Canonical V1 Example

End-to-end categorical model assembly

Simon Frost

2026-03-29

Table of contents

Overview

This vignette demonstrates FunctorFlow v1's core design goal: building AI systems by categorical construction. We walk through the canonical v1 workflow:

1. Define two KET models as first-class categorical model objects
2. Declare interface morphisms from each into a shared context
3. Build a pullback representing the joint constraint-compatible model
4. Verify the construction satisfies the universal property
5. Compile and run the pullback on concrete data
6. Emit a proof certificate for the construction
7. Construct a universal morphism from an external cone

This is the kind of compositional assembly that FunctorFlow v0 could not express — where categorical structure is the *actual method of assembly*, not merely a descriptive label.

Step 1: Define KET Models as Categorical Objects

We start with two KET modules that process different modalities — text and code — each with its own value space and attention incidence structure.

```
using FunctorFlow

# KET model for semantic text understanding
ket_text = ket_block(;
    name = :TextKET,
    value_kind = :tokens,
    incidence_kind = :attention,
    description = "Semantic text understanding via KET attention"
)

# KET model for code structure understanding
ket_code = ket_block(;
    name = :CodeKET,
    value_kind = :ast_nodes,
    incidence_kind = :syntax_edges,
    description = "Code structure understanding via KET attention"
)

println("TextKET: ", length(ket_text.objects), " objects, ",
        length(ket_text.operations), " operations")
println("CodeKET: ", length(ket_code.objects), " objects, ",
        length(ket_code.operations), " operations")
```

```
TextKET: 3 objects, 1 operations
CodeKET: 3 objects, 1 operations
```

Now we promote these diagrams to first-class categorical model objects — the key v1 abstraction. A `CategoricalModelObject` wraps a diagram with an ambient category, typed interface ports, boundary maps, and semantic laws that can be verified.

```
text_obj = CategoricalModelObject(ket_text;
    ambient_category = :FinVect,
    semantic_laws = [
        (_obj -> length(_obj.interface_ports) > 0),
        (_obj -> _obj.diagram !== nothing)
    ]
```

```
)

code_obj = CategoricalModelObject(ket_code;
    ambient_category = :FinVect,
    semantic_laws = [
        (_obj -> length(_obj.interface_ports) > 0),
        (_obj -> _obj.diagram !== nothing)
    ]
)

println("TextKET ports: ", length(text_obj.interface_ports))
println("CodeKET ports: ", length(code_obj.interface_ports))
println("TextKET boundary maps: ", length(text_obj.boundary_maps))
```

```
TextKET ports: 3
CodeKET ports: 3
TextKET boundary maps: 0
```

Verify that the semantic laws hold:

```
for (name, obj) in [("TextKET", text_obj), ("CodeKET", code_obj)]
    results = check_laws(obj)
    passed = all(last, results)
    println("$name laws: ", passed ? "✓ all passed" : "× some failed")
end
```

```
TextKET laws: ✓ all passed
CodeKET laws: ✓ all passed
```

Step 2: Register Models and Define Morphisms

Register both models in the global registry so they can be referenced by name in later constructions:

```
register_model!(text_obj)
register_model!(code_obj)
println("Registered models: ", collect(keys(MODEL_REGISTRY)))
```

```
Registered models: [:TextKET, :CodeKET]
```

Define interface morphisms from each model into a shared context object. These morphisms express how each model's output maps into a common representational space.

```
f = ModelMorphism(:text_to_shared, :TextKET, :SharedContext;
    functor_data = x -> x,
    metadata = Dict{Symbol, Any}(:role => :projection, :modality => :text))

g = ModelMorphism(:code_to_shared, :CodeKET, :SharedContext;
    functor_data = x -> x,
    metadata = Dict{Symbol, Any}(:role => :projection, :modality => :code))

println("f: ", f.source, " → ", f.target)
println("g: ", g.source, " → ", g.target)
```

```
f: TextKET → SharedContext
g: CodeKET → SharedContext
```

We can also compose morphisms and apply them to model objects:

```
# Apply the text projection to get the shared context image
shared_from_text = FunctorFlow.apply(f, text_obj)
println("Image of TextKET under f: ", shared_from_text.name)
println("  ambient category: ", shared_from_text.ambient_category)
```

```
Image of TextKET under f: SharedContext
  ambient category: FinVect
```

Step 3: Build the Pullback

The pullback is the central v1 construction: it takes two models that both map into a shared context and produces the most general model that is simultaneously compatible with both.

