Universal Constructions

Pullbacks, pushouts, products, and coproducts

Simon Frost

Table of contents

Introduction

Universal constructions are the canonical patterns of category theory — limits and colimits that characterize how objects and morphisms can be combined. They include:

Construction	Type	Intuition
Product	Limit	Parallel combination (AND)
Coproduct	Colimit	Choice / branching (OR)
Pullback	Limit	Join / merge over shared target
Pushout	Colimit	Gluing over shared source
Equalizer	Limit	Subspace where two maps agree
Coequalizer	Colimit	Quotient forcing two maps to agree

FunctorFlow.jl provides first-class functions for all six constructions. Each returns a result struct containing the new diagram elements (objects and morphisms) that realize the universal property.

Setup

```
using FunctorFlow
```

Products

A product of two objects A and B is an object $A \times B$ equipped with projection morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. It represents the most general way to "observe both A and B simultaneously."

```
D_A = Diagram(:ComponentA)
add_object!(D_A, :A; kind=:state, description="First component")

D_B = Diagram(:ComponentB)
add_object!(D_B, :B; kind=:state, description="Second component")

result = FunctorFlow.product(D_A, D_B)
```

```
ProductResult(:Product, Diagram :Product ⟨2 objects, 0 morphisms, 0 Kan, 0 losses⟩, [:factor_1
```

The result struct contains the product diagram and the namespace prefixes for each factor.

```
println("Product name:  ", result.name)
println("Projections:   ", result.projections)
println("Product diagram objects: ", collect(keys(result.product_diagram.objects)))
```

```
Product name:  Product
Projections:   [:factor_1, :factor_2]
Product diagram objects: [:factor_1__A, :factor_2__B]
```

Coproducts

A coproduct of A and B is the dual: an object $A \sqcup B$ with injection morphisms $\iota_1 : A \to A \sqcup B$ and $\iota_2 : B \to A \sqcup B$. It represents a choice — data arrives from either A or B.

```
D_L = Diagram(:LeftBranch)
add_object!(D_L, :A; kind=:state, description="Left branch")

D_R = Diagram(:RightBranch)
add_object!(D_R, :B; kind=:state, description="Right branch")

result_cp = FunctorFlow.coproduct(D_L, D_R)
```

```
CoproductResult(:Coproduct, Diagram :Coproduct ⟨2 objects, 0 morphisms, 0 Kan, 0 losses⟩, [:su
```

```
println("Coproduct name:  ", result_cp.name)
println("Injections:      ", result_cp.injections)
println("Coproduct diagram objects: ", collect(keys(result_cp.coproduct_diagram.objects)))
```

```
Coproduct name:  Coproduct
Injections:      [:summand_1, :summand_2]
Coproduct diagram objects: [:summand_1__A, :summand_2__B]
```

In an AI context, coproducts model branching architectures: the system can receive input from multiple sources and route them into a unified representation.

Pullbacks

A pullback is the limit of a cospan: given morphisms $f : A \to C$ and $g : B \to C$ that share a common target, the pullback P is the "most general" object that maps into both A and B such that the resulting square commutes.

$$\begin{array}{ccc} P & \xrightarrow{p_1} & A \\ \downarrow{\scriptstyle p_2} & & \downarrow{\scriptstyle f} \\ B & \xrightarrow{g} & C \end{array}$$

```
D1 = Diagram(:LeftSource)
add_object!(D1, :A; kind=:state, description="Left source")
add_object!(D1, :C; kind=:state, description="Shared target")
add_morphism!(D1, :f, :A, :C; description="Left map")

D2 = Diagram(:RightSource)
add_object!(D2, :B; kind=:state, description="Right source")
add_object!(D2, :C; kind=:state, description="Shared target")
add_morphism!(D2, :g, :B, :C; description="Right map")

result_pb = FunctorFlow.pullback(D1, D2; over=:SharedInterface)
```

```
PullbackResult(:Pullback, Diagram :Pullback ⟨5 objects, 2 morphisms, 0 Kan, 0 losses⟩, :left,
```

```
println("Pullback name:       ", result_pb.name)
println("Projection 1:        ", result_pb.projection1)
println("Projection 2:        ", result_pb.projection2)
println("Pullback objects:    ", collect(keys(result_pb.cone.objects)))
println("Pullback operations: ", collect(keys(result_pb.cone.operations)))
```

```
Pullback name:       Pullback
Projection 1:        left
Projection 2:        right
Pullback objects:    [:left__A, :left__C, :right__B, :right__C, :SharedInterface]
Pullback operations: [:left__f, :right__g]
```

The pullback object P comes equipped with projections $p_1 : P \to A$ and $p_2 : P \to B$. The universal property guarantees that $f \circ p_1 = g \circ p_2$.

