Topos Foundations

Sheaves, classifiers, and local-to-global coherence

Simon Frost

Table of contents

Introduction

Topos theory provides the foundational framework for FunctorFlow's local-to-global construction. In a topos, every logical operation has a spatial counterpart:

Topos concept	Intuition
Subobject classifier Ω	Generalized truth values
Sheaf	Local data that glues consistently
Section	A local data patch over a domain
Coherence check	Do local patches agree on overlaps?
Internal predicate	A proposition valued in Ω

The central pattern is the sheaf condition: local information patches can be glued into a globally consistent whole if and only if they agree on their overlaps. This is precisely the requirement for multi-source data fusion, distributed knowledge graphs, and modular AI systems.

FunctorFlow.jl provides first-class types for subobject classifiers, sheaf sections, coherence checks, and internal predicates, along with a `build_sheaf_diagram` function that compiles them into an executable diagram.

Setup

```
using FunctorFlow
```

SubobjectClassifier

A `SubobjectClassifier` defines the truth-value object Ω for the topos. In classical logic, $\Omega = \{\text{true}, \text{false}\}$. In a topos, Ω can carry richer truth values — for example, three-valued logic with `true`, `false`, and `unknown`.

```
Ω = SubobjectClassifier(:TruthValues;
    truth_object=:Omega,
    true_map=:true_map
)
```

```
SubobjectClassifier(:TruthValues, :Omega, :true_map, Set([Symbol("true"), Symbol("false")]), D
```

```
println("Name:         ", Ω.name)
println("Truth object: ", Ω.truth_object)
println("True map:     ", Ω.true_map)
```

```
Name:         TruthValues
Truth object: Omega
True map:     true_map
```

The `truth_object` names the truth-value object Ω in the diagram, and the `true_map` names the morphism $\text{true} : 1 \to \Omega$. Richer truth structures (three-valued, fuzzy, probabilistic) can be encoded via metadata and characteristic maps.

SheafSection

A `SheafSection` represents a local data patch: a piece of information defined over a specific domain (open set). A collection of sections forms a presheaf; when they satisfy the gluing condition, they form a sheaf.

```
s1 = SheafSection(:EntitySection;
    base_space=:PubMedAbstracts,
    section_data=Dict(:BRCA1 ⇒ "tumor suppressor",
                      :TP53 ⇒ "guardian of the genome"),
    domain=Set([:BRCA1, :TP53]),
    metadata=Dict(:source ⇒ "PubMed", :confidence ⇒ 0.95)
)
```

```
SheafSection(:EntitySection, :PubMedAbstracts, Dict(:TP53 => "guardian of the genome", :BRCA1
```

```
s2 = SheafSection(:RelationSection;
    base_space=:ClinicalTrials,
    section_data=Dict(:BRCA1 ⇒ "tumor suppressor",
                      :EGFR ⇒ "receptor tyrosine kinase"),
    domain=Set([:BRCA1, :EGFR]),
    metadata=Dict(:source ⇒ "ClinicalTrials.gov", :confidence ⇒ 0.88)
)
```

```
SheafSection(:RelationSection, :ClinicalTrials, Dict(:BRCA1 => "tumor suppressor", :EGFR => "r
```

```
println("Section 1 base space: ", s1.base_space, ", domain: ", s1.domain)
println("Section 2 base space: ", s2.base_space, ", domain: ", s2.domain)
```

```
Section 1 base space: PubMedAbstracts, domain: Set([:TP53, :BRCA1])
Section 2 base space: ClinicalTrials, domain: Set([:BRCA1, :EGFR])
```

Notice that both sections contain information about `:BRCA1` — this is the overlap where coherence will be checked.

SheafCoherenceCheck

A `SheafCoherenceCheck` verifies whether sections agree on their overlapping domains. If they do, the sections can be glued into a global section; if not, the coherence failure is reported.

```
overlap_checker = (s_a, s_b) -> begin
    # Check if sections agree on their overlapping domain
    overlap_domain = intersect(s_a.domain, s_b.domain)
    all(k -> get(s_a.section_data, k, nothing) == get(s_b.section_data, k, nothing), overlap_d
end

check = SheafCoherenceCheck([s1, s2];
    overlap_checker=overlap_checker,
    stability_penalty=0.1
)
```

```
SheafCoherenceCheck(SheafSection[SheafSection(:EntitySection, :PubMedAbstracts, Dict(:TP53 =>
```

```
println("Sections:          ", [s.name for s in check.sections])
println("Stability penalty: ", check.stability_penalty)
println("Has overlap checker: ", check.overlap_checker !== nothing)
```

```
Sections:          [:EntitySection, :RelationSection]
Stability penalty: 0.1
Has overlap checker: true
```

On the overlap `[:BRCA1]`, section 1 assigns `"tumor suppressor"` and section 2 also assigns `"tumor suppressor"` — so these sections are coherent and can be glued.