$$\begin{array}{ccc} \text{TextKET} \times_{\text{SharedContext}} \text{CodeKET} & \xrightarrow{\pi_1} & \text{TextKET} \\ \downarrow \pi_2 & & \downarrow f \\ \text{CodeKET} & \xrightarrow{g} & \text{SharedContext} \end{array}$$

```
pb = pullback(ket_text, ket_code; over = :SharedContext, name = :JointModel)

println("Pullback: ", pb.name)
println("  Cone objects: ", length(pb.cone.objects))
println("  Cone operations: ", length(pb.cone.operations))
println("  Shared object: ", pb.shared_object)
println("  Projection 1 (left): ", pb.projection1)
println("  Projection 2 (right): ", pb.projection2)
println("  Interface morphisms: ", pb.interface_morphisms)
```

```
Pullback: JointModel
  Cone objects: 9
  Cone operations: 4
  Shared object: SharedContext
  Projection 1 (left): left
  Projection 2 (right): right
  Interface morphisms: [:proj_left_output, :proj_right_output]
```

Inspect the cone diagram's structure — it contains copies of both models plus the shared object and commuting constraints:

```
println("Objects in pullback cone:")
for (name, obj) in pb.cone.objects
    println("  ", name, " (", obj.kind, ")")
end

println("\nOperations:")
for (name, op) in pb.cone.operations
    println("  ", name, ": ", typeof(op).name.name)
end

println("\nLosses (commuting constraints):")
for (name, loss) in pb.cone.losses
    println("  ", name, ": ", typeof(loss).name.name)
end
```

```
Objects in pullback cone:
  left__Values (messages)
  left__Incidence (relation)
  left__ContextualizedValues (contextualized_messages)
  right__Values (messages)
```

```
  right__Incidence (relation)
  right__ContextualizedValues (contextualized_messages)
  SharedContext (shared_interface)
  left__aggregate (object)
  right__aggregate (object)

Operations:
  left__aggregate: KanExtension
  right__aggregate: KanExtension
  proj_left_output: Morphism
  proj_right_output: Morphism

Losses (commuting constraints):
  JointModel_commuting: ObstructionLoss
```

Step 4: Verify the Universal Property

Call `verify()` to check that the pullback satisfies its universal property: both factors are present, the shared object exists, and the commuting constraint is enforced.

```
v = verify(pb)

println("Verification passed: ", v.passed)
println("Construction type: ", v.construction)
println("\nIndividual checks:")
for (check, passed) in v.checks
    status = passed ? "✓" : "×"
    println("  $status $check")
end
```

```
Verification passed: true
Construction type: pullback

Individual checks:
  ✓ has_commuting_constraint
  ✓ has_left_factor
  ✓ has_interface_morphisms
  ✓ has_shared_object
  ✓ has_right_factor
```

Step 5: Compile and Run

Compile the pullback into an executable `CompiledDiagram` and run it on concrete data:

```
compiled = compile_construction(pb)
println("Compiled: ", typeof(compiled))
println("  Diagram: ", compiled.diagram.name)
println("  Reducers: ", length(compiled.reducers))
```

```
Compiled: CompiledDiagram
  Diagram: JointModel
  Reducers: 7
```

Supply input data for both modalities and the shared context:

```
input_data = Dict(
    :left__Values => Dict(:t1 => [1.0, 0.5, 0.3], :t2 => [0.3, 0.8, 0.1]),
    :left__Incidence => Dict((:t1, :t2) => 0.7),
    :right__Values => Dict(:n1 => [0.1, 0.9, 0.4], :n2 => [0.4, 0.6, 0.2]),
    :right__Incidence => Dict((:n1, :n2) => 0.5),
    :SharedContext => Dict(:shared => [1.0, 1.0, 1.0])
)

# Bind identity implementations for projection morphisms
morphisms = Dict{Symbol, Function}(
    :proj_left_output => identity,
    :proj_right_output => identity,
)

result = FunctorFlow.run(compiled, input_data; morphisms = morphisms)

println("Computed values:")
for (k, v) in result.values
    println("  ", k, " → ", typeof(v))
end

println("\nLosses (commuting square):")
for (k, v) in result.losses
    println("  ", k, " = ", v)
end
```

```
Computed values:
  SharedContext → Dict{Any, Any}
  right__aggregate → Dict{Any, Any}
  proj_right_output → Dict{Any, Any}
  left__Values → Dict{Symbol, Vector{Float64}}
  proj_left_output → Dict{Any, Any}
  right__Values → Dict{Symbol, Vector{Float64}}
  right__Incidence → Dict{Tuple{Symbol, Symbol}, Float64}
  left__aggregate → Dict{Any, Any}
  left__Incidence → Dict{Tuple{Symbol, Symbol}, Float64}

Losses (commuting square):
  JointModel_commuting = 0.0
```

The commuting square loss being 0 means that the projections through both paths agree — the pullback constraint is satisfied at runtime.

