
```

```
Boundary path: 8
Direct filler: 8
Horn obstruction: 0.0
```

This is a lightweight entry point for simplicial regularization: consistency is enforced not just pairwise, but over triangular compositions.

`higher_horn_block` extends the same idea to longer boundary chains and multiple competing fillers. That lets one diagram encode a family of higher-order regularizers instead of a single triangle.

```
higher_horn = higher_horn_block(;
    config=HigherHornConfig(
        filler_faces=[:d03_exact, :d03_relaxed]
    ),
    boundary_face_impls=[x -> x + 1, x -> x * 2, x -> x - 3],
    filler_impls=[x -> ((x + 1) * 2) - 3, x -> ((x + 1) * 2) - 2]
)
println(summary(higher_horn))
```

```
FunctorFlow.Diagram
```

```
higher_horn_result = FunctorFlow.run(higher_horn, Dict(:Vertex0 => 3))
println("Higher boundary path: ", higher_horn_result.values[:higher_horn_boundary])
println("Exact filler: ", higher_horn_result.values[:d03_exact])
println("Relaxed filler: ", higher_horn_result.values[:d03_relaxed])
println("Higher horn obstruction: ", higher_horn_result.losses[:higher_horn_obstruction])
```

```
Higher boundary path: 5
Exact filler: 5
Relaxed filler: 6
Higher horn obstruction: 1.0
```

Behavioral Quotients via a Coequalizer

`bisimulation_quotient_block` turns a relation witness into two parallel behavior maps and then takes their coequalizer. The resulting quotient object is the categorical abstraction of "these two trajectories are behaviorally indistinguishable."

```
quotient = bisimulation_quotient_block()
println(summary(quotient))
println("Ports: ", collect(keys(quotient.ports)))
println("Objects: ", collect(keys(quotient.objects)))
println("Operations: ", collect(keys(quotient.operations)))
```

```
FunctorFlow.Diagram
Ports: [:relation, :left_behavior, :right_behavior, :output]
Objects: [:base__BisimRelation, :base__StateA, :base__StateB, :base__Behavior, :behavior_quoti
Operations: [:base__proj_left, :base__proj_right, :base__observe_a, :base__observe_b, :base__l
```

The key structure is:

- a relation object `R`
- projections `R → StateA` and `R → StateB`
- observation morphisms into a common behavior space
- a coequalizer that identifies behaviorally equivalent observations

This is a useful template for state abstraction in world models, option discovery, or policy compression.

The dedicated symbolic quotient walkthrough lives in 24 Bisimulation Quotient, and the Lux-backed counterpart is `build_bisimulation_quotient_lux_model` in 07 Lux Neural Backend.

Related Docs

- 04 Block Library for the reusable CATAGI block summary and API map.
- 07 Lux Neural Backend for the differentiable versions of the TopoCoend, horn, and bisimulation blocks.
- 23 TopoCoend Triage and 24 Bisimulation Quotient for focused single-block walkthroughs.

Takeaways

These additions make the `FunctorFlow` block layer more explicitly neurosymbolic:

- Planning becomes Kan aggregation + repair + edit consistency.
- Causal assembly becomes sheaf gluing + regrounding.
- Learned neighborhoods become explicit categorical objects.
- Higher-order coherence becomes a horn-filling obstruction, from triangles to horn families.
- State abstraction becomes a coequalizer over behavioral witnesses.

The most immediately useful pattern is still BASKET □ ROCKET, but it now sits inside a broader family of categorical agent architectures rather than being a one-off workflow block.