Kan Extensions

Universal aggregation (Σ) and completion (Δ)

Simon Frost

Table of contents

Introduction . . . 1
Setup . . . 2
Left Kan (Σ): Universal Aggregation . . . 2
Right Kan (Δ): Universal Completion . . . 3
Custom Reducers . . . 3
Reducer Gallery . . . 4
:sum — Sum of values . . . 5
:mean — Arithmetic mean . . . 5
:concat — Concatenation . . . 5
:majority — Most common value . . . 5
:set_union — Union of collections . . . 6
:tuple — Collect into tuples . . . 6
:first_non_null — First non-nothing value . . . 7
The Duality . . . 7

Introduction

Kan extensions are often called "the most universal construction in category theory." In Functor-Flow, they serve as the foundational operation for two dual patterns:

- Left Kan extension (Σ) = universal aggregation. It pushes forward values along a relation and reduces them. This single abstraction subsumes attention, pooling, message passing, and context fusion.
- Right Kan extension (Δ) = universal completion. It fills in missing or partial values along a compatibility relation. This covers denoising, repair, and reconciliation.

FunctorFlow uses the mathematical symbols Σ and Δ for these operations, following the convention from algebraic Julia packages like CategoricalProjectionModels.jl where left and right Kan extensions are denoted as sigma and delta pushforwards.

Both patterns take a source value, a relation (the "along" functor), and a reducer that determines how grouped values are combined.

Setup

```
using FunctorFlow
```

Left Kan (Σ): Universal Aggregation

A left Kan extension groups source values by a relation and reduces each group. Consider a set of node values and an incidence relation defining neighborhoods:

```
D = Diagram(:LeftKanDemo)
add_object!(D, :Values; kind=:messages)
add_object!(D, :Incidence; kind=:relation)
Σ(D, :Values; along=:Incidence, reducer=:sum, name=:aggregate)

compiled = compile_to_callable(D)

values = Dict(:a ⇒ 1, :b ⇒ 2, :c ⇒ 3, :d ⇒ 4)
incidence = Dict(:x ⇒ [:a, :b], :y ⇒ [:b, :c, :d], :z ⇒ [:a])

result = FunctorFlow.run(compiled, Dict(:Values ⇒ values, :Incidence ⇒ incidence))
println("Sum aggregation: ", result.values[:aggregate])
```

```
Sum aggregation: Dict(:y => 9, :z => 1, :x => 3)
```

Target `:x` receives `1 + 2 = 3`, target `:y` receives `2 + 3 + 4 = 9`, and target `:z` receives `1`.

This is the same pattern as:

- Attention: values are value vectors, relation is the attention matrix
- Pooling: values are features, relation groups features into pools
- Message passing: values are node messages, relation is the adjacency

Right Kan (Δ): Universal Completion

A right Kan extension completes partial data using a compatibility relation. Where Σ aggregates many values into one, Δ fills in missing values from compatible sources.

```
D2 = Diagram(:RightKanDemo)
add_object!(D2, :PartialValues; kind=:partial)
add_object!(D2, :Compatibility; kind=:relation)
Δ(D2, :PartialValues; along=:Compatibility, name=:complete)

compiled2 = compile_to_callable(D2)

partial = Dict(:a ⇒ nothing, :b ⇒ 42, :c ⇒ nothing, :d ⇒ 99)
compat = Dict(:a ⇒ [:b, :c], :c ⇒ [:d, :a])

result2 = FunctorFlow.run(compiled2, Dict(:PartialValues ⇒ partial, :Compatibility ⇒ compat))
println("Completed: ", result2.values[:complete])
```

```
Completed: Dict{Any, Any}(:a => 42, :c => 99)
```

For target `:a`, the compatible sources are `:b` (42) and `:c` (nothing). The `:first_non_null` reducer returns 42. For `:c`, the compatible sources are `:d` (99) and `:a` (nothing), yielding 99.

Custom Reducers

You can write and bind custom reducers. A reducer is a function with the signature `(source_values, relation, metadata) -> result`, where `source_values` is a `Dict` of source key-value pairs, `relation` maps target keys to vectors of source keys, and the result is a `Dict` of target key-value pairs.

```
D3 = Diagram(:CustomReducer)
add_object!(D3, :Vals; kind=:messages)
add_object!(D3, :Rel; kind=:relation)
Σ(D3, :Vals; along=:Rel, reducer=:weighted_sum, name=:weighted_sum)

# Custom reducer: weighted sum where weight = position index
function my_weighted_sum(source_values, relation, metadata)
    relation_dict = FunctorFlow.normalize_relation(relation)
    result = Dict{Any, Any}()
    for (target, sources) in relation_dict
```

```
        total = 0.0
        for (i, src) in enumerate(sources)
            val = get(source_values, src, 0)
            total += i * val
        end
        result[target] = total
    end
    return result
end

bind_reducer!(D3, :weighted_sum, my_weighted_sum)

compiled3 = compile_to_callable(D3)
result3 = FunctorFlow.run(compiled3, Dict(
    :Vals => Dict(:a => 10, :b => 20),
    :Rel => Dict(:x => [:a, :b])
))
println("Weighted sum: ", result3.values[:weighted_sum])
```

```
Weighted sum: Dict{Any, Any}(:x => 50.0)
```

With weights `[1, 2]` and values `[10, 20]`, the result is `1*10 + 2*20 = 50`.