In data terms, a pullback is a join: given two datasets that map into a common schema, the pullback is their join on that schema.

Pushouts

A pushout is the colimit of a span: given $f : C \to A$ and $g : C \to B$ that share a common source, the pushout Q is the "smallest" object that both A and B map into, identifying elements that came from the same element of C.

$$\begin{array}{ccc} C & \xrightarrow{f} & A \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle q_1} \\ B & \xrightarrow{q_2} & Q \end{array}$$

```
D3 = Diagram(:LeftTarget)
add_object!(D3, :C; kind=:state, description="Shared source")
add_object!(D3, :A; kind=:state, description="Left target")
add_morphism!(D3, :f, :C, :A; description="Left map")

D4 = Diagram(:RightTarget)
add_object!(D4, :C; kind=:state, description="Shared source")
add_object!(D4, :B; kind=:state, description="Right target")
add_morphism!(D4, :g, :C, :B; description="Right map")

result_po = FunctorFlow.pushout(D3, D4; along=:SharedSubobject)
```

```
PushoutResult(:Pushout, Diagram :Pushout ⟨5 objects, 2 morphisms, 0 Kan, 0 losses⟩, :left, :ri
```

```
println("Pushout name:       ", result_po.name)
println("Injection 1:        ", result_po.injection1)
println("Injection 2:        ", result_po.injection2)
println("Pushout objects:    ", collect(keys(result_po.cocone.objects)))
println("Pushout operations: ", collect(keys(result_po.cocone.operations)))
```

```
Pushout name:       Pushout
Injection 1:        left
Injection 2:        right
Pushout objects:    [:left__C, :left__A, :right__C, :right__B, :SharedSubobject]
Pushout operations: [:left__f, :right__g]
```

The pushout Q comes with injections $q_1 : A \to Q$ and $q_2 : B \to Q$. In practice, pushouts model gluing: two data sources that share a common origin are merged into one, with the shared part identified.

Equalizers

An equalizer is the limit of a parallel pair: given two morphisms $f, g : A \to B$, the equalizer E is the subobject of A on which f and g agree.

$$E \xrightarrow{e} A \underset{g}{\overset{f}{\rightrightarrows}} B$$

```
D_eq = Diagram(:EqualizerExample)
add_object!(D_eq, :A; kind=:state, description="Source")
add_object!(D_eq, :B; kind=:state, description="Target")

add_morphism!(D_eq, :f, :A, :B; description="First parallel map")
add_morphism!(D_eq, :g, :A, :B; description="Second parallel map")

result_eq = FunctorFlow.equalizer(D_eq, :f, :g)
```

```
EqualizerResult(:Equalizer, Diagram :Equalizer ⟨2 objects, 2 morphisms, 0 Kan, 1 losses⟩, :bas
```

```
println("Equalizer name: ", result_eq.name)
println("Equalizer map:  ", result_eq.equalizer_map)
println("Equalizer diagram objects: ", collect(keys(result_eq.equalizer_diagram.objects)))
println("Equalizer diagram losses:  ", collect(keys(result_eq.equalizer_diagram.losses)))
```

```
Equalizer name: Equalizer
Equalizer map:  base__f
Equalizer diagram objects: [:base__A, :base__B]
Equalizer diagram losses:  [:Equalizer_eq_loss]
```

The equalizer is useful in learning contexts: it identifies the subspace of inputs where two different processing paths produce identical outputs — the "agreement kernel."

