Proof Visualization for Graphical Structures

Bhakti Shah; May 17, 2024, CS Master's Thesis Presentation
Committee: Stuart Kurtz, Robert Rand (Advisor), John Reppy

How do we reason about graphical languages diagrammatically in a proof assistant?

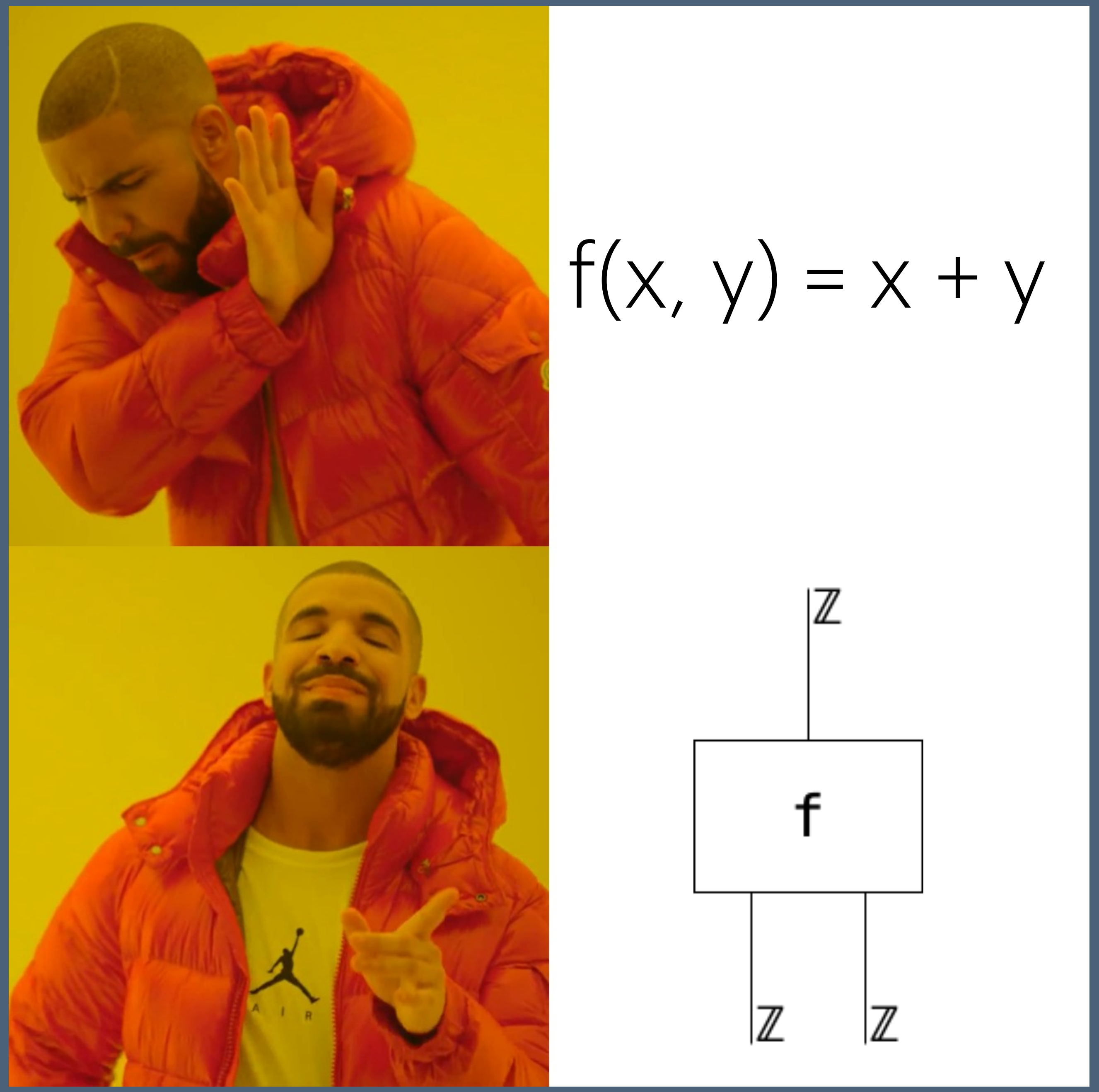
How do we reason about graphical languages diagrammatically in a proof assistant?

How do we reason about graphical languages diagrammatically in a proof assistant?

String diagrams associated with a class of categories.

How do we reason about graphical languages **diagrammatically** in a proof assistant?

How do we reason about graphical languages diagrammatically in a proof assistant?



How do we reason about graphical languages diagrammatically in a proof assistant?

How do we reason about graphical languages diagrammatically in a proof assistant?

The interactive theorem prover, Coq.

???

What is a string diagram? What is a category? What is an interactive theorem prover? What is that diagram?

A simpler setting

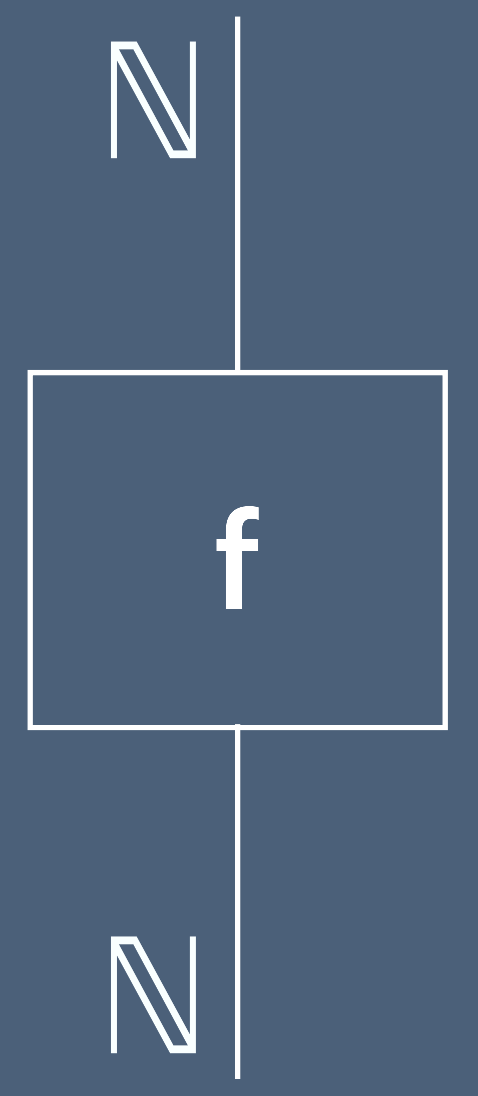
A simpler setting

Process theories

- *Process* = a **box** that takes some number of inputs, and produces some number of outputs.
- *Types* = each input and output is represented by a **wire**, that has a specified type.`

$x \in \mathbb{N}$

$f(x) = x + 1$



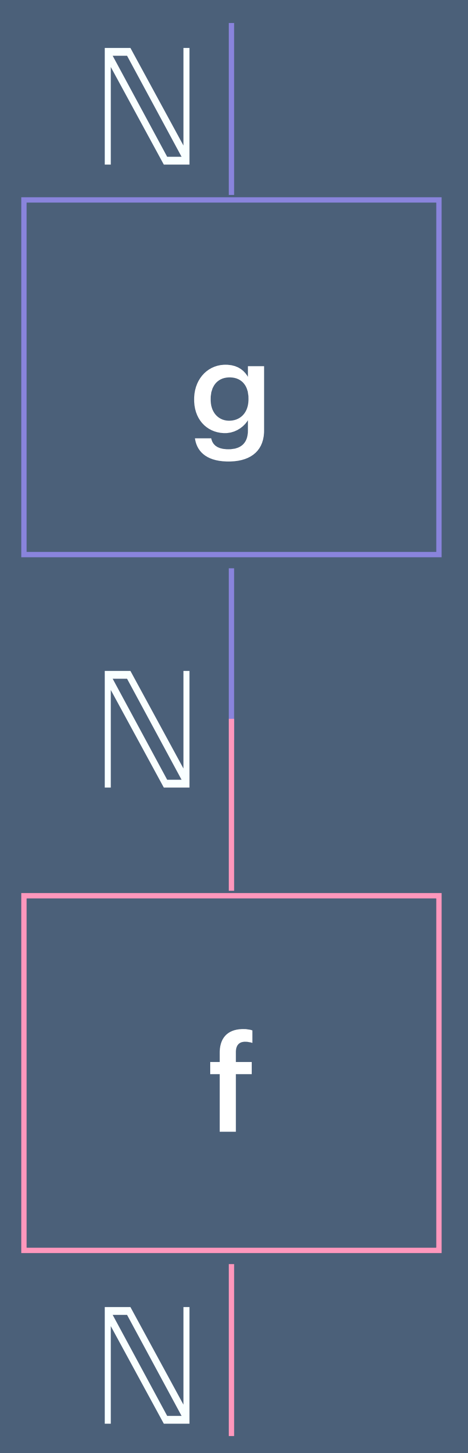
Process theory

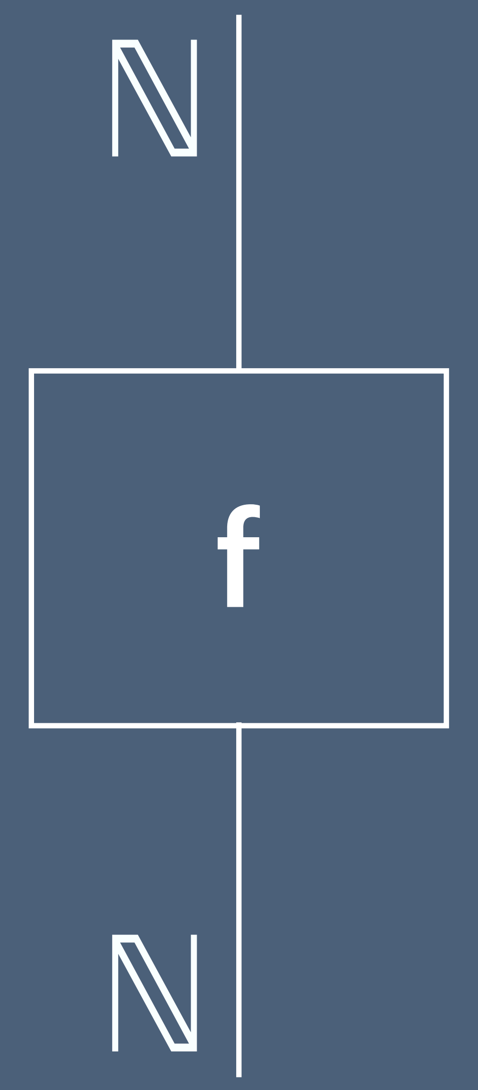
- A process theory is:
  - *T*: a collection of types,
  - *P*: a collection of processes using input and output types from *T*,
  - An operation that can map a diagram of processes in *P* to a singular process in *P*.
  - We also have *identity* wires, which are just boxes that “do nothing”.
  - ***Process theories are graphical languages.***

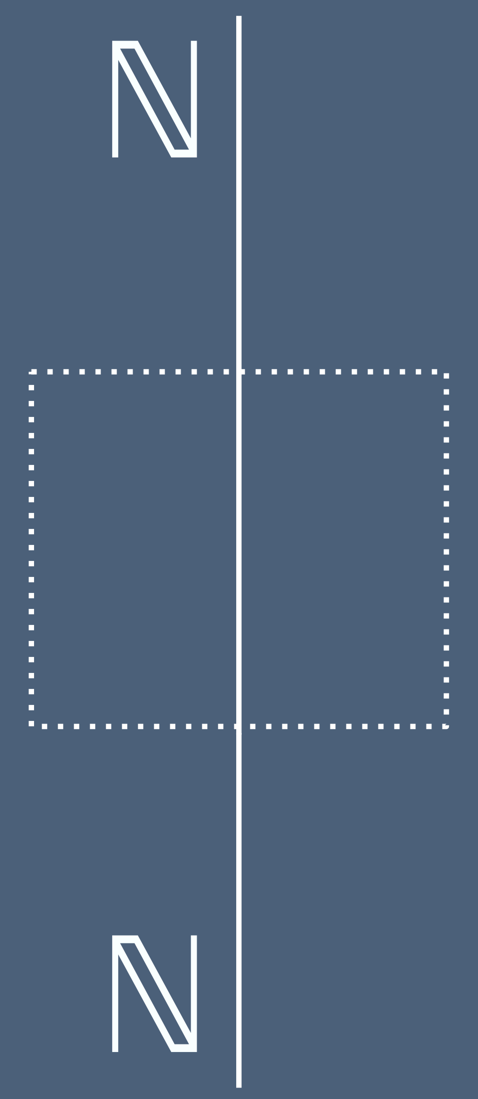
Process theory

Specifically, functions.