InternalPredicate

An `InternalPredicate` is a proposition valued in the subobject classifier Ω. It maps elements to truth values, enabling fine-grained membership queries within the topos.

```
pred = InternalPredicate(:IsHighConfidence, Ω;
    characteristic_map=entry -> haskey(entry, :confidence) && entry[:confidence] > 0.9
)
```

```
InternalPredicate(:IsHighConfidence, SubobjectClassifier(:TruthValues, :Omega, :true_map, Set(
```

```
println("Predicate name: ", pred.name)
println("Classifier:     ", pred.classifier.name)
println("Has char. map:  ", pred.characteristic_map !== nothing)
```

```
Predicate name: IsHighConfidence
Classifier:     TruthValues
Has char. map:  true
```

The predicate `:IsHighConfidence` maps each entry to `:true` if its confidence exceeds 0.9, and `:unknown` otherwise. This is evaluated internally within the topos — the result is a truth value in Ω, not a bare Boolean.

Building a Sheaf Diagram

`build_sheaf_diagram` compiles sections and their overlap structure into a full FunctorFlow `Diagram`. It creates:

1. An object for each section's domain
2. A morphism for each section (mapping domain to values)
3. Gluing morphisms for each overlap, enforcing the sheaf condition

```
sections = [s1, s2]

D = build_sheaf_diagram(sections; name=:DocumentKG)
```

```
Diagram :DocumentKG
  Objects:
    LocalSections::local_claims
    Overlap::overlap_region
    GlobalSection::global_state
  Operations:
    glue = Δ(LocalSections, along=Overlap, target=GlobalSection, reducer=:set_union)
  Ports:
    → sections (local_claims)
    → overlaps (overlap_region)
    ← global (global_state)
```

```
println("Objects:    ", collect(keys(D.objects)))
println("Operations: ", collect(keys(D.operations)))
println("Ports:      ", collect(keys(D.ports)))
```

```
Objects:    [:LocalSections, :Overlap, :GlobalSection]
Operations: [:glue]
Ports:      [:sections, :overlaps, :global]
```

The gluing morphisms connect overlapping domains and ensure that the values assigned to shared elements are consistent. When compiled and run, the diagram propagates local data patches through the gluing morphisms, producing a globally consistent knowledge graph.

```
ir = to_ir(D)
println("Diagram IR name: ", ir.name)
println("IR objects:      ", length(ir.objects))
println("IR operations:   ", length(ir.operations))
```

```
Diagram IR name: DocumentKG
IR objects:      3
IR operations:   1
```

Connection to Democritus

The `democritus_gluing_block` implements the same sheaf-gluing pattern as a reusable block. "Democritus" refers to the atomic decomposition and recomposition of knowledge — breaking information into local atoms (sections) and gluing them back together.

```
demo = democritus_gluing_block(; name=:DemoGlue)
```

```
Diagram :DemoGlue
  Objects:
    LocalClaims::local_claims
    OverlapRegion::overlap_region
    GlobalState::global_state
  Operations:
    glue = Δ(LocalClaims, along=OverlapRegion, target=GlobalState, reducer=:set_union)
  Ports:
    → input (local_claims)
    → relation (overlap_region)
    ← output (global_state)
```

```
println("Democritus block objects: ", collect(keys(demo.objects)))
println("Democritus block ops:    ", collect(keys(demo.operations)))
```

```
Democritus block objects: [:LocalClaims, :OverlapRegion, :GlobalState]
Democritus block ops:    [:glue]
```

The Democritus block is a pre-wired sheaf diagram with standard section and gluing morphisms. It provides the same guarantees as a manually constructed sheaf diagram but with less boilerplate.

You can inspect the structure to see the gluing morphisms.

```
for (name, op) in demo.operations
    if op isa Morphism
        println("  $name: $(op.source) → $(op.target) (morphism)")
    elseif op isa KanExtension
        println("  $name: $(op.source) along $(op.along) → $(op.target) (kan)")
    end
end
```

```
  glue: LocalClaims along OverlapRegion → GlobalState (kan)
```

Coherence Checking

The `check_coherence` function evaluates the sheaf condition on a `SheafCoherenceCheck`, verifying locality, gluing, and stability across all section pairs.

```
result = check_coherence(check)
println("Coherence passed: ", result.passed)
println("Locality: ", result.locality)
println("Gluing: ", result.gluing)
println("Stability: ", result.stability)
```

```
Coherence passed: true
Locality: true
Gluing: true
Stability: true
```

Because both `s1` and `s2` assign `"tumor suppressor"` to `:BRCA1`, the coherence check passes — locality, gluing, and stability are all satisfied.