Coequalizers

A coequalizer is the colimit dual of the equalizer: given two morphisms $f, g : A \to B$, the coequalizer Q is the quotient of B that identifies $f(a)$ with $g(a)$ for all $a \in A$.

$$A \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow{q} Q$$

Where an equalizer *finds* where two maps agree (a subobject), a coequalizer *forces* two maps to agree (a quotient).

```
D_coeq = Diagram(:CoequalizerExample)
add_object!(D_coeq, :A; kind=:state, description="Source")
add_object!(D_coeq, :B; kind=:state, description="Target")

add_morphism!(D_coeq, :f, :A, :B; description="First parallel map")
add_morphism!(D_coeq, :g, :A, :B; description="Second parallel map")

result_coeq = coequalizer(D_coeq, :f, :g)
```

```
CoequalizerResult(:Coequalizer, Diagram :Coequalizer ⟨3 objects, 3 morphisms, 0 Kan, 1 losses⟩
```

```
println("Coequalizer name:     ", result_coeq.name)
println("Quotient object:      ", result_coeq.quotient_object)
println("Coequalizer map (q):  ", result_coeq.coequalizer_map)
println("Diagram objects:      ", collect(keys(result_coeq.coequalizer_diagram.objects)))
println("Diagram losses:       ", collect(keys(result_coeq.coequalizer_diagram.losses)))
```

```
Coequalizer name:     Coequalizer
Quotient object:      Coequalizer_Quotient
Coequalizer map (q):  Coequalizer_q
Diagram objects:      [:base__A, :base__B, :Coequalizer_Quotient]
Diagram losses:       [:Coequalizer_coeq_loss]
```

The coequalizer enforces $q \circ f = q \circ g$ via an obstruction loss. In AI terms, coequalizers model:

- Equivalence classes: collapsing representations that two maps agree should be identified
- Symmetry quotienting: removing redundant structure by identifying symmetric states
- Consensus merging: merging outputs that multiple processing paths declare equivalent

Running a coequalizer

```
bind_morphism!(result_coeq.coequalizer_diagram, Symbol(:base__, :f), x -> x * 2)
bind_morphism!(result_coeq.coequalizer_diagram, Symbol(:base__, :g), x -> x .+ 1)
bind_morphism!(result_coeq.coequalizer_diagram, result_coeq.coequalizer_map, identity)

compiled_coeq = compile_construction(result_coeq)
result_coeq_run = FunctorFlow.run(compiled_coeq, Dict(Symbol(:base__, :A) => [1.0, 2.0, 3.0]))
println("q∘f: ", result_coeq_run.values[Symbol(:Coequalizer, :_qf)])
println("q∘g: ", result_coeq_run.values[Symbol(:Coequalizer, :_qg)])
println("Coequalizer loss: ", result_coeq_run.losses)
```

```
q∘f: [2.0, 4.0, 6.0]
q∘g: [2.0, 3.0, 4.0]
Coequalizer loss: Dict(:Coequalizer_coeq_loss => 2.23606797749979)
```

The loss measures how far $q \circ f$ and $q \circ g$ are from agreeing — training on this loss pushes the quotient map q toward identifying the images of f and g.

Verifying a coequalizer

```
v_coeq = verify(result_coeq)
println("Verification passed: ", v_coeq.passed)
println("Checks: ", v_coeq.checks)
```

```
Verification passed: true
Checks: Dict{Symbol, Bool}(:has_coeq_loss => 1, :has_coequalizer_map => 1, :map_targets_quotie
```

Composing Universal Constructions

Universal constructions compose naturally. For example, we can first take a product, then use the result as input to a pullback.

```
D_X = Diagram(:ComponentX)
add_object!(D_X, :X; kind=:state)

D_Y = Diagram(:ComponentY)
add_object!(D_Y, :Y; kind=:state)

# Step 1: Product of X and Y
prod_result = FunctorFlow.product(D_X, D_Y; name=:XY_Product)
println("Product projections: ", prod_result.projections)
```

```
Product projections: [:factor_1, :factor_2]
```

Now take the product diagram and another diagram with a shared morphism target, then form the pullback.

```
D_W = Diagram(:ComponentW)
add_object!(D_W, :W; kind=:state)

pb_result = FunctorFlow.pullback(prod_result.product_diagram, D_W; over=:SharedInterface)
```

```
PullbackResult(:Pullback, Diagram :Pullback ⟨4 objects, 0 morphisms, 0 Kan, 0 losses⟩, :left,
```

```
println("Composed pullback objects: ", collect(keys(pb_result.cone.objects)))
```

```
Composed pullback objects: [:left__factor_1__X, :left__factor_2__Y, :right__W, :SharedInterfac
```

This pattern — product then pullback — arises naturally when you want to join two parallel channels (X and Y) and then intersect the result with a third source (W) over a shared target (Z).

