Diagram Composition

Composing sub-diagrams with ports and adapters

Simon Frost

Table of contents

Introduction

FunctorFlow diagrams can be composed hierarchically. A parent diagram can include child sub-diagrams using `include!()`, which embeds all objects and operations under a namespace. Ports provide stable typed interfaces that survive composition, and adapters handle principled type coercion when port types don't match. This enables modular, reusable architectural patterns.

Setup

```
using FunctorFlow
```

Ports

Ports expose semantic interfaces on a diagram. They declare which objects or operations are externally accessible, along with their direction and type. Think of ports as the typed "sockets" of a diagram component.

```
D = Diagram(:Encoder)
add_object!(D, :Tokens; kind=:messages)
add_object!(D, :Neighbors; kind=:relation)
add_object!(D, :Output; kind=:contextualized_messages)
Σ(D, :Tokens; along=:Neighbors, target=:Output, reducer=:sum, name=:aggregate)

# Expose ports
expose_port!(D, :input, :Tokens; direction=INPUT, port_type=:messages)
expose_port!(D, :relation, :Neighbors; direction=INPUT, port_type=:relation)
expose_port!(D, :output, :Output; direction=OUTPUT, port_type=:contextualized_messages)

d = as_dict(to_ir(D))
println("Ports: ", collect(keys(d["ports"])))
```

```
Ports: [1, 2, 3]
```

You can also inspect individual ports:

```
p = get_port(D, :input)
println("Port :input → ref=", p.ref, ", direction=", p.direction, ", type=", p.port_type)
```

```
Port :input → ref=Tokens, direction=INPUT, type=messages
```

Pre-built blocks from the block library already have ports defined:

```
D_ket = ket_block()
d_ket = as_dict(to_ir(D_ket))
println("KET block ports: ", collect(keys(d_ket["ports"])))
```

```
KET block ports: [1, 2, 3]
```

Including Sub-Diagrams

Use `include!()` to embed one diagram inside another. All objects and operations from the child diagram are copied into the parent under a namespace prefix.

```
# Create a parent diagram
parent = Diagram(:Pipeline)
add_object!(parent, :RawInput; kind=:input)

# Create a child encoder diagram
encoder = Diagram(:Encoder)
add_object!(encoder, :Values; kind=:messages)
add_object!(encoder, :Incidence; kind=:relation)
Σ(encoder, :Values; along=:Incidence, reducer=:sum, name=:aggregate)

# Include the encoder in the parent under namespace :enc
inc = include!(parent, encoder; namespace=:enc)

d_parent = as_dict(to_ir(parent))
println("Parent objects: ", collect(keys(d_parent["objects"])))
println("Parent operations: ", collect(keys(d_parent["operations"])))
```

```
Parent objects: [1, 2, 3]
Parent operations: [1]
```

All child elements are prefixed with the namespace: `:enc__Values`, `:enc__Incidence`, `:enc__aggregate`.

The returned `IncludedDiagram` object provides helpers for referencing namespaced elements:

```
println("Namespaced object: ", object_ref(inc, :Values))
println("Namespaced operation: ", operation_ref(inc, :aggregate))
```

```
Namespaced object: enc__Values
Namespaced operation: enc__aggregate
```

Object Aliases

When including a sub-diagram, you can wire parent objects into child slots using `object_aliases`. This maps child object names to existing parent object names, so they share the same data at runtime.

```
parent2 = Diagram(:AliasDemo)
add_object!(parent2, :SharedData; kind=:messages)
add_object!(parent2, :SharedRelation; kind=:relation)

child = Diagram(:SubBlock)
add_object!(child, :Input; kind=:messages)
add_object!(child, :Rel; kind=:relation)
Σ(child, :Input; along=:Rel, reducer=:sum, name=:agg)

# Alias child's :Input to parent's :SharedData, and child's :Rel to :SharedRelation
inc2 = include!(parent2, child; namespace=:sub,
                object_aliases=Dict(:Input ⇒ :SharedData, :Rel ⇒ :SharedRelation))

# The aliased objects point to parent objects
println("Input maps to: ", object_ref(inc2, :Input))
println("Rel maps to: ", object_ref(inc2, :Rel))

d2 = as_dict(to_ir(parent2))
println("Parent objects: ", collect(keys(d2["objects"])))
```

```
Input maps to: SharedData
Rel maps to: SharedRelation
Parent objects: [1, 2]
```

Adapters

When composing diagrams, port types may not align. For example, one block's output type might be `:contextualized_messages` while the next block expects `:plan_candidates`. Adapters provide principled type bridges.