- The process theory of functions:
  - T: [ℕ …],
  - P: the set of all functions on types in T.
  - The (associative) function composition operator $\circ$, combining two functions in P to form a unique function in P.
  - The identity wires are just identity functions for every term of types in T.

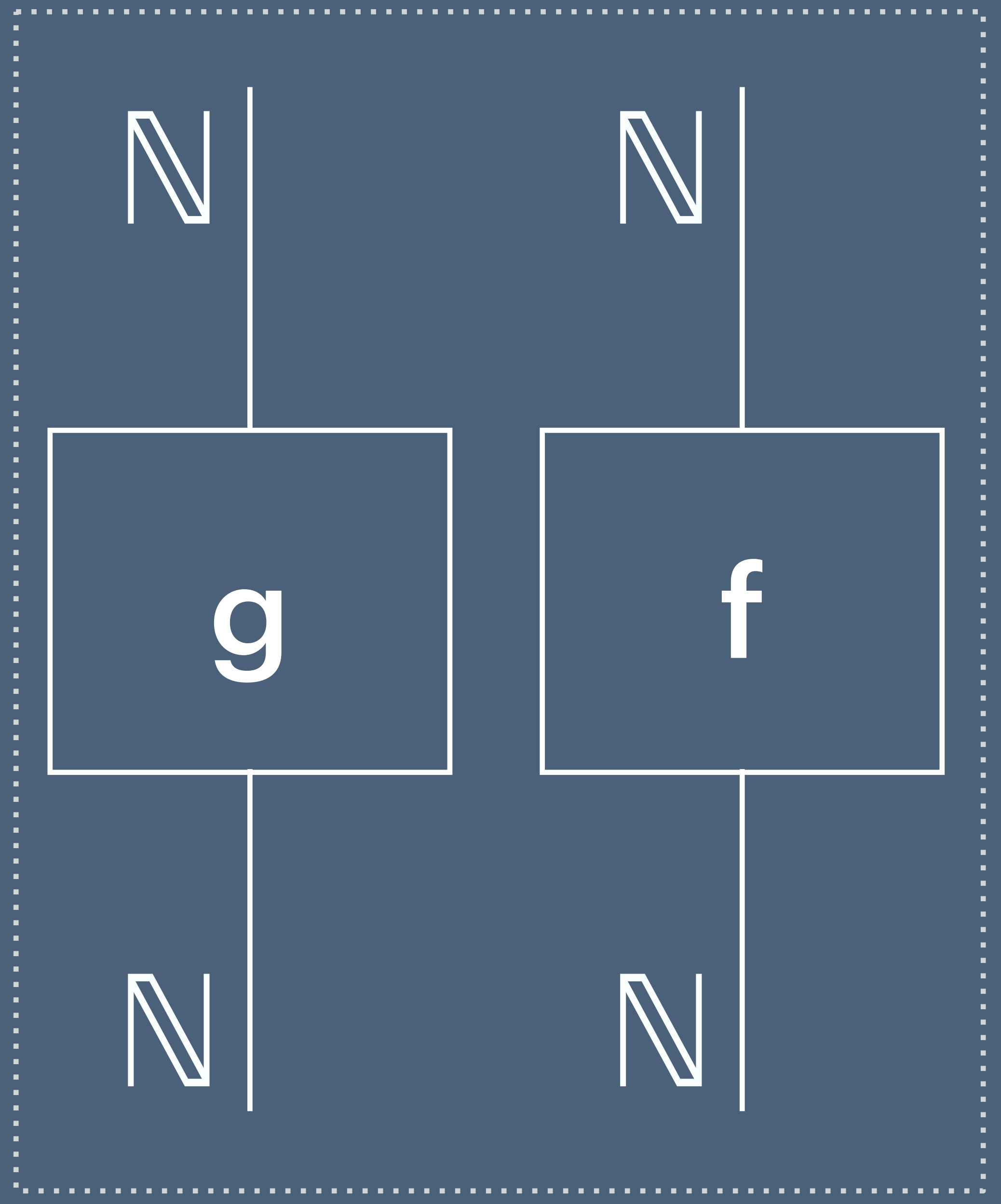




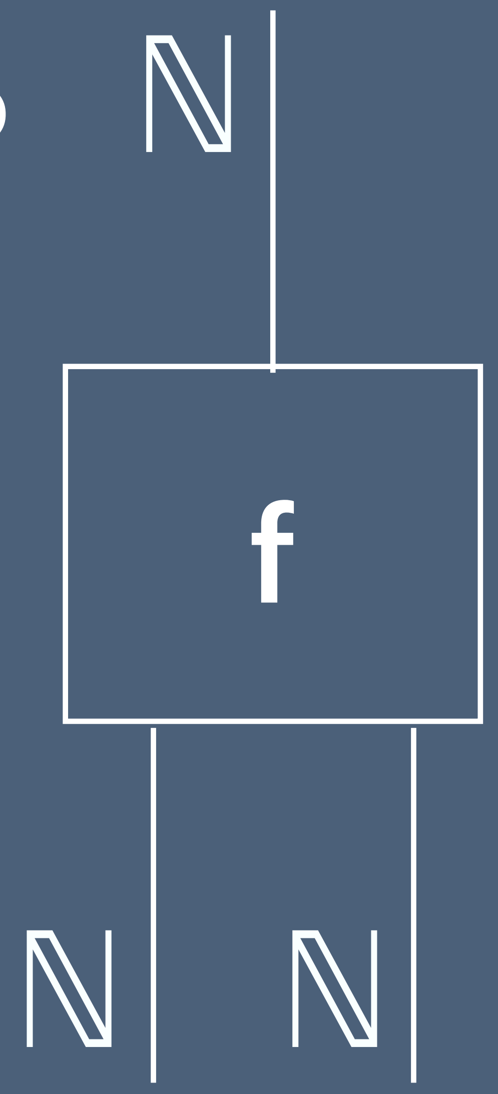


Process theory

Specifically, functions.



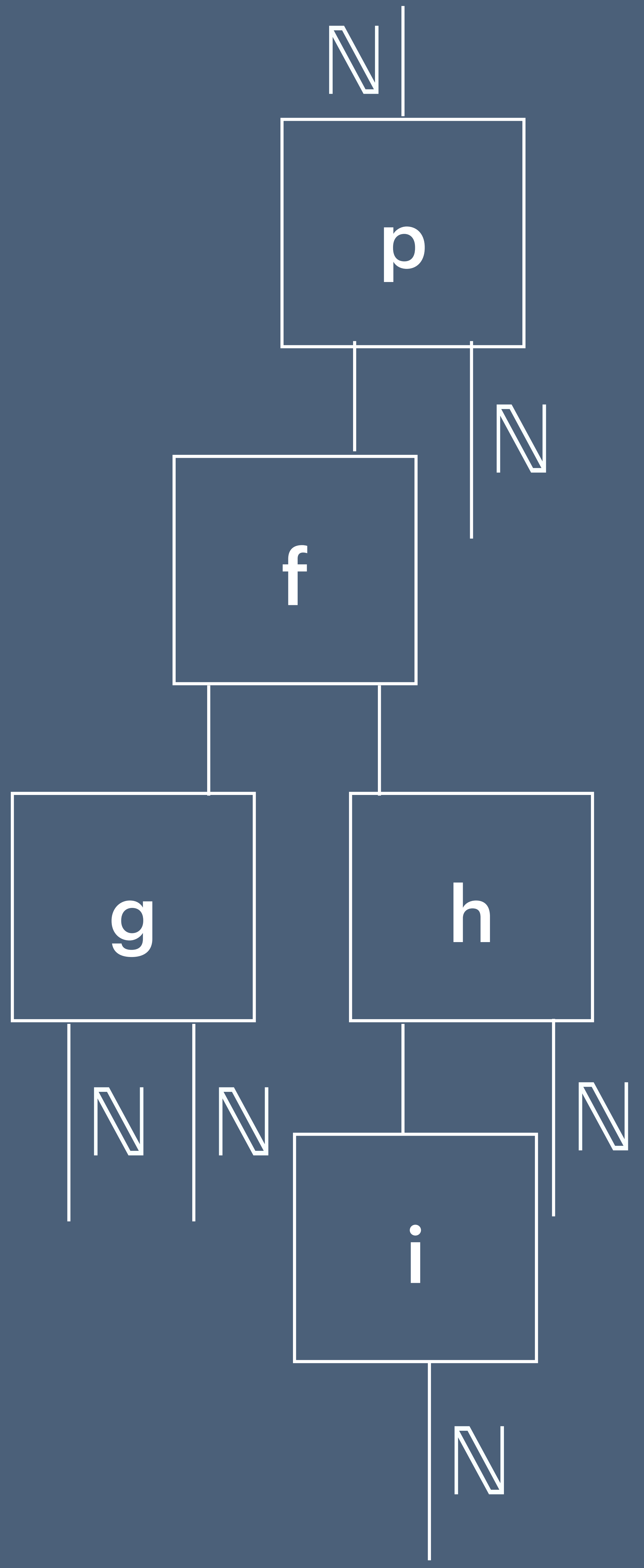
- The process theory of functions:
  - *T*: [ℕ …],
  - *P*: the set of all functions on types in *T*.
  - The (associative) function composition operator ∘, combining two functions in *P* to form a unique function in *P*.
  - The identity wires are just identity functions for every term of types in *T*.
  - The cartesian product operator ⊗, forming a *pair* of functions or inputs.



The stage is set.

Diagrams >>> Text.

Which one is better?



p(f(g(n1, n2), h(i(n3), n4)), n5)

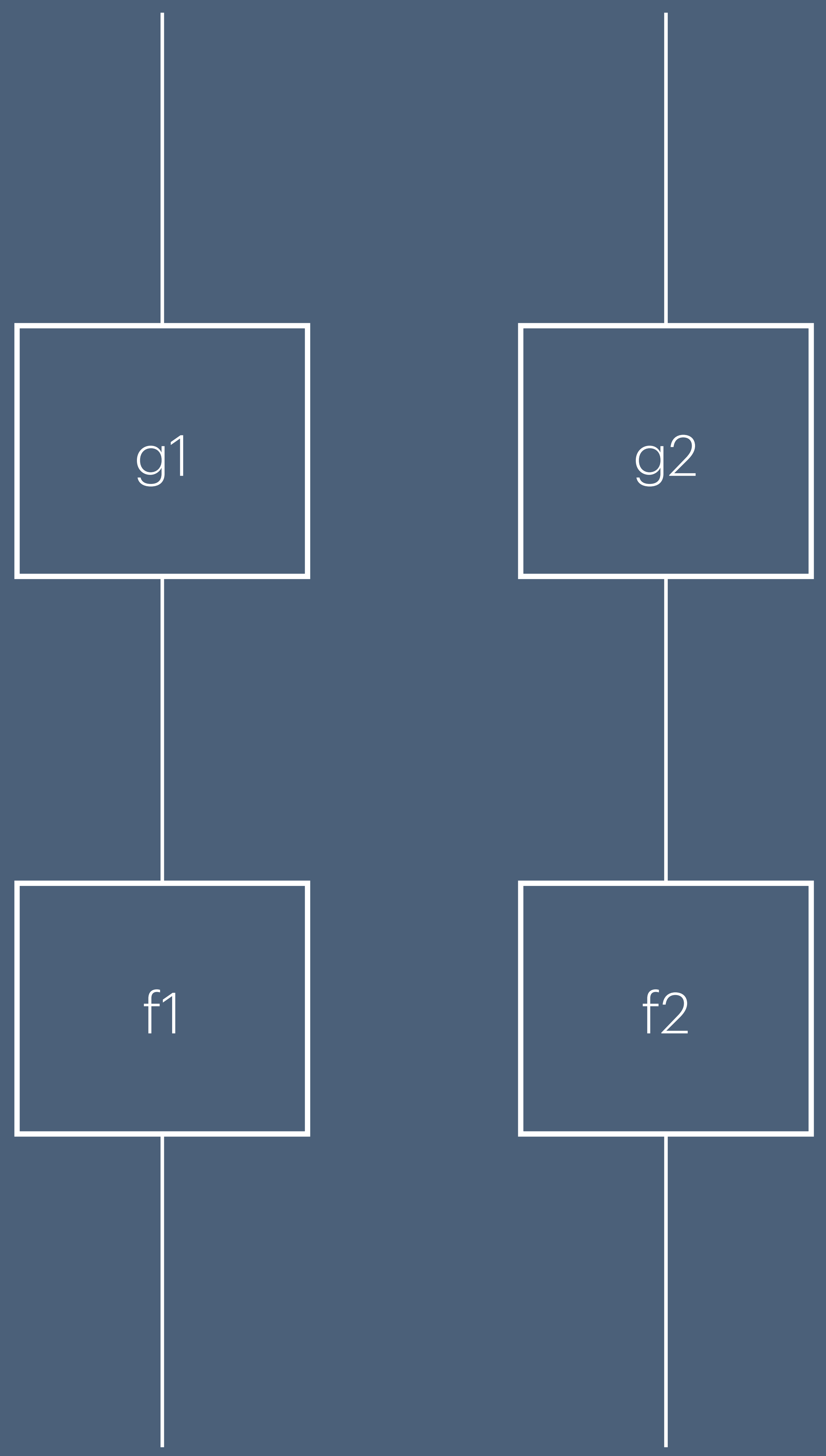
Diagrams can be proofs.

Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$

- $(g1 \otimes g2) \circ (f1 \otimes f2)$

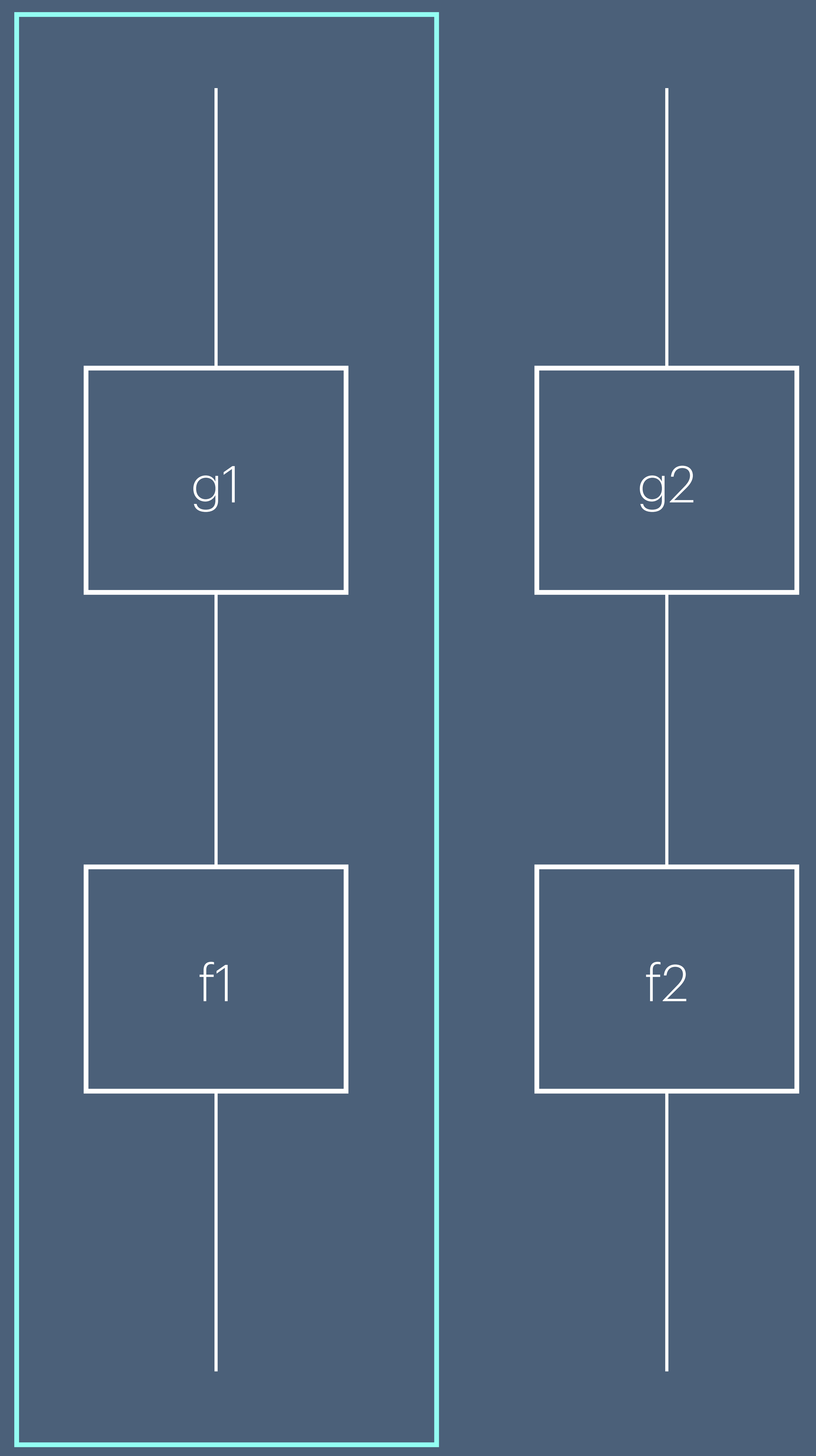


Textually distinct,

Diagrammatically equivalent.

- (g1 ∘ f1) ⊗ (g2 ∘ f2)

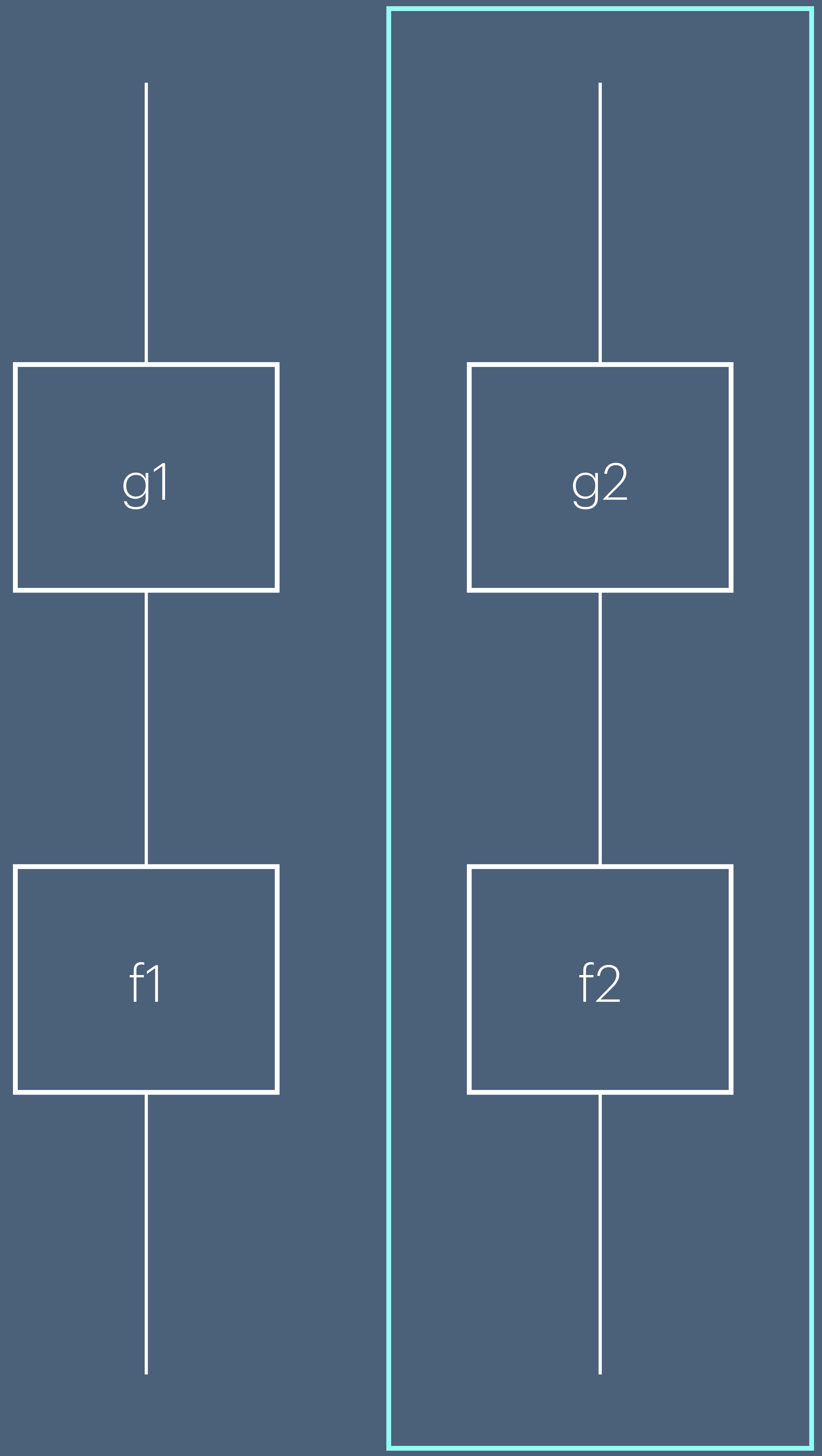
- (g1 ⊗ g2) ∘ (f1 ⊗ f2)



Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$
- $(g1 \otimes g2) \circ (f1 \otimes f2)$