Verification

Every universal construction returned by FunctorFlow can be verified — the `verify` function checks that the result satisfies the required universal property (e.g., commutativity of the pullback square, existence of projections for products).

Verifying a product

```
v_prod = verify(result)
println("Verification passed: ", v_prod.passed)
println("Checks: ", v_prod.checks)
```

```
Verification passed: true
Checks: Dict{Symbol, Bool}(:has_factor_factor_1 => 1, :has_factor_factor_2 => 1)
```

Verifying a coproduct

```
v_cp = verify(result_cp)
println("Verification passed: ", v_cp.passed)
println("Checks: ", v_cp.checks)
```

```
Verification passed: true
Checks: Dict{Symbol, Bool}(:has_summand_summand_1 => 1, :has_summand_summand_2 => 1)
```

Verifying a pullback

```
v_pb = verify(result_pb)
println("Verification passed: ", v_pb.passed)
println("Checks: ", v_pb.checks)
```

```
Verification passed: false
Checks: Dict{Symbol, Bool}(:has_commuting_constraint => 0, :has_left_factor => 1, :has_interfa
```

Verifying a pushout

```
v_po = verify(result_po)
println("Verification passed: ", v_po.passed)
println("Checks: ", v_po.checks)
```

```
Verification passed: false
Checks: Dict{Symbol, Bool}(:has_left_factor => 1, :has_interface_morphisms => 0, :has_shared_o
```

Verifying an equalizer

```
v_eq = verify(result_eq)
println("Verification passed: ", v_eq.passed)
println("Checks: ", v_eq.checks)
```

```
Verification passed: true
Checks: Dict{Symbol, Bool}(:has_eq_loss => 1, :has_equalizer_map => 1)
```

Verification is lightweight and runs entirely on the symbolic diagram — no numerical computation is needed. If a construction is ill-formed (e.g., a missing projection), the `checks` list will contain the failing condition.

Compilation

Once a construction is verified, it can be compiled into a `CompiledDiagram` — a lower-level representation ready for code generation or execution. The `compile_construction` function performs this step.

```
compiled_pb = compile_construction(result_pb)
println("Compiled type: ", typeof(compiled_pb))
```

```
Compiled type: CompiledDiagram
```

```
compiled_po = compile_construction(result_po)
println("Compiled type: ", typeof(compiled_po))
```

```
Compiled type: CompiledDiagram
```

```
compiled_eq = compile_construction(result_eq)
println("Compiled type: ", typeof(compiled_eq))
```

```
Compiled type: CompiledDiagram
```

Compilation freezes the diagram structure and resolves all symbolic references, producing an object that downstream tooling (e.g., Julia code emitters or ONNX exporters) can consume directly.

Universal Morphisms

The defining feature of a universal construction is its universal morphism: given any other cone (or cocone) over the same diagram, there exists a unique mediating morphism from the external cone into the limit (or from the colimit into the external cocone).

FunctorFlow makes this explicit via `universal_morphism`. We construct a test cone — a diagram that maps into the same cospan as the pullback — and ask for the mediating morphism.

```
D_cone = FunctorFlow.Diagram(:TestCone)
add_object!(D_cone, :T; kind=:state, description="Test apex")
add_object!(D_cone, :A; kind=:state, description="Left leg target")
add_object!(D_cone, :B; kind=:state, description="Right leg target")
add_object!(D_cone, :C; kind=:state, description="Shared target")
add_morphism!(D_cone, :t_to_a, :T, :A; description="Left leg")
```

```
add_morphism!(D_cone, :t_to_b, :T, :B; description="Right leg")
add_morphism!(D_cone, :f, :A, :C; description="Left map")
add_morphism!(D_cone, :g, :B, :C; description="Right map")

mediating = universal_morphism(result_pb, D_cone)
println("Universal morphism diagram: ", mediating.name)
```

```
Universal morphism diagram: mediating
```

The returned diagram encodes the unique mediating morphism $u : T \to P$ such that $p_1 \circ u = t\_to\_a$ and $p_2 \circ u = t\_to\_b$. This is the categorical guarantee: the pullback is the "best" such object.