Coherence failure

When sections disagree on their overlapping domain, the coherence check reports a failure:

```
sec_bad = SheafSection(:s_bad;
    base_space=:X,
    section_data=[100, 200, 300],
    domain=Set([:a, :b, :c])
)
check_bad = SheafCoherenceCheck([s1, sec_bad];
    overlap_checker=(s1, s2) -> false
)
result_bad = check_coherence(check_bad)
println("Coherence with disagreeing sections: ", result_bad.passed)
```

```
Coherence with disagreeing sections: false
```

The overlap checker explicitly returns `false`, so the coherence check fails — demonstrating how FunctorFlow surfaces inconsistencies rather than silently merging conflicting data.

Predicate Evaluation

`evaluate_predicate` applies an `InternalPredicate`'s characteristic map to a datum and returns a truth value in Ω as a `Symbol`.

```
omega = SubobjectClassifier(:Omega)
pred_pos = InternalPredicate(:positive, omega;
    characteristic_map=x -> x > 0
)

tv = evaluate_predicate(pred_pos, 5)
println("evaluate_predicate(pred_pos, 5) = ", tv)    # Symbol("true")

tv2 = evaluate_predicate(pred_pos, -3)
println("evaluate_predicate(pred_pos, -3) = ", tv2)  # Symbol("false")
```

```
evaluate_predicate(pred_pos, 5) = true
evaluate_predicate(pred_pos, -3) = false
```

The result is `Symbol("true")` or `Symbol("false")` — truth values in the subobject classifier, not bare Julia `Bools`. This distinction matters: in a richer topos, truth values can include `Symbol("unknown")`, `Symbol("probable")`, etc.

Subobject Classification

`classify_subobject` applies a characteristic function to every element of a data collection, producing a dictionary mapping each element to its truth value in Ω.

```
data = Dict(:a ⇒ 5, :b ⇒ -3, :c ⇒ 10, :d ⇒ 0)
classification = classify_subobject(omega, x -> x > 0, data)
for (k, v) in sort(collect(classification))
    println("  $k → $v")
end
```

```
  a → true
  b → false
  c → true
  d → false
```

This is the topos-theoretic analogue of filtering: instead of discarding elements, we classify every element by its membership in the subobject. The result is a complete map from the ambient set into Ω.

Internal Logic Operators

FunctorFlow provides internal logic operators that compose predicates within the topos. These operators produce new `InternalPredicates` whose characteristic maps combine the originals pointwise.

```
p = InternalPredicate(:positive, omega;
    characteristic_map=x -> x > 0
)
q = InternalPredicate(:even, omega;
    characteristic_map=x -> x % 2 == 0
)
```

```
InternalPredicate(:even, SubobjectClassifier(:Omega, :Omega, :true_map, Set([Symbol("true"), S
```

Conjunction (**`internal_and`**)

```
pq = internal_and(p, q)
println("4 is positive AND even: ", pq.characteristic_map(4))
println("3 is positive AND even: ", pq.characteristic_map(3))
```

```
4 is positive AND even: true
3 is positive AND even: false
```

Disjunction (**internal_or**)

```
p_or_q = internal_or(p, q)
println("-2 is positive OR even: ", p_or_q.characteristic_map(-2))
println("7 is positive OR even:  ", p_or_q.characteristic_map(7))
```

```
-2 is positive OR even: true
7 is positive OR even:  true
```

Negation (**internal_not**)

```
not_p = internal_not(p)
println("-1 is NOT positive: ", not_p.characteristic_map(-1))
println("5 is NOT positive:  ", not_p.characteristic_map(5))
```

```
-1 is NOT positive: true
5 is NOT positive:  false
```

These operators respect the internal logic of the topos: they work over any truth-value structure Ω supports, not just classical Boolean logic.

Interpretation

Topos foundations connect FunctorFlow to three practical patterns in AI:

Document-to-KG pipelines

Each document produces a local knowledge graph (a section over its domain). The sheaf condition ensures that entities mentioned in multiple documents receive consistent representations. The gluing morphisms implement entity resolution and fact reconciliation.

Multi-source data fusion

Different data sources (clinical trials, literature, expert annotations) are modeled as sections over overlapping domains. The coherence check verifies agreement on shared entities, and the sheaf diagram fuses them into a unified representation. Disagreements are surfaced as coherence failures rather than silently overwritten.

Distributed consensus

In a distributed system, each node holds a local view (section) of the global state. The sheaf condition is precisely the consensus requirement: all nodes must agree on their shared observations. The topos framework provides formal criteria for when local updates can be consistently merged and when they cannot.

The topos-theoretic perspective ensures that FunctorFlow's local-to-global constructions are not ad hoc — they inherit the full power of sheaf theory, including restriction maps, gluing axioms, and the internal logic of the topos. This makes the system both principled and practically useful for any task that involves assembling a global picture from local pieces.