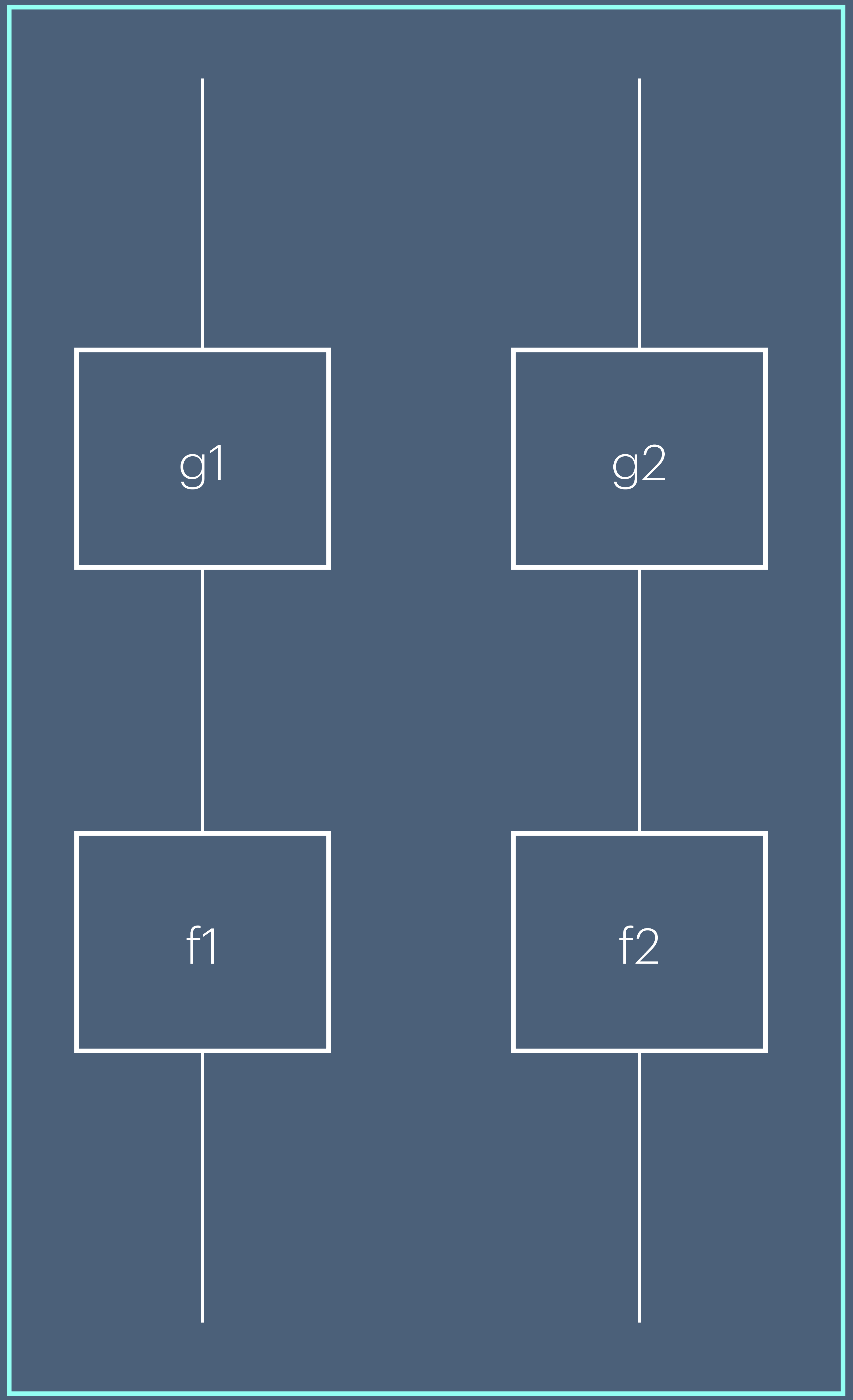


Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$

- $(g1 \otimes g2) \circ (f1 \otimes f2)$

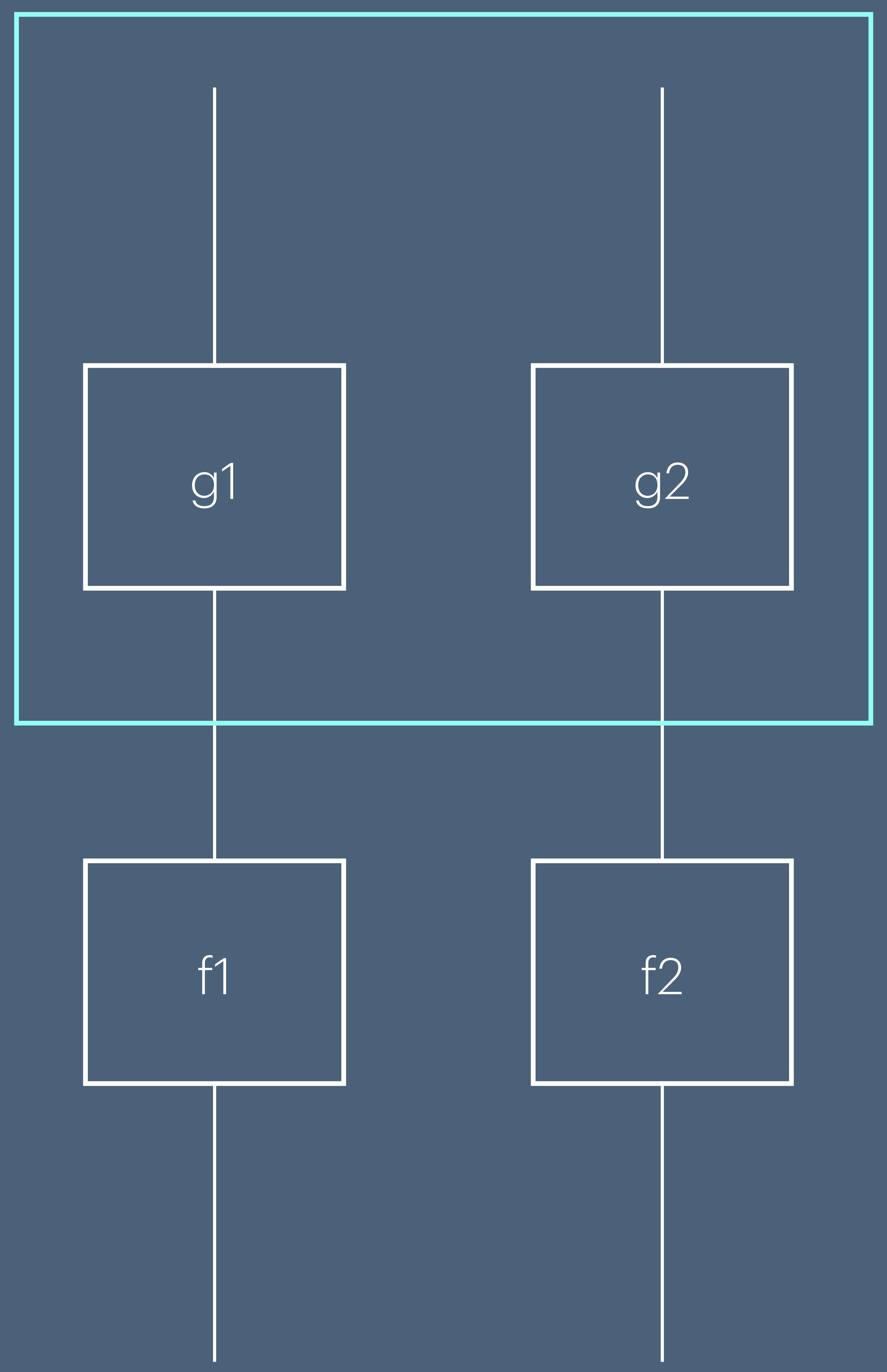


Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$

- $(g1 \otimes g2) \circ (f1 \otimes f2)$

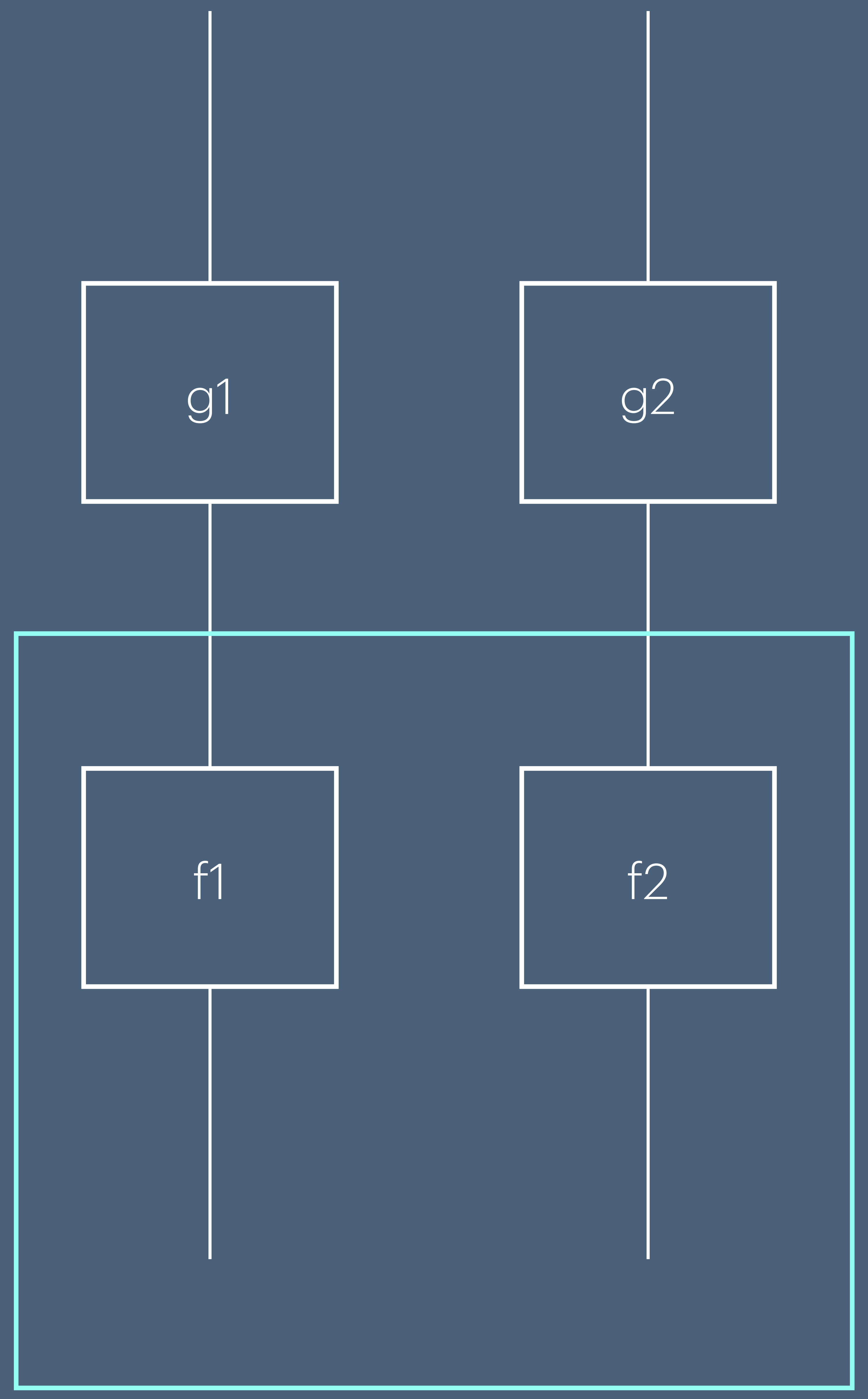


Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$

- $(g1 \otimes g2) \circ (f1 \otimes f2)$

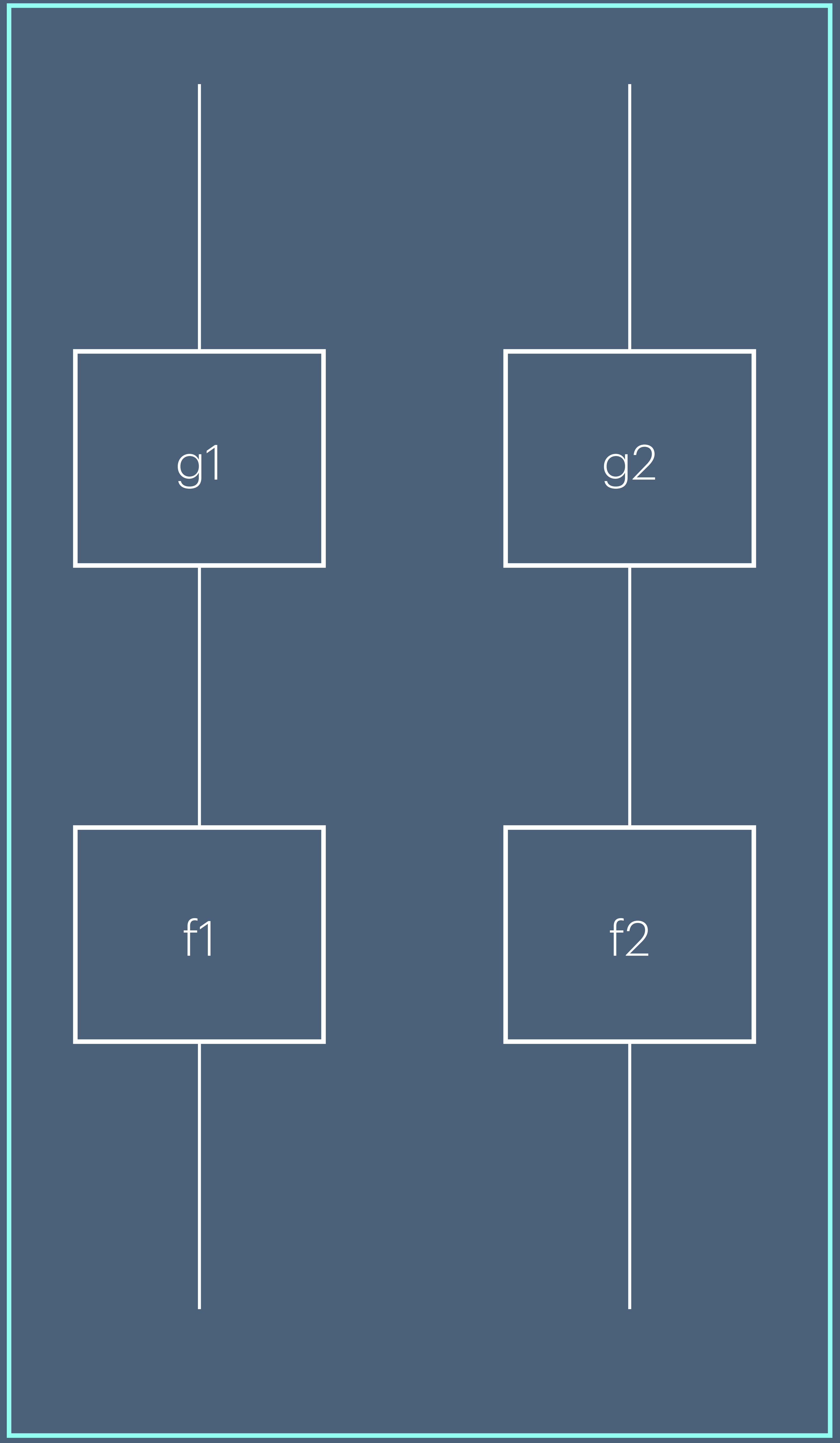


Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$

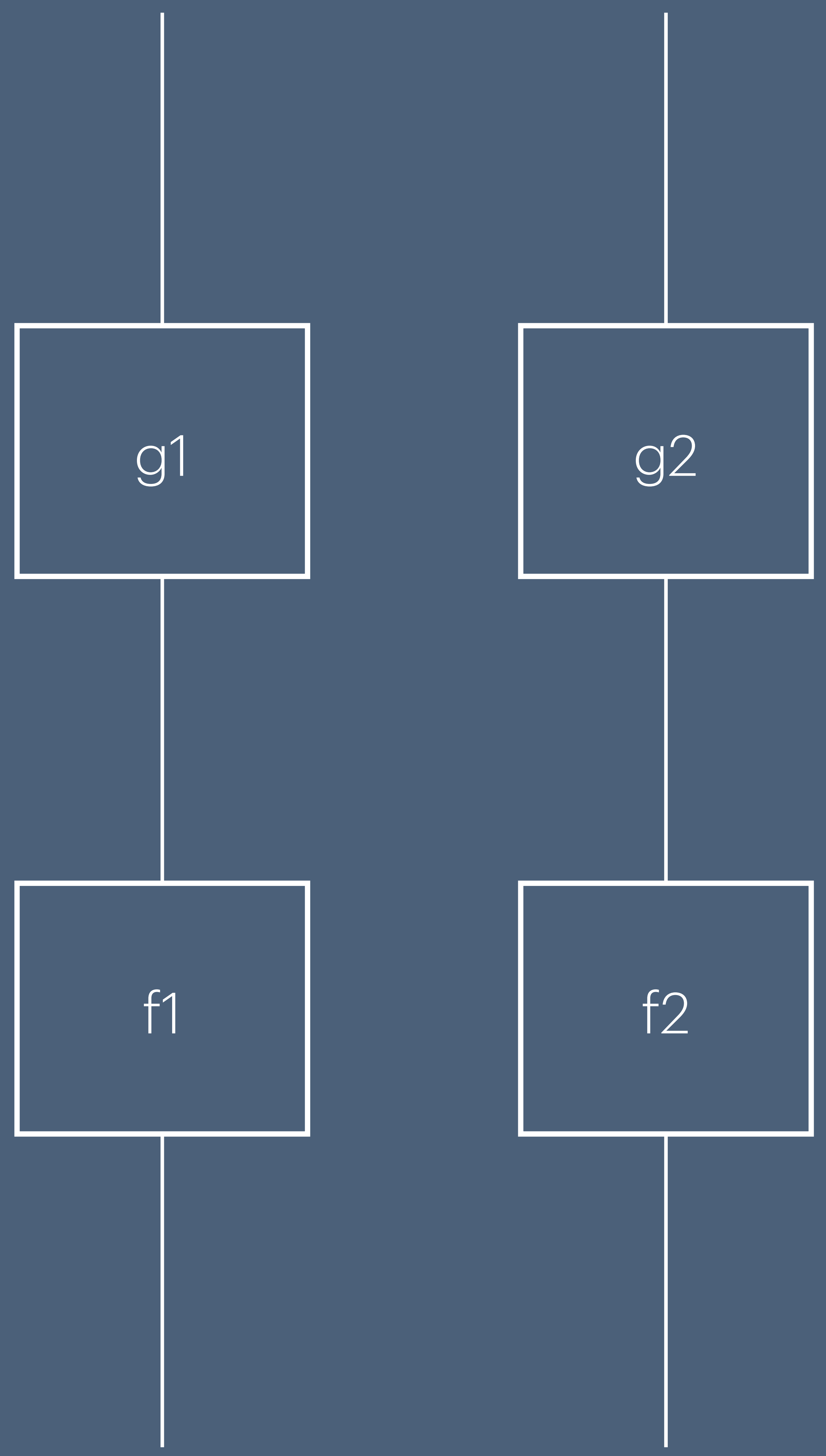
- $(g1 \otimes g2) \circ (f1 \otimes f2)$



Textually distinct,

Diagrammatically equivalent.

- $(g1 \circ f1) \otimes (g2 \circ f2)$