Proof Certificates

For auditing and documentation, FunctorFlow can render a human-readable proof certificate summarizing what was constructed, what universal property it satisfies, and what checks were performed.

```
cert_pb = render_construction_certificate(result_pb)
println(cert_pb)
```

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

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "Pullback",
  objects := ["left__A", "left__C", "right__B", "right__C", "SharedInterface"],
  operations := [{ name := "left__f", kind := OperationKind.morphism, refs := ["left__A", "lef
  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 := "SharedInterface",
  interfaceMorphisms := []
}

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

end FunctorFlowProofs.Generated.Pullback
```

```
cert_po = render_construction_certificate(result_po)
println(cert_po)
```

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

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "Pushout",
  objects := ["left__C", "left__A", "right__C", "right__B", "SharedSubobject"],
  operations := [{ name := "left__f", kind := OperationKind.morphism, refs := ["left__C", "lef
  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


-- Pushout construction declaration
def pushoutDecl : ConstructionDecl := {
  kind := ConstructionKind.pushout,
  diagram := exportedDiagram,
  injection1 := "left",
  injection2 := "right",
  sharedObject := "SharedSubobject",
  interfaceMorphisms := []
}

end FunctorFlowProofs.Generated.Pushout
```

```
cert_eq = render_construction_certificate(result_eq)
println(cert_eq)
```

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

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "Equalizer",
  objects := ["base__A", "base__B"],
  operations := [{ name := "base__f", kind := OperationKind.morphism, refs := ["base__A", "bas
  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

end FunctorFlowProofs.Generated.Equalizer
```

Proof certificates are plain-text summaries intended for inclusion in design documents or automated reports. They provide a traceable record that the construction was well-formed before any code was generated from it.

Unicode Operators: ⊗ and ⊕

FunctorFlow provides unicode infix operators for the two most common universal constructions:

- `⊗` (product): `D1 ⊗ D2` is equivalent to `product(D1, D2)`
- `⊕` (coproduct): `D1 ⊕ D2` is equivalent to `coproduct(D1, D2)`

```
D_X = Diagram(:X)
add_object!(D_X, :X; kind=:state)

D_Y = Diagram(:Y)
add_object!(D_Y, :Y; kind=:state)

# Product via unicode operator
prod_unicode = D_X ⊗ D_Y
println("⊗ product name: ", prod_unicode.name)
println("⊗ product objects: ", collect(keys(prod_unicode.product_diagram.objects)))

# Coproduct via unicode operator
coprod_unicode = D_X ⊕ D_Y
println("⊕ coproduct name: ", coprod_unicode.name)
println("⊕ coproduct objects: ", collect(keys(coprod_unicode.coproduct_diagram.objects)))
```

```
⊗ product name: X_⊗_Y
⊗ product objects: [:factor_1__X, :factor_2__Y]
⊕ coproduct name: X_⊕_Y
⊕ coproduct objects: [:summand_1__X, :summand_2__Y]
```

These operators make categorical architecture code read like mathematical notation.

Interpretation

Universal constructions provide a design vocabulary for AI architectures:

- Products model parallel channels: process A and B independently but keep both results available. This is the pattern behind multi-modal architectures where vision and text features are maintained in parallel.
- Coproducts model choice or branching: the system handles inputs from either A or B. This captures routing, mixture-of-experts gates, or any architecture that selects among alternatives.
- Pullbacks model joins: given two representations that share a common target schema, the pullback is their intersection. This is the categorical version of a database join, or in neural terms, a cross-attention merge.
- Pushouts model gluing: two representations that share a common origin are fused into one, identifying the shared part. This is the pattern behind data fusion and the Democritus gluing block.
- Equalizers model agreement: the subspace where two different computations produce the same answer. This is related to consensus mechanisms and can serve as a learned filter.
- Coequalizers model quotienting: collapsing a representation by identifying elements that two maps declare equivalent. This captures symmetry reduction, equivalence-class learning, and representation compression — the dual of the equalizer's "agreement kernel."

By expressing architectures in terms of these universal constructions, FunctorFlow ensures that the resulting diagrams have well-defined compositional semantics — they can be combined, nested, and reasoned about using the full machinery of category theory.