Registering Adapters

```
D_adapt = Diagram(:AdapterDemo)
add_object!(D_adapt, :Input; kind=:messages)
add_object!(D_adapt, :Output; kind=:plan_candidates)
add_morphism!(D_adapt, :transform, :Input, :Output)
bind_morphism!(D_adapt, :transform, x -> x)

# Register an adapter for type coercion
```

```
register_adapter!(D_adapt, :msg_to_candidates;
                  source_type=:messages,
                  target_type=:plan_candidates,
                  implementation=x -> x,
                  description="Identity coercion from messages to candidates")

d_adapt = as_dict(to_ir(D_adapt))
println("Adapters: ", length(D_adapt.adapters))
```

```
Adapters: 1
```

Adapter Libraries

FunctorFlow provides a standard adapter library and lets you build custom ones:

```
# The standard library ships with common coercions
std_lib = STANDARD_ADAPTER_LIBRARY
println("Standard library: ", std_lib.name)
println("Adapters: ", length(std_lib.adapters))
for a in std_lib.adapters
    println("  ", a.name, ": ", a.source_type, " → ", a.target_type)
end
```

```
Standard library: standard
Adapters: 3
  context_to_candidates: contextualized_messages → plan_candidates
  plan_candidates_to_plan: plan_candidates → plan
  string_plan_to_plan_steps: plan → plan_steps
```

Using Adapter Libraries

Install an entire adapter library into a diagram with `use_adapter_library!`:

```
D_lib = Diagram(:WithLibrary)
add_object!(D_lib, :X; kind=:contextualized_messages)
add_object!(D_lib, :Y; kind=:plan_candidates)

use_adapter_library!(D_lib, STANDARD_ADAPTER_LIBRARY)
```

```
d_lib = as_dict(to_ir(D_lib))
println("Installed adapters: ", length(D_lib.adapters))
```

```
Installed adapters: 3
```

Coercion

Once adapters are registered, use `coerce!()` to insert a coercion morphism that adapts an object's type:

```
D_coerce = Diagram(:CoerceDemo)
add_object!(D_coerce, :Input; kind=:contextualized_messages)

# Register the adapter
register_adapter!(D_coerce, :ctx_to_candidates;
                  source_type=:contextualized_messages,
                  target_type=:plan_candidates,
                  implementation=x -> x)

# Insert a coercion morphism
coercion_name = coerce!(D_coerce, :Input; to_type=:plan_candidates)
println("Generated coercion morphism: ", coercion_name)

d_coerce = as_dict(to_ir(D_coerce))
println("Operations after coerce: ", collect(keys(d_coerce["operations"])))
```

```
Generated coercion morphism: _coerce_Input_to_plan_candidates
Operations after coerce: [1]
```

Real Example: Building a Pipeline

Let's compose a KET block (prediction via left Kan) and a completion block (repair via right Kan) into a predict-then-repair pipeline.

```
# Create building blocks
predictor = ket_block(; config=KETBlockConfig(
    name=:Predictor,
    reducer=:sum
))
```

```
repairer = completion_block(; config=CompletionBlockConfig(
    name=:Repairer
))

# Create parent pipeline
pipeline = Diagram(:PredictRepairPipeline)
add_object!(pipeline, :InputValues; kind=:messages)
add_object!(pipeline, :PredictRelation; kind=:relation)
add_object!(pipeline, :RepairRelation; kind=:relation)

# Include predictor, aliasing its Values input to our InputValues
inc_pred = include!(pipeline, predictor; namespace=:predict,
                    object_aliases=Dict(:Values ⇒ :InputValues,
                                        :Incidence ⇒ :PredictRelation))

# Include repairer
inc_repair = include!(pipeline, repairer; namespace=:repair,
                      object_aliases=Dict(:Compatibility ⇒ :RepairRelation))

d_pipeline = as_dict(to_ir(pipeline))
println("Pipeline objects: ", collect(keys(d_pipeline["objects"])))
println("Pipeline operations: ", collect(keys(d_pipeline["operations"])))
```

```
Pipeline objects: [1, 2, 3, 4, 5, 6]
Pipeline operations: [1, 2]
```

Namespacing

When a sub-diagram is included under a namespace, all its elements are prefixed with `namespace__` (double underscore). This prevents name collisions when the same block type is included multiple times.

```
parent3 = Diagram(:MultiEncoder)

encoder1 = ket_block(; config=KETBlockConfig(name=:Enc1))
encoder2 = ket_block(; config=KETBlockConfig(name=:Enc2))

inc1 = include!(parent3, encoder1; namespace=:layer1)
inc2 = include!(parent3, encoder2; namespace=:layer2)
```

```
d3 = as_dict(to_ir(parent3))
println("All objects:")
for name in sort(collect(keys(d3["objects"])))
    println("  ", name)
end
```

```
All objects:
  1
  2
  3
  4
  5
  6
```

Each layer gets its own prefixed copies of `:Values`, `:Incidence`, and `:ContextualizedValues`, so they operate independently. You can wire them together by adding morphisms between the namespaced objects in the parent diagram, or by using object aliases to share objects across layers.