Causal Semantics

RN-Kan-Do-Calculus for causal reasoning

Simon Frost

Table of contents

Introduction

FunctorFlow's causal semantics are based on the RN-Kan-Do-Calculus, a categorical framework that re-interprets Pearl's do-calculus through Kan extensions:

Causal operation	Categorical construction	Kan direction
Conditioning $P(Y \mid X)$	Right Kan extension	RIGHT
Intervention $P(Y \mid \mathrm{do}(X))$	Left Kan extension	LEFT
Counterfactual $P(Y_x \mid X')$	Composition of Kan extensions	LEFT + RIGHT

The key insight is that conditioning (passive observation) corresponds to right Kan extension (a limit, computing the "best completion"), while intervention (actively setting a variable) corresponds to left Kan extension (a colimit, pushing information forward). This gives the do-calculus a diagrammatic and functorial foundation.

FunctorFlow provides `CausalContext`, `CausalDiagram`, and transport rules to build and reason about causal models purely within the categorical framework.

Setup

```
using FunctorFlow
```

CausalContext

A `CausalContext` declares the observational and interventional regimes. It specifies the two perspectives between which causal reasoning operates.

```
ctx = CausalContext(:SmokingContext;
    observational_regime=:obs,
    interventional_regime=:do
)
```

```
CausalContext(:SmokingContext, :obs, :do, Dict{Symbol, Any}())
```

```
println("Name:                   ", ctx.name)
println("Observational regime:   ", ctx.observational_regime)
println("Interventional regime:  ", ctx.interventional_regime)
```

```
Name:                   SmokingContext
Observational regime:   obs
Interventional regime:  do
```

The context serves as the "type signature" of the causal problem — it tells FunctorFlow which regimes are in play.

CausalDiagram

`build_causal_diagram` creates a `CausalDiagram` — a diagram with explicit causal Kan semantics. It creates objects for observations, causal structure, and both interventional and conditional state targets. A left Kan extension handles intervention (do-calculus pushforward) and a right Kan extension handles conditioning (observational completion).

```
cd = build_causal_diagram(:SmokingCancer; context=ctx)
```

```
CausalDiagram(:SmokingCancer, CausalContext(:SmokingContext, :obs, :do, Dict{Symbol, Any}()),
```

```
println("Name:              ", cd.name)
println("Conditioning Kan:  ", cd.conditioning_kan)
println("Intervention Kan:  ", cd.intervention_kan)
```

```
Name:              SmokingCancer
Conditioning Kan:  condition
Intervention Kan:  intervene
```

This declaration encodes the causal reasoning structure: intervention (left Kan) pushes information forward through the causal model, while conditioning (right Kan) completes partial observations.

Inspecting the Diagram

The `CausalDiagram` wraps a full FunctorFlow `Diagram` in `.base_diagram`. Let's inspect its structure.

```
D = cd.base_diagram
```

```
Diagram :SmokingCancer
  Objects:
    Observations::observations
    CausalStructure::causal_structure
    InterventionalState::interventional_state
    ConditionalState::conditional_state
  Operations:
    intervene = Σ(Observations, along=CausalStructure, target=InterventionalState, reducer=:su
    condition = Δ(Observations, along=CausalStructure, target=ConditionalState, reducer=:first
  Ports:
```

```
    → observations (observations)
    → causal_structure (causal_structure)
    ← intervention (interventional_state)
    ← conditioning (conditional_state)
```

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

```
Objects:    [:Observations, :CausalStructure, :InterventionalState, :ConditionalState]
Operations: [:intervene, :condition]
Ports:      [:observations, :causal_structure, :intervention, :conditioning]
```

The builder creates:

- Objects for observations, causal structure, interventional state, and conditional state
- A left Kan extension (`:intervene`) for the intervention — pushing forward the do-operator
- A right Kan extension (`:condition`) for conditioning — completing partial observations
- Ports exposing inputs (observations, causal structure) and outputs (intervention, conditioning)

This diagrammatic encoding means the causal model is not just metadata — it is an executable categorical structure.

Causal Transport

`causal_transport` constructs a transport diagram between two causal regimes. It includes both source and target causal diagrams under namespaces, and optionally a density ratio morphism (the RN layer) for reweighting.

Basic transport

```
# Create a second causal diagram for a target regime
ctx_target = CausalContext(:TargetContext;
    observational_regime=:obs,
    interventional_regime=:do
)
cd_target = build_causal_diagram(:TargetRegime; context=ctx_target)

transport_D = causal_transport(cd, cd_target; name=:BasicTransport)
```

```
Diagram :BasicTransport
  Objects:
    source_regime__Observations::observations
    source_regime__CausalStructure::causal_structure
    source_regime__InterventionalState::interventional_state
    source_regime__ConditionalState::conditional_state
    target_regime__Observations::observations
    target_regime__CausalStructure::causal_structure
    target_regime__InterventionalState::interventional_state
    target_regime__ConditionalState::conditional_state
  Operations:
    source_regime__intervene = Σ(source_regime__Observations, along=source_regime__CausalStruc
    source_regime__condition = Δ(source_regime__Observations, along=source_regime__CausalStruc
    target_regime__intervene = Σ(target_regime__Observations, along=target_regime__CausalStruc
    target_regime__condition = Δ(target_regime__Observations, along=target_regime__CausalStruc
  Ports:
    → source_obs (observations)
    → target_obs (observations)
```

```
println("Transport objects:    ", collect(keys(transport_D.objects)))
println("Transport operations: ", collect(keys(transport_D.operations)))
```

```
Transport objects:    [:source_regime__Observations, :source_regime__CausalStructure, :source_
Transport operations: [:source_regime__intervene, :source_regime__condition, :target_regime__i
```

Transport with density ratio

The density ratio $\rho = p_{\text{do}}(y)/p_{\text{obs}}(y)$ enables computing interventional expectations from observational data. When a `density_ratio` function is provided, a `:DensityRatio` object and `:rn_reweight` morphism are added to the transport diagram.

```
transport_rn = causal_transport(cd, cd_target;
    density_ratio=x -> x,
    name=:RNTransport
)
println("Transport with RN objects: ", collect(keys(transport_rn.objects)))
println("Transport with RN ops:     ", collect(keys(transport_rn.operations)))
```

```
Transport with RN objects: [:source_regime__Observations, :source_regime__CausalStructure, :so
Transport with RN ops:     [:source_regime__intervene, :source_regime__condition, :target_regi
```

The density ratio morphism acts as the RN (Radon-Nikodym) layer that bridges the observational and interventional regimes.

Interventional Expectation

`interventional_expectation` computes $\mathbb{E}_{\text{do}}[Y]$ from observational data by executing the causal diagram's left Kan (intervention) and right Kan (conditioning) extensions. We provide a `Dict` containing `:Observations` and `:CausalStructure`.

```
obs_data = Dict(
    :Observations ⇒ Dict(:a ⇒ 1.0, :b ⇒ 2.0, :c ⇒ 3.0),
    :CausalStructure ⇒ Dict((:a, :b) ⇒ true, (:b, :c) ⇒ true)
)
result = interventional_expectation(cd, obs_data)
println("Intervention result: ", result[:intervention])
println("Conditioning result: ", result[:conditioning])
println("All values:          ", result[:all_values])
```

```
Intervention result: Dict{Any, Any}()
Conditioning result: Dict{Any, Any}()
All values:          Dict{Symbol, Any}(:Observations => Dict(:a => 1.0, :b => 2.0, :c => 3.0),
```

When a `density_ratio_fn` is supplied, each observation is reweighted by $\rho(y) = p_{\text{do}}(y)/p_{\text{obs}}(y)$ before aggregation — this is the importance-weighting step that turns observational averages into interventional expectations.

```
# Simple density ratio: upweight each observation by 1.5×
result_rn = interventional_expectation(cd, obs_data;
    density_ratio_fn = v -> 1.5
)
println("Reweighted intervention: ", result_rn[:intervention])
println("Reweighted conditioning: ", result_rn[:conditioning])
```

```
Reweighted intervention: Dict{Any, Any}(:a => 1.5, :b => 3.0, :c => 4.5)
Reweighted conditioning: Dict{Any, Any}()
```

Identifiability

Before estimating a causal effect we need to know whether it is identifiable — whether the interventional distribution can be expressed purely in terms of observational quantities. `is_identifiable` inspects the diagram structure and applies do-calculus rules.

```
ident = is_identifiable(cd, :Y)
println("Identifiable: ", ident.identifiable)
println("Rule applied: ", ident.rule)
println("Reasoning:    ", ident.reasoning)
```

```
Identifiable: true
Rule applied: adjustment
Reasoning:    Both Kan extensions share source and causal structure; adjustment formula applie
door criterion
```

The function returns a `NamedTuple` with:

- `identifiable` — whether the effect is identifiable from the diagram
- `rule` — which do-calculus rule was applied (`:adjustment`, `:no_causal_path`, etc.)
- `reasoning` — a human-readable explanation

When the diagram has both left and right Kan extensions sharing a common source and causal structure, the adjustment formula (back-door criterion) applies:

```
ident2 = is_identifiable(cd, :InterventionalState)
println("Identifiable: ", ident2.identifiable)
println("Rule applied: ", ident2.rule)
```

```
Identifiable: true
Rule applied: adjustment
```

We can also supply an `observed` argument listing which variables are observed:

```
ident3 = is_identifiable(cd, :Y; observed=[:Observations])
println("Identifiable: ", ident3.identifiable)
println("Reasoning:    ", ident3.reasoning)
```

```
Identifiable: true
Reasoning:    Both Kan extensions share source and causal structure; adjustment formula applie
door criterion
```

Radon-Nikodym Density Ratio Transport

The density ratio morphism in `causal_transport` is the categorical encoding of the Radon-Nikodym derivative $\frac{dP_{\text{do}}}{dP_{\text{obs}}}$. It bridges two regimes: the observational regime (where we collect data) and the interventional regime (where we want to reason).

The transport diagram with a density ratio adds three components:

1. **`:DensityRatio`** — an object representing the ratio $\rho(y)$
2. **`:rn_reweight`** — a morphism from source observations to the density ratio, applying the supplied function
3. **`:ReweightedEstimate`** — an object holding the importance-weighted result

```
# A density ratio that doubles the weight of each observation
rn_fn = x -> 2.0 * x
transport_rn_detail = causal_transport(cd, cd_target;
    density_ratio=rn_fn,
    name=:DetailedRNTransport
)
println("Objects:    ", collect(keys(transport_rn_detail.objects)))
println("Operations: ", collect(keys(transport_rn_detail.operations)))
println("Ports:      ", collect(keys(transport_rn_detail.ports)))
```

```
Objects:    [:source_regime__Observations, :source_regime__CausalStructure, :source_regime__In
Operations: [:source_regime__intervene, :source_regime__condition, :target_regime__intervene,
Ports:      [:density_ratio, :reweighted, :source_obs, :target_obs]
```

The `:rn_reweight` morphism stores the density ratio function as its implementation, making the RN layer a first-class part of the diagram rather than an external post-processing step. This is essential for composability — transports can be chained or nested, and the density ratio propagates correctly.

```
# Inspect the RN morphism metadata
rn_op = transport_rn_detail.operations[:rn_reweight]
println("RN morphism source: ", rn_op.source)
println("RN morphism target: ", rn_op.target)
println("Causal role:        ", rn_op.metadata[:causal_role])
```

```
RN morphism source: source_regime__Observations
RN morphism target: DensityRatio
Causal role:        rn_layer
```