- $(g1 \otimes g2) \circ (f1 \otimes f2)$



Inside a proof assistant

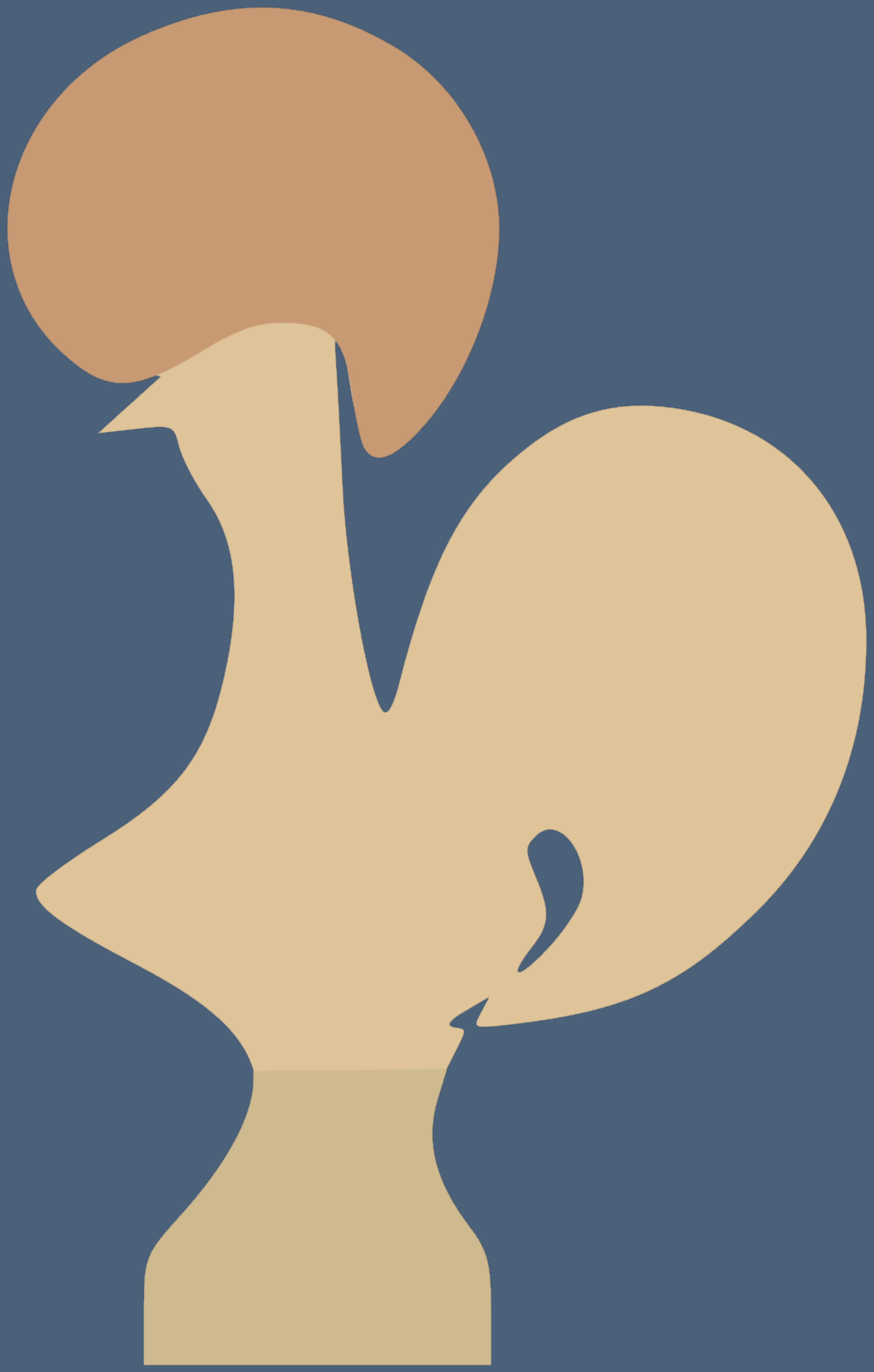
Proof assistants

- Prove properties of a program, using a mathematical specification.
- *Dependently-typed proof assistants* utilize the Curry-Howard-Lambek Isomorphism to construct programs as proofs.
- Reason about structure using induction and recursion.

Proof assistants

Coq

- Interactive, dependently-typed proof assistant.
- Reason about structure *interactively.*

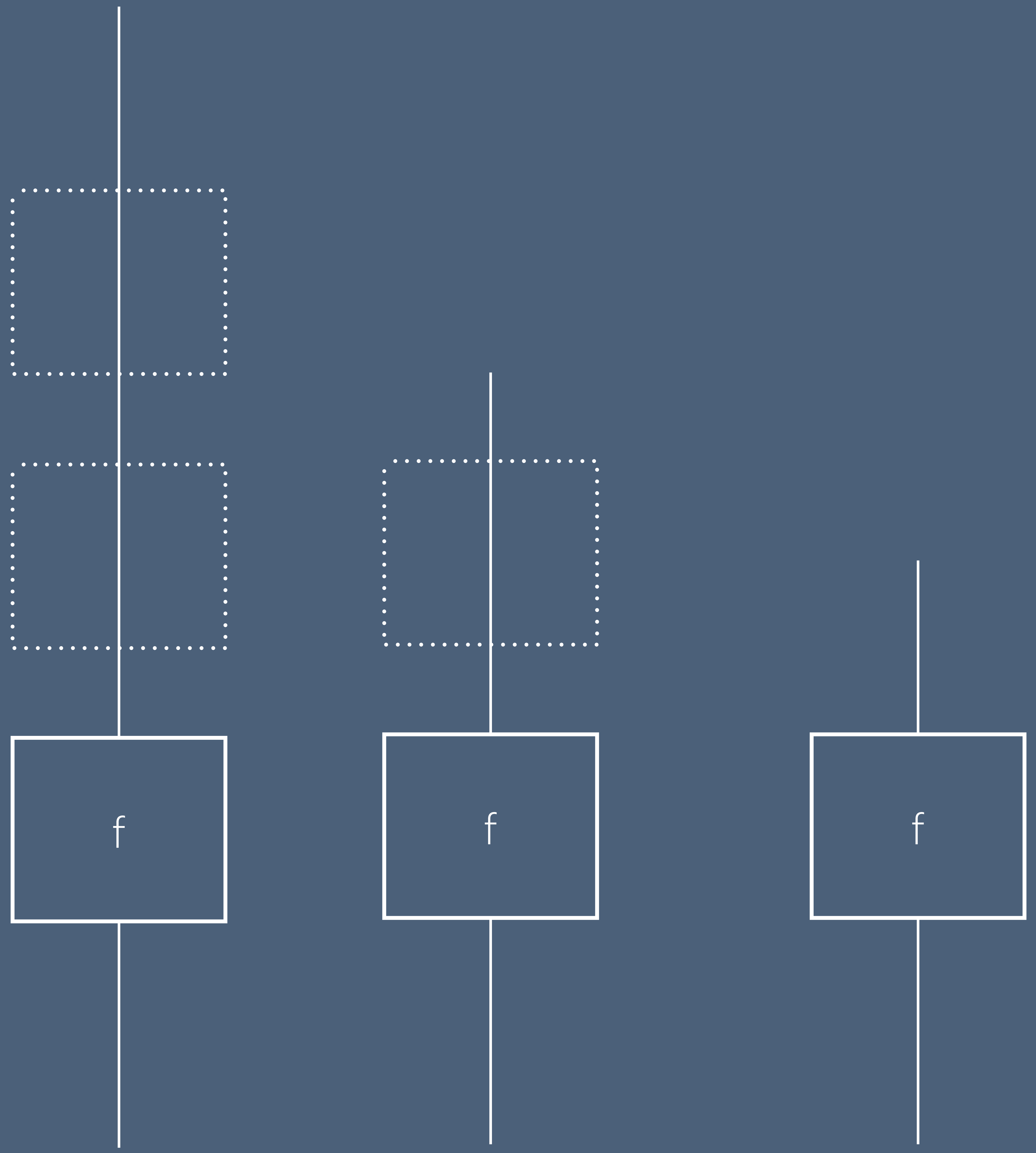


Several intermediate stages

Intermediate stages

- Identity wires do not change the diagram semantically, but they do *structurally*.

- In an interactive proof assistant, *structure matters*.

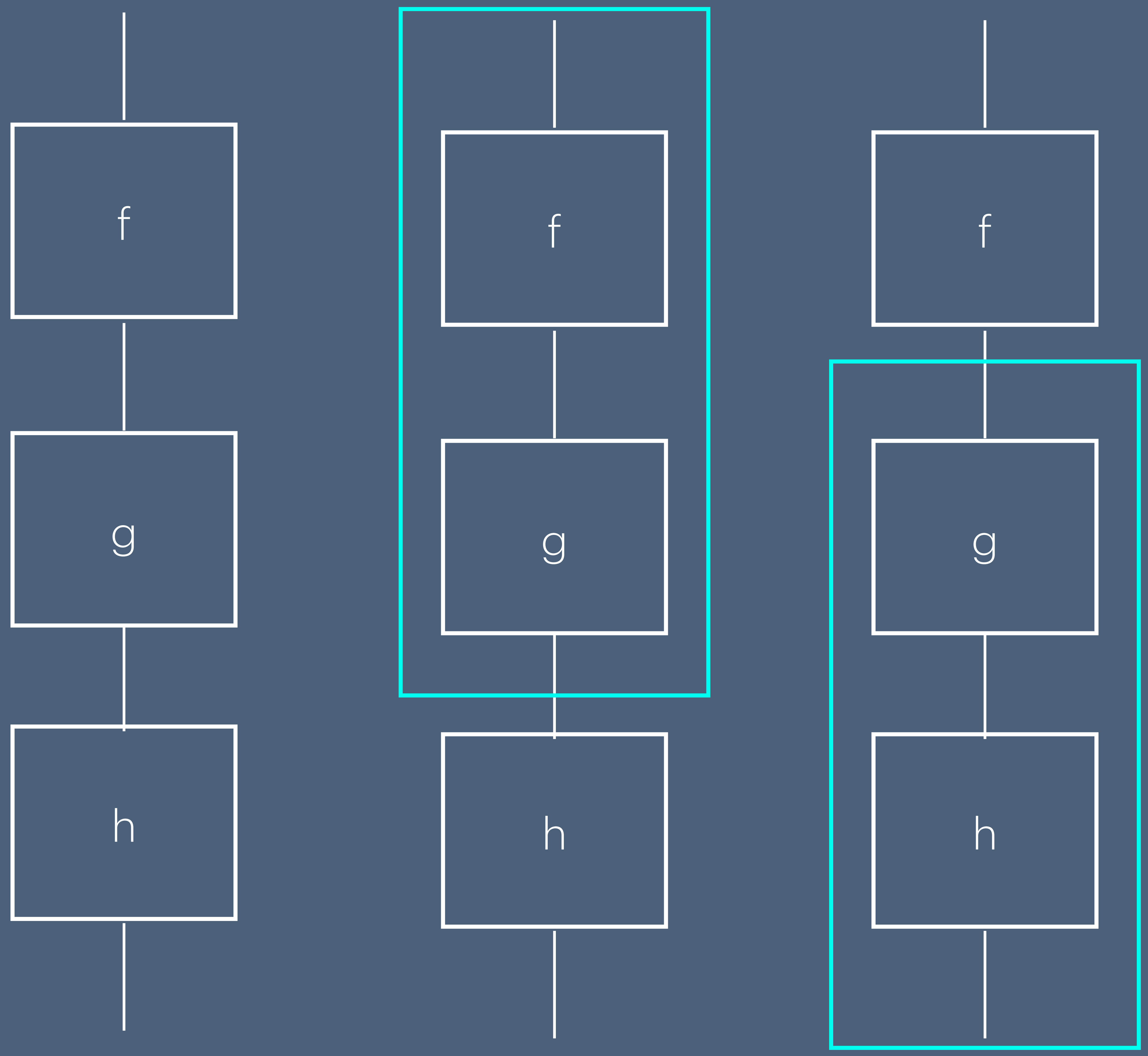


Canonical diagrams employ structural abstractions.

Structural abstractions

The proof assistant cares.

- $f \circ (g \circ h) = (f \circ g) \circ h$
- Proof assistant needs explicit associative information.
- Diagram insufficient!



What do we have so far?

- The simplicity of diagrams is not easy to translate into a purely textual form.
- A diagram may itself be a proof.
- We must reason about several intermediate stages.
- Canonical diagrammatic representations abstract over structural details.

To work with a graphical language in a proof assistant:

- We must do so graphically,
- But using a diagrammatic representation that is more verbose than the canonical one;
- We have several intermediate stages,
- Hence automated diagrammatic generation is desirable.

How do we reason about graphical languages diagrammatically in a proof assistant?

String diagrams associated with a **class of categories**.

Process theory → Category theory

… What is category theory?

- Simplify complex systems via identification of common patterns,
- In our case, *structural* properties.
- To understand how it helps, we do need to know what it *is*.

Category

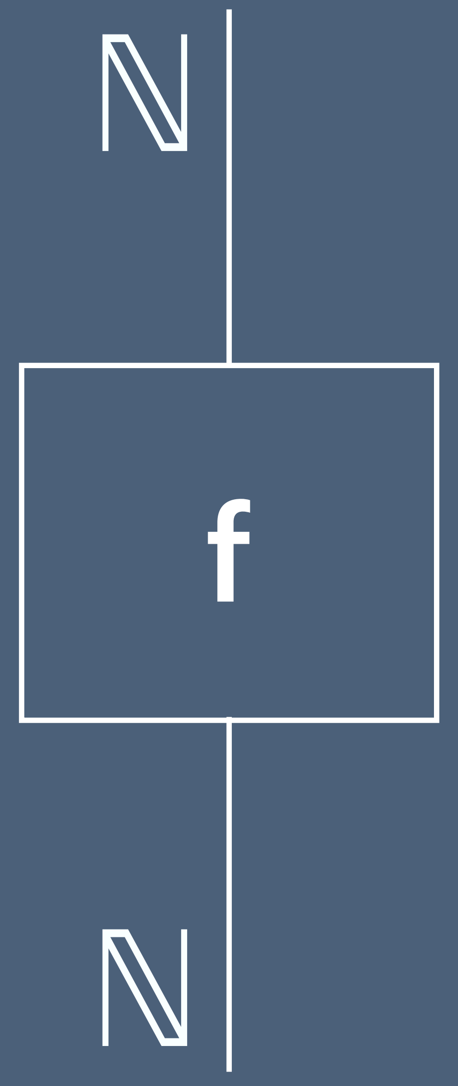
The big bad definition.

- A category **C** comprises:
  - A collection of *objects,* represented A,B,C...
  - A collection of *arrows (or morphisms)* from objects to objects, represented f,g,h....
  - Operations assigning a domain and a codomain for every arrow f, such that if f has domain A and codomain B, we write $f : A \rightarrow B$,
  - A composition operator $\circ$ such that for every pair of arrows $f : A \rightarrow B$, $g : B \rightarrow C$, there exists a composite arrow $g \circ f : A \rightarrow C$, satisfying an associative law $h \circ (g \circ f) = (h \circ g) \circ f$,
  - For every object A, an identity arrow $id_A : A \rightarrow A$, such that $\forall f : A \rightarrow B$, $id_B \circ f = f$ and $f \circ id_A = f$.

Sounds … familiar?

Sounds … familiar?

The process theory we've seen so far forms a category.



Process ↔ Category

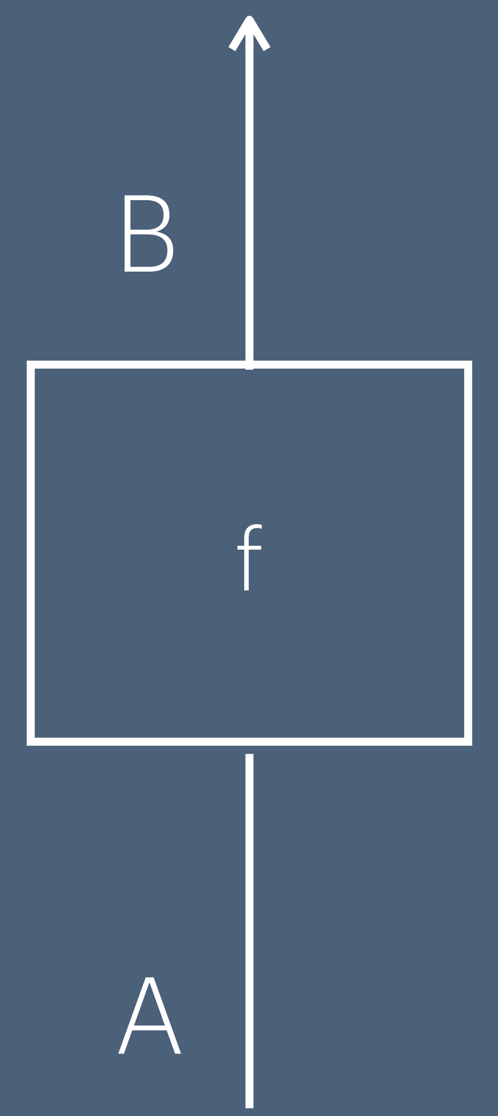
Process	**Category**
Wire	Object
Box	Arrow / Morphism
Identity wire	Identity arrow
Process Composition	Morphism Composition

Process ↔ Category

Process	**Category**
Wire	Object
Box	Arrow / Morphism
Identity wire	Identity arrow
Process Composition	Morphism Composition

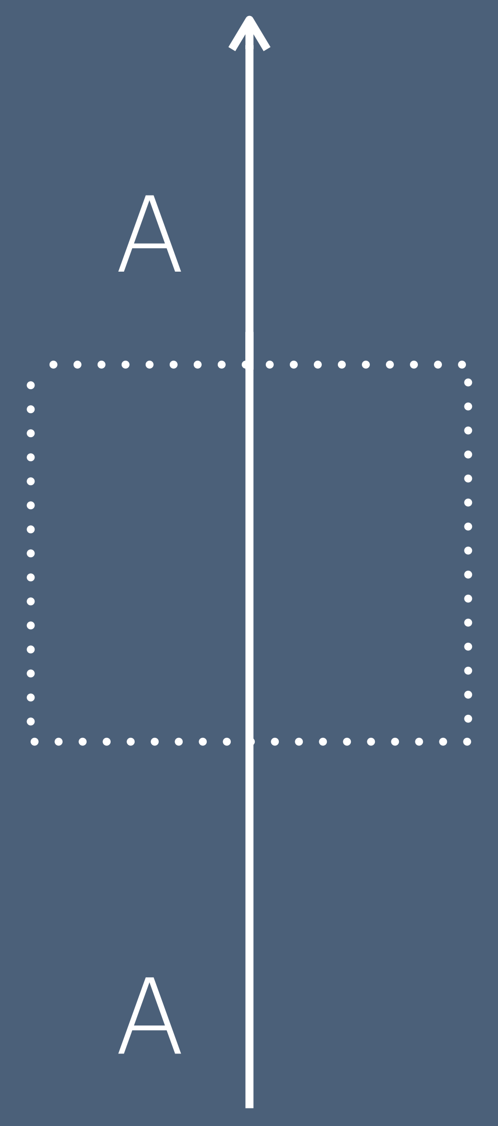
Process ↔ Category

Process	Category
Wire	Object
Box	Arrow / Morphism
Identity wire	Identity arrow
Process Composition	Morphism Composition



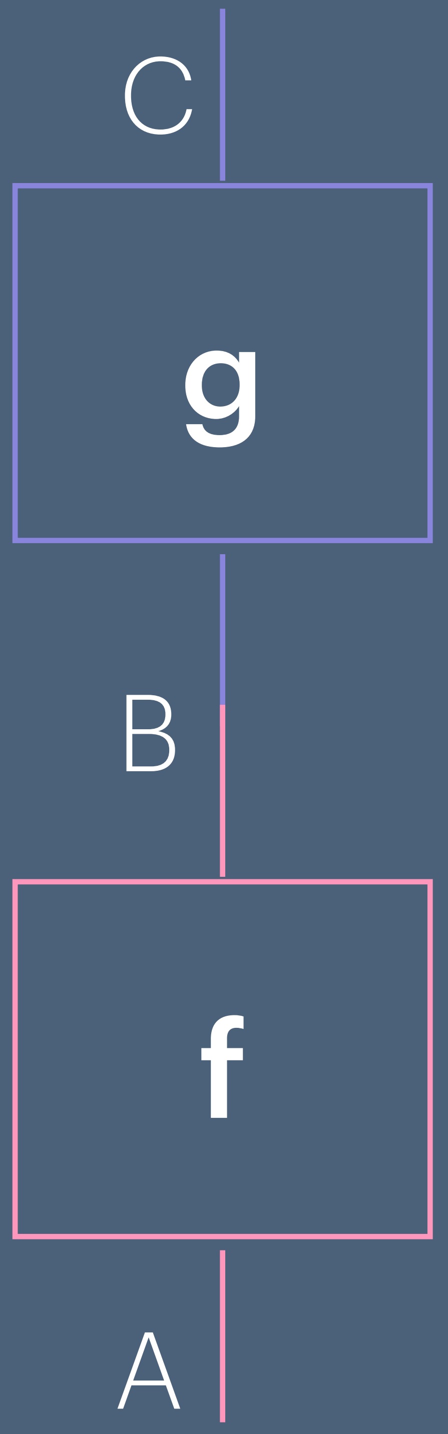
Process ↔ Category

Process	Category
Wire	Object
Box	Arrow / Morphism
Identity wire	Identity arrow
Process Composition	Morphism Composition



Process ⟷ Category

Process	Category
Wire	Object
Box	Arrow / Morphism
Identity wire	Identity arrow
Process Composition	Morphism Composition



Add in $\otimes$, and we have … a monoidal category.

Monoidal Category

- A category C is *monoidal* if it consists of:
  - A *bifunctor* $\otimes : C \times C \to C$, meaning:
    $id_A \otimes id_B = id_{A \otimes B}$
    $(f' \otimes g') \circ (f \otimes g) = (f' \circ f) \otimes (g' \circ g)$,
  - An object $e \in C$ called the *unit* object ,
  - Natural isomorphisms:
    $\alpha = \alpha_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$
    $\lambda = \lambda_A : I \otimes A \simeq A$
    $\rho = \rho_A : A \otimes I \simeq A$

Such that $\forall$ A, B, C, D, E, the diagrams below commute:

$$\begin{tikzcd}
& (A \otimes (B \otimes C)) \otimes D \arrow[rr, "\alpha_{A,B\otimes C,D}"] & & A \otimes ((B \otimes C) \otimes D) \arrow[dr, "id_A \otimes \alpha_{B,C,D}"] & \\
((A \otimes B) \otimes C) \otimes D \arrow[ur, "\alpha_{A,B,C} \otimes id_D"] \arrow[drr, "\alpha_{A\otimes B,C,D}"'] & & & & A \otimes (B \otimes (C \otimes D)) \\
& & (A \otimes B) \otimes (C \otimes D) \arrow[urr, "\alpha_{A,B,C\otimes D}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes I) \otimes B \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "\rho_A \otimes id_B"'] & & A \otimes (I \otimes B) \arrow[dl, "id_A \otimes \lambda_B"] \\
& A \otimes B &
\end{tikzcd}$$

Monoidal Category

- A category C is *monoidal* if it consists of:
  - A *bifunctor* ⊗ : C × C → C, meaning:
    $id_A \otimes id_B = id_{A \otimes B}$
    $(f' \otimes g') \circ (f \otimes g) = (f' \circ f) \otimes (g' \circ g)$,
  - An object e ∈ C called the *unit* object ,
  - Natural isomorphisms:
    $\alpha = \alpha_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$
    $\lambda = \lambda_A : I \otimes A \simeq A$
    $\rho = \rho_A : A \otimes I \simeq A$

Such that ∀ A, B, C, D, E, the diagrams below commute:

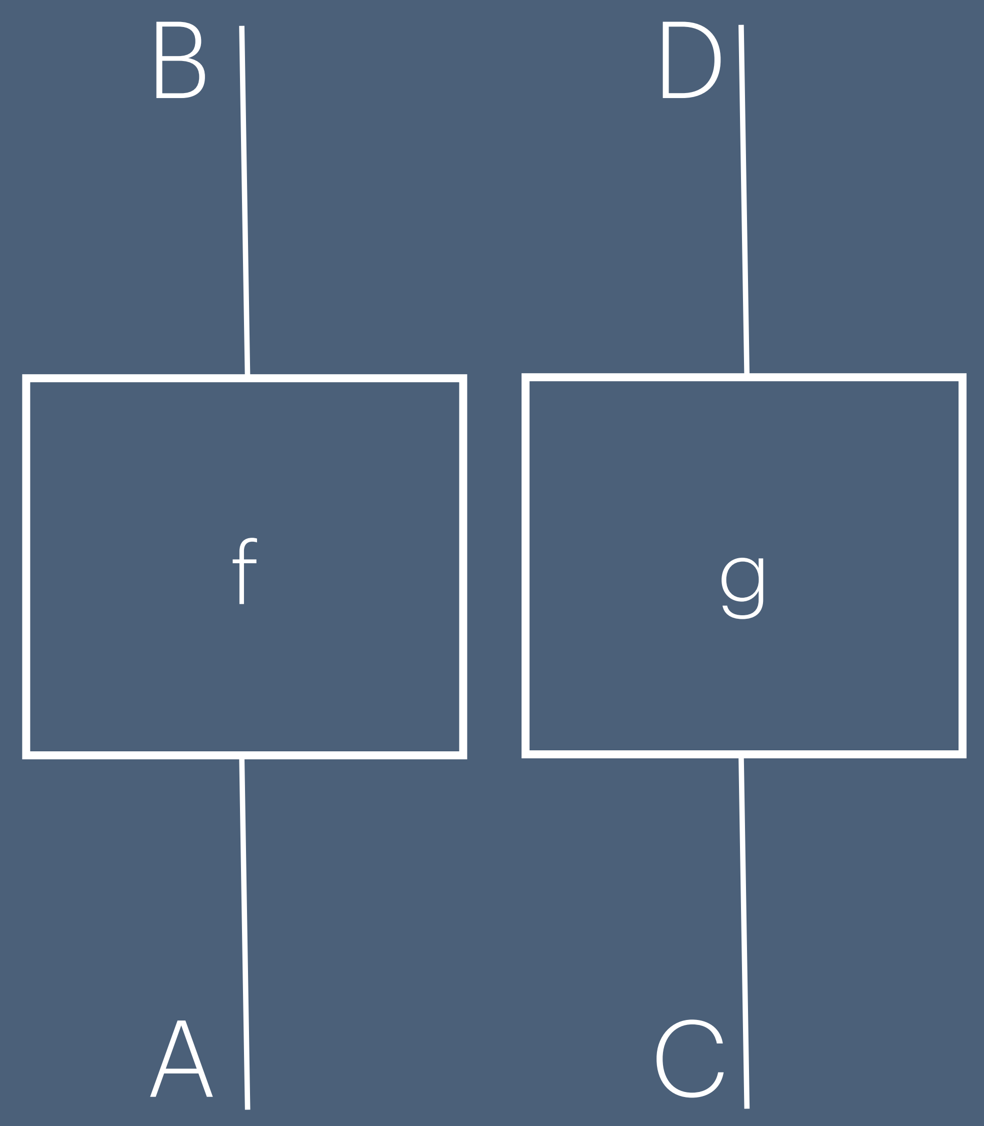
$$\begin{tikzcd}
 & (A \otimes (B \otimes C)) \otimes D \arrow[rr, "\alpha_{A,B\otimes C,D}"] & & A \otimes ((B \otimes C) \otimes D) \arrow[dr, "id_A \otimes \alpha_{B,C,D}"] & \\
((A \otimes B) \otimes C) \otimes D \arrow[ur, "\alpha_{A,B,C} \otimes id_D"] \arrow[drr, "\alpha_{A\otimes B,C,D}"'] & & & & A \otimes (B \otimes (C \otimes D)) \\
 & & (A \otimes B) \otimes (C \otimes D) \arrow[urr, "\alpha_{A,B,C\otimes D}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes I) \otimes B \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "\rho_A \otimes id_B"'] & & A \otimes (I \otimes B) \arrow[dl, "id_A \otimes \lambda_B"] \\
 & A \otimes B &
\end{tikzcd}$$

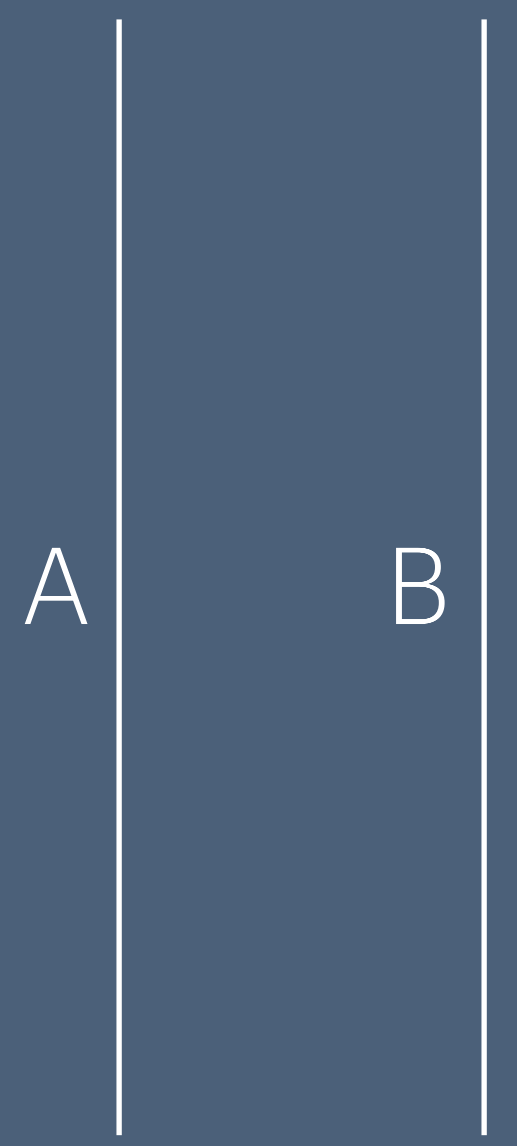
Diagrams >>> definitions

We add ⊗

Operating on both objects and categories.



$f \otimes g$



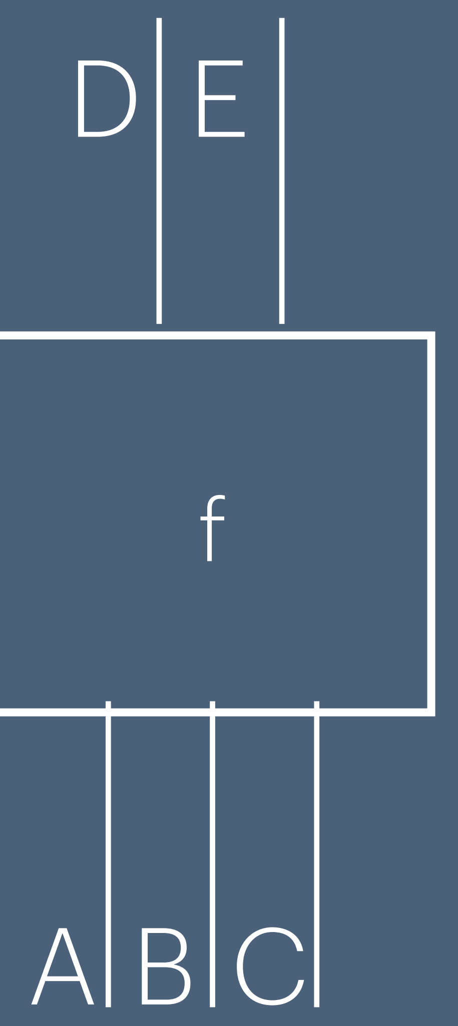
$A \otimes B$

We also add a unit object

Whose diagrammatic representation is just empty.

How does this impact *structure?*
We'll see :)

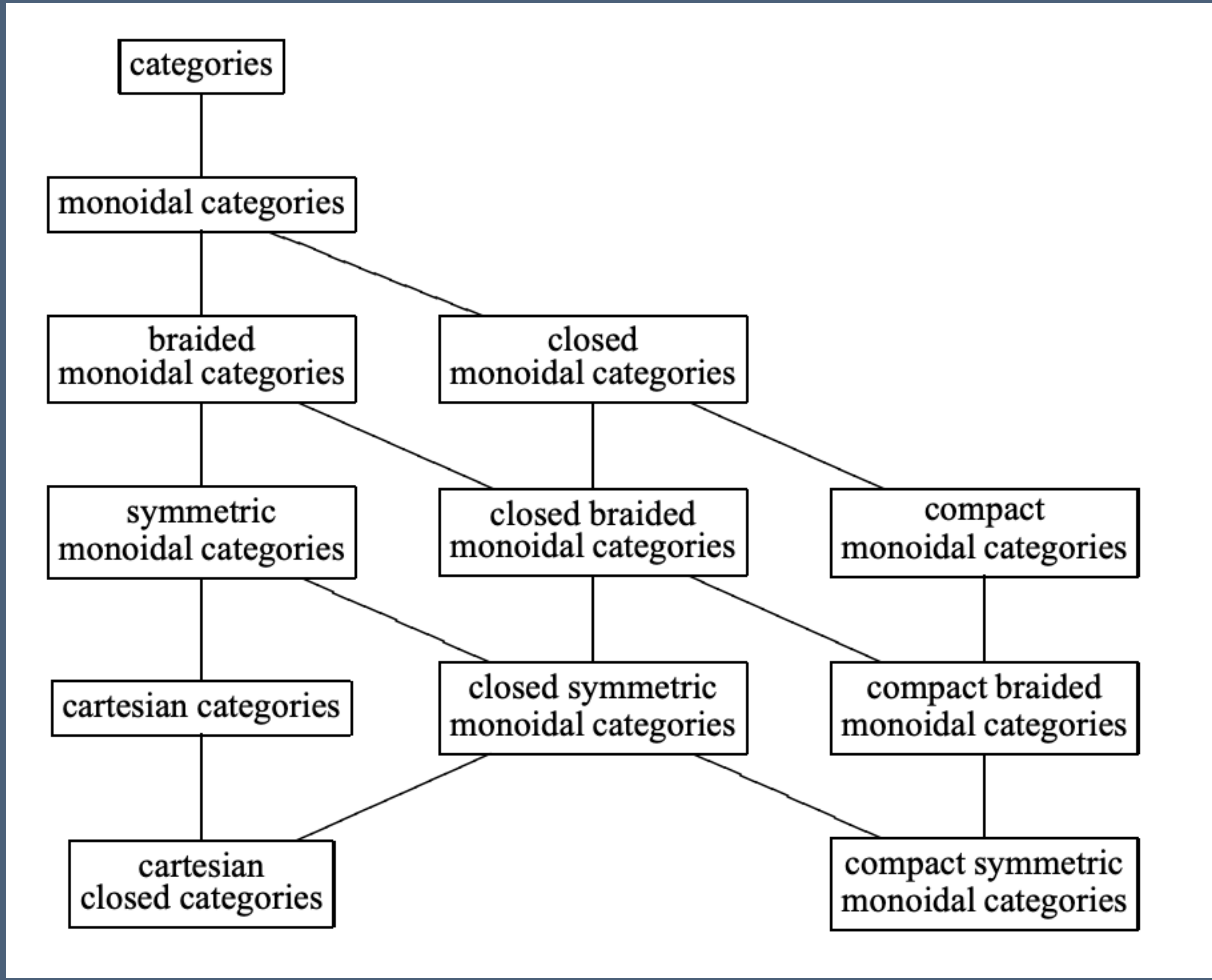
We now have processes with multiple inputs and outputs, with categorical semantics.



Categorical hierarchy...

We can keep going...

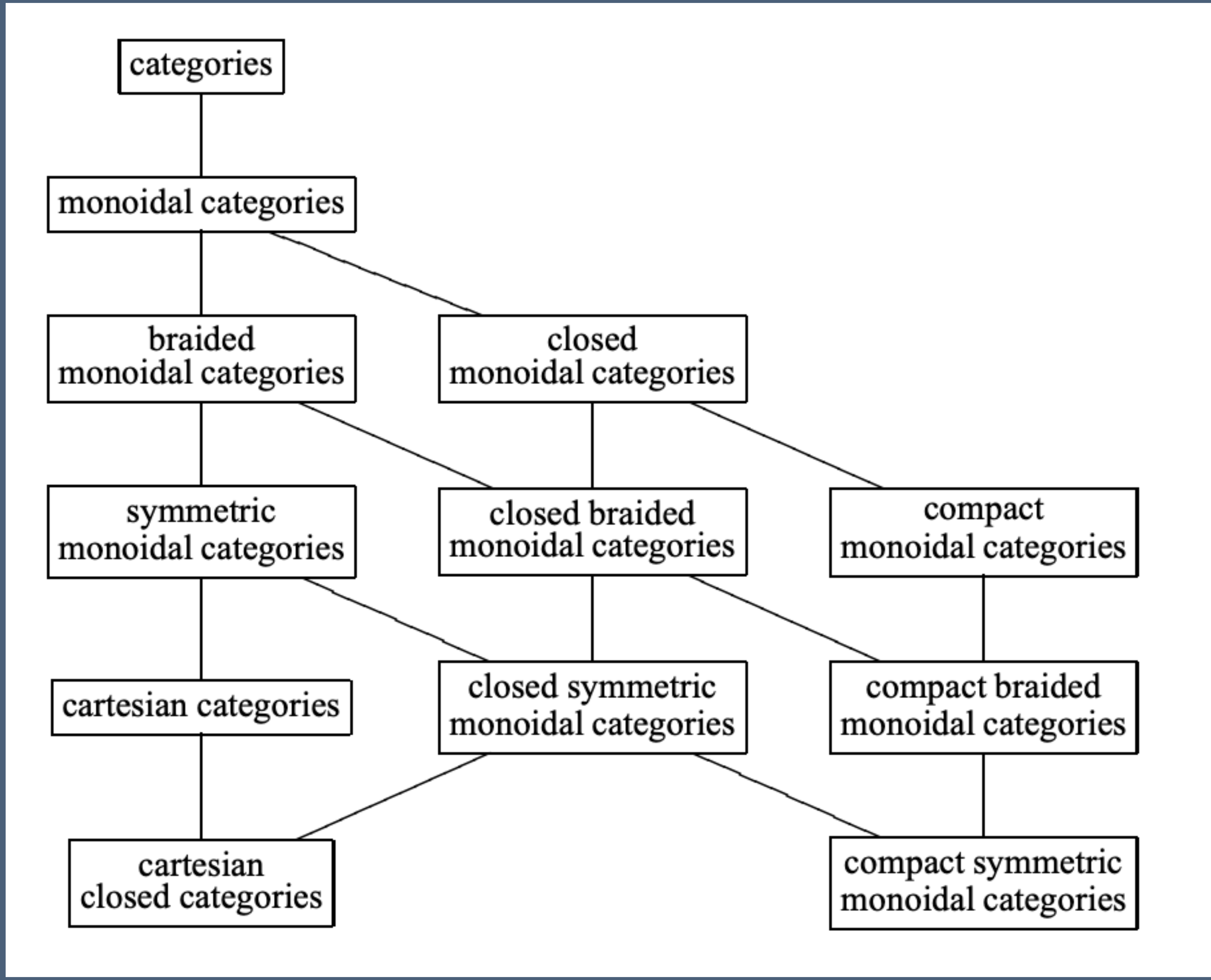
- As we go to more complex classes, we add more structure.
- We could be here forever if we went through all of these ...



Categorical hierarchy…

We can keep going…

- As we go to more complex classes, we add more structure.
- We could be here forever if we went through all of these …
- So let's not do that.

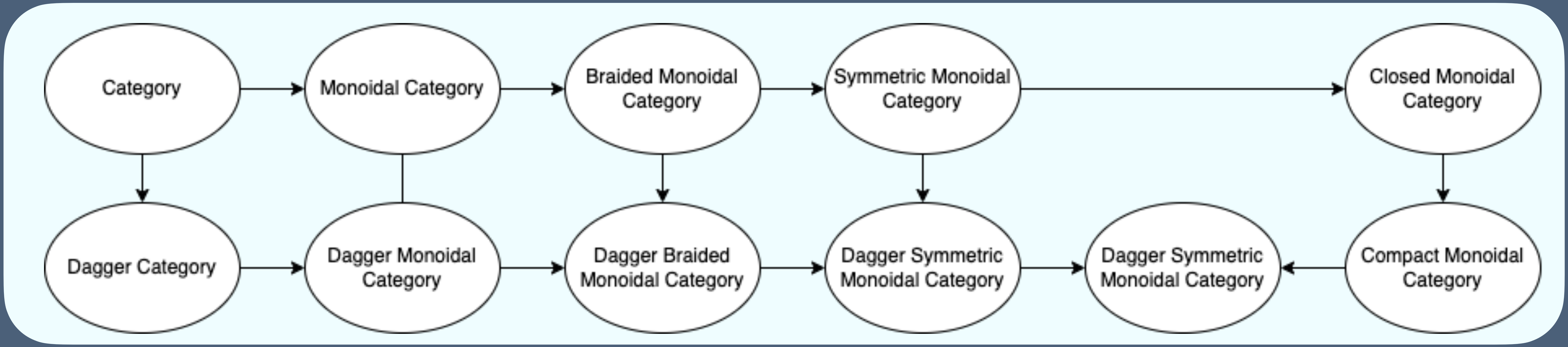


ViCAR

ViCAR

Visualizing Categories with Automated Rewriting

- Framework for reasoning about (monoidal) categories in Coq.
- Specifically, these classes.



Why do we care?

Visualizing Categories with Automated Rewriting

- Several commonly encountered constructs can be instantiated with categorical semantics.
- For example, matrices, relations, simply-typed lambda calculus …
- Verification methodologies may coincide due to structural similarities.
- We want to take advantage of shared structure so categorical properties can be utilized in proof.

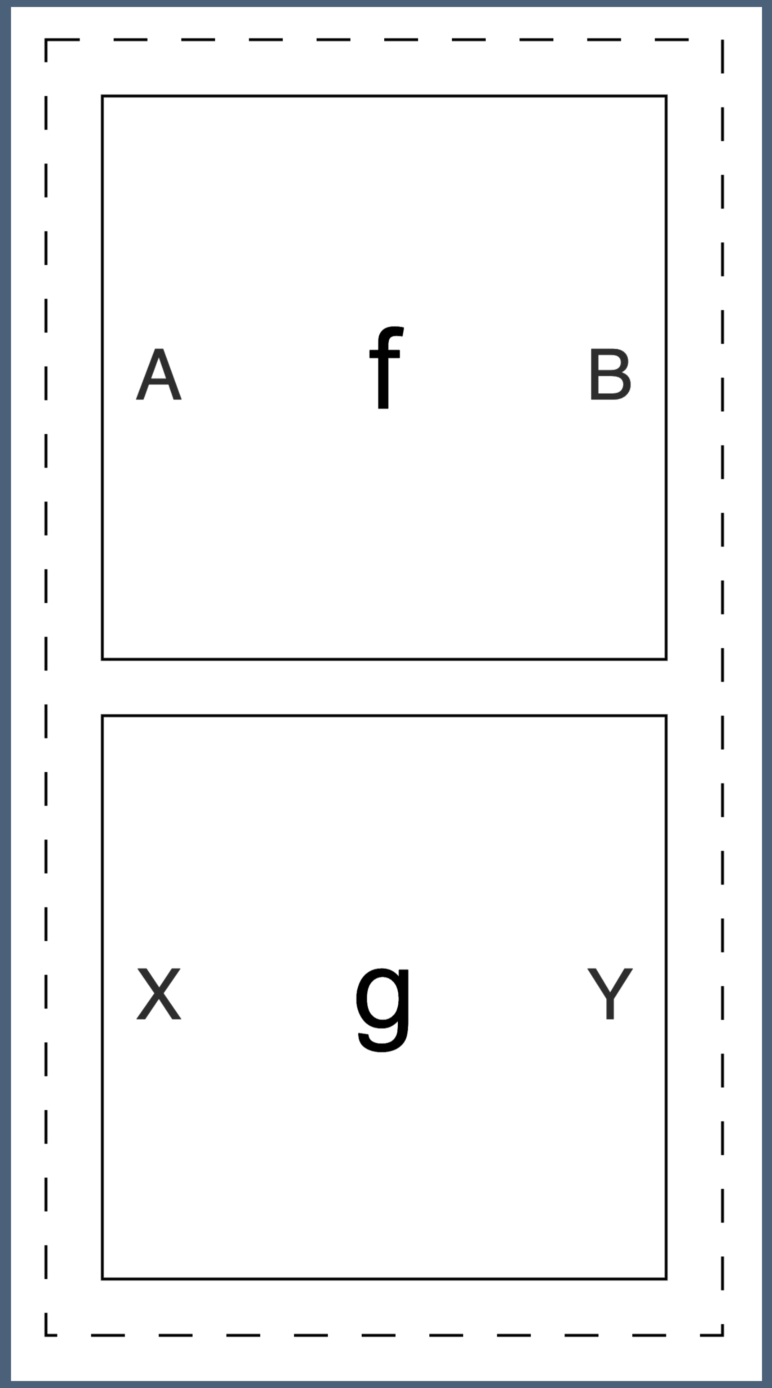
ViCAR

Visualizing Categories with Automated Rewriting