Step 6: Proof Certificate

Emit a Lean 4 proof certificate that the pullback satisfies its declared interface:

```
cert = render_construction_certificate(pb)
println(cert)
```

```
-- Auto-generated by FunctorFlow.jl
namespace FunctorFlowProofs.Generated.JointModel

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "JointModel",
  objects := ["left__Values", "left__Incidence", "left__ContextualizedValues", "right__Values"
  operations := [{ name := "left__aggregate", kind := OperationKind.leftKan, refs := ["left__V
  ports := []
}

def exportedArtifact : LoweringArtifact := {
  diagram := exportedDiagram,
  resolvedRefs := true,
  portsClosed := true
}
```

```
theorem exportedArtifact_checks : exportedArtifact.check = true := by native_decide

theorem exportedArtifact_sound : exportedArtifact.Sound :=
  LoweringArtifact.sound_of_check exportedArtifact_checks


-- Pullback construction declaration
def pullbackDecl : ConstructionDecl := {
  kind := ConstructionKind.pullback,
  diagram := exportedDiagram,
  projection1 := "left",
  projection2 := "right",
  sharedObject := "SharedContext",
  interfaceMorphisms := ["proj_left_output", "proj_right_output"]
}

-- Commuting square theorem: both projections agree on shared interface
theorem pullback_commutes : pullbackDecl.CommutingSquare := by
  exact ConstructionDecl.commuting_of_loss exportedArtifact

end FunctorFlowProofs.Generated.JointModel
```

We can also produce a standard Lean certificate for the underlying diagram:

```
lean_cert = render_lean_certificate(pb.cone)
println(first(lean_cert, 500), "...")
```

```
-- Auto-generated by FunctorFlow.jl
namespace FunctorFlowProofs.Generated.JointModel

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "JointModel",
  objects := ["left__Values", "left__Incidence", "left__ContextualizedValues", "right__Values"
  operations := [{ name := "left__aggregate", kind := OperationKind.leftKan, refs := ["left__V
```

Step 7: Universal Morphism

Given any other model (a “test cone”) that also maps into both factors, the universal property guarantees a unique mediating morphism into the pullback:

```
# Build a test cone — any other model that wants to interact with both TextKET and CodeKET
test_cone = ket_block(; name = :ExternalModel)
mediating = universal_morphism(pb, test_cone)

println("Mediating diagram: ", mediating.name)
println("  Objects: ", length(mediating.objects))
println("  Operations: ", length(mediating.operations))
println("\n  Mediating objects:")
for (name, obj) in mediating.objects
    println("    ", name, " (", obj.kind, ")")
end
```

```
Mediating diagram: mediating
  Objects: 13
  Operations: 6

  Mediating objects:
    cone__Values (messages)
    cone__Incidence (relation)
    cone__ContextualizedValues (contextualized_messages)
    pullback__left__Values (messages)
    pullback__left__Incidence (relation)
    pullback__left__ContextualizedValues (contextualized_messages)
    pullback__right__Values (messages)
    pullback__right__Incidence (relation)
    pullback__right__ContextualizedValues (contextualized_messages)
    pullback__SharedContext (shared_interface)
    pullback__left__aggregate (object)
    pullback__right__aggregate (object)
    cone__aggregate (object)
```

Beyond the Canonical Example

This end-to-end pipeline demonstrates the v1 design thesis:

- Categorical objects elevate KET models from macro expansions to first-class compositional entities
- Universal constructions (pullbacks, pushouts, etc.) serve as the actual assembly method — not ad hoc glue code
- Verification checks structural properties at the categorical level
- Compilation lowers constructions to executable systems while preserving their provenance

- Proof certificates record the construction faithfully for formal verification
- Universal morphisms express the interface guarantee: any compatible model factors uniquely through the construction

The same pattern extends to pushouts (model merging), products (independent combination), coproducts (hypothesis aggregation), and equalizers (consistency enforcement). Together they provide a principled compositional language for building AI systems from categorical structure.

```
# Clean up the model registry
delete!(MODEL_REGISTRY, :TextKET)
delete!(MODEL_REGISTRY, :CodeKET)
println("✅ Canonical v1 example complete")
```

```
✅ Canonical v1 example complete
```