Causal Verification

FunctorFlow's `verify` function checks the structural properties of universal constructions. While `verify` is primarily designed for pullbacks, pushouts, products, and coproducts, we can verify the transport diagram by compiling it and checking that its structure is well-formed.

```
# Verify transport diagram structure
transport_v = causal_transport(cd, cd_target;
    density_ratio=x -> x,
    name=:VerifiableTransport
)

# Check that essential causal components are present
has_rn = haskey(transport_v.objects, :DensityRatio)
has_reweight = haskey(transport_v.operations, :rn_reweight)
has_apply = haskey(transport_v.operations, :apply_weights)
has_src = any(startswith(String(k), "source_regime") for k in keys(transport_v.objects))
has_tgt = any(startswith(String(k), "target_regime") for k in keys(transport_v.objects))

println("Has DensityRatio object:    ", has_rn)
println("Has rn_reweight morphism:   ", has_reweight)
println("Has apply_weights morphism: ", has_apply)
println("Has source regime objects:  ", has_src)
println("Has target regime objects:  ", has_tgt)
println("All causal checks passed:   ", all([has_rn, has_reweight, has_apply, has_src, has_tgt
```

```
Has DensityRatio object:    true
Has rn_reweight morphism:   true
Has apply_weights morphism: true
Has source regime objects:  true
Has target regime objects:  true
All causal checks passed:   true
```

Counterfactual Reasoning

Counterfactuals combine both directions of Kan extension. The question "What would have happened to Y if X had been x, given that we observed $X = x'$?" involves:

1. Right Kan (conditioning): update beliefs given the observation $X = x'$
2. Left Kan (intervention): intervene to set $X = x$ in the updated model

We can build a causal diagram that includes both Kan directions:

```
ctx_cf = CausalContext(:CounterfactualContext;
    observational_regime=:observed,
    interventional_regime=:counterfactual
)

cd_cf = build_causal_diagram(:CounterfactualModel; context=ctx_cf)
println("Counterfactual diagram objects: ", collect(keys(cd_cf.base_diagram.objects)))
println("Counterfactual diagram ops:    ", collect(keys(cd_cf.base_diagram.operations)))
```

```
Counterfactual diagram objects: [:Observations, :CausalStructure, :InterventionalState, :Condi
Counterfactual diagram ops:    [:intervene, :condition]
```

The counterfactual diagram contains both left and right Kan extensions, encoding the two-step "observe then intervene" process.

We can also compose the counterfactual model with a transport:

```
transport_cf = causal_transport(cd, cd_cf; name=:CounterfactualTransport)
println("Counterfactual transport objects: ", collect(keys(transport_cf.objects)))
```

```
Counterfactual transport objects: [:source_regime__Observations, :source_regime__CausalStructu
```

Interpretation

The RN-Kan-Do-Calculus provides a principled categorical foundation for causal reasoning:

- Right Kan extension = Conditioning. Given observed data, the right Kan extension computes the "best completion" — the most general way to extend partial information. This is the categorical analog of Bayesian conditioning $P(Y \mid X)$.
- Left Kan extension = Intervention. The do-operator actively severs incoming causal arrows and pushes information forward. The left Kan extension is a colimit — it aggregates over all ways the intervention can propagate. This is the categorical analog of $P(Y \mid \text{do}(X))$.
- Counterfactuals = Composition. By composing right (condition) and left (intervene) Kan extensions, we get the counterfactual $P(Y_x \mid X = x')$: first absorb the evidence, then perform the hypothetical intervention.

This framework unifies Pearl's ladder of causation (association, intervention, counterfactual) within a single diagrammatic language. The transport rules (backdoor, frontdoor, IV) are functorial transformations that map causal diagrams to statistical estimands while preserving the categorical structure.