Reducer Gallery

FunctorFlow ships with 7 built-in reducers. Let's demonstrate each one using the same source data and relation:

```
values = Dict(:a => 10, :b => 20, :c => 30)
relation = Dict(:x => [:a, :b, :c], :y => [:a])

function demo_reducer(reducer_sym)
    D = Diagram(Symbol(:demo_, reducer_sym))
    add_object!(D, :V; kind=:messages)
    add_object!(D, :R; kind=:relation)
    Σ(D, :V; along=:R, reducer=reducer_sym, name=:result)
    compiled = compile_to_callable(D)
    result = FunctorFlow.run(compiled, Dict(:V => values, :R => relation))
    return result.values[:result]
end
```

```
demo_reducer (generic function with 1 method)
```

:sum — Sum of values

```
println("sum: ", demo_reducer(:sum))
```

```
sum: Dict(:y => 10, :x => 60)
```

:mean — Arithmetic mean

```
println("mean: ", demo_reducer(:mean))
```

```
mean: Dict(:y => 10.0, :x => 20.0)
```

:concat — Concatenation

```
# concat works on strings and vectors
values_str = Dict(:a ⇒ "hello", :b ⇒ " ", :c ⇒ "world")

D_concat = Diagram(:ConcatDemo)
add_object!(D_concat, :V; kind=:messages)
add_object!(D_concat, :R; kind=:relation)
Σ(D_concat, :V; along=:R, reducer=:concat, name=:result)
compiled_concat = compile_to_callable(D_concat)
result_concat = FunctorFlow.run(compiled_concat, Dict(
    :V ⇒ values_str,
    :R ⇒ Dict(:x ⇒ [:a, :b, :c])
))
println("concat: ", result_concat.values[:result])
```

```
concat: Dict{Any, Any}(:x => "hello world")
```

:majority — Most common value

```
values_vote = Dict(:a ⇒ "yes", :b ⇒ "no", :c ⇒ "yes", :d ⇒ "yes")
relation_vote = Dict(:x ⇒ [:a, :b, :c, :d])

D_maj = Diagram(:MajorityDemo)
add_object!(D_maj, :V; kind=:messages)
add_object!(D_maj, :R; kind=:relation)
Σ(D_maj, :V; along=:R, reducer=:majority, name=:result)
compiled_maj = compile_to_callable(D_maj)
result_maj = FunctorFlow.run(compiled_maj, Dict(:V ⇒ values_vote, :R ⇒ relation_vote))
println("majority: ", result_maj.values[:result])
```

```
majority: Dict{Any, Any}(:x => "yes")
```

:set_union — Union of collections

```
values_sets = Dict(:a ⇒ Set([1, 2]), :b ⇒ Set([2, 3]), :c ⇒ Set([4]))
relation_sets = Dict(:x ⇒ [:a, :b, :c])

D_union = Diagram(:SetUnionDemo)
add_object!(D_union, :V; kind=:messages)
add_object!(D_union, :R; kind=:relation)
Σ(D_union, :V; along=:R, reducer=:set_union, name=:result)
compiled_union = compile_to_callable(D_union)
result_union = FunctorFlow.run(compiled_union, Dict(:V ⇒ values_sets, :R ⇒ relation_sets))
println("set_union: ", result_union.values[:result])
```

```
set_union: Dict{Any, Any}(:x => Set(Any[4, 2, 3, 1]))
```

:tuple — Collect into tuples

```
println("tuple: ", demo_reducer(:tuple))
```

```
tuple: Dict{Symbol, Tuple{Int64, Vararg{Int64}}}(:y => (10,), :x => (10, 20, 30))
```

:first_non_null — First non-nothing value

```
values_partial = Dict(:a ⇒ nothing, :b ⇒ 42, :c ⇒ nothing)
relation_partial = Dict(:x ⇒ [:a, :b, :c])

D_fnn = Diagram(:FirstNonNullDemo)
add_object!(D_fnn, :V; kind=:partial)
add_object!(D_fnn, :R; kind=:relation)
Σ(D_fnn, :V; along=:R, reducer=:first_non_null, name=:result)
compiled_fnn = compile_to_callable(D_fnn)
result_fnn = FunctorFlow.run(compiled_fnn, Dict(:V ⇒ values_partial, :R ⇒ relation_partial))
println("first_non_null: ", result_fnn.values[:result])
```

```
first_non_null: Dict{Any, Any}(:x => 42)
```

The Duality

Left and right Kan extensions are dual. Applied to the same data, they answer different questions:

- Left Kan (Σ) (aggregation): "What is the combined value at each target?"
- Right Kan (Δ) (completion): "What value should fill each missing slot?"

```
D_dual = Diagram(:Duality)
add_object!(D_dual, :Data; kind=:messages)
add_object!(D_dual, :Relation; kind=:relation)

Σ(D_dual, :Data; along=:Relation, reducer=:sum, name=:aggregated)
Δ(D_dual, :Data; along=:Relation, name=:completed)

compiled_dual = compile_to_callable(D_dual)

data = Dict(:a ⇒ 10, :b ⇒ nothing, :c ⇒ 30)
rel = Dict(:x ⇒ [:a, :b], :y ⇒ [:b, :c])

result_dual = FunctorFlow.run(compiled_dual, Dict(:Data ⇒ data, :Relation ⇒ rel))
println("Left Kan Σ (sum): ", result_dual.values[:aggregated])
println("Right Kan Δ (first_non_null): ", result_dual.values[:completed])
```

```
Left Kan Σ (sum): Dict(:y => 30, :x => 10)
Right Kan Δ (first_non_null): Dict{Any, Any}(:y => 30, :x => 10)
```

The Σ operator sums available values per target group, while Δ fills in missing values from compatible neighbors. Together, they form the predict-and-repair duality at the heart of FunctorFlow's architectural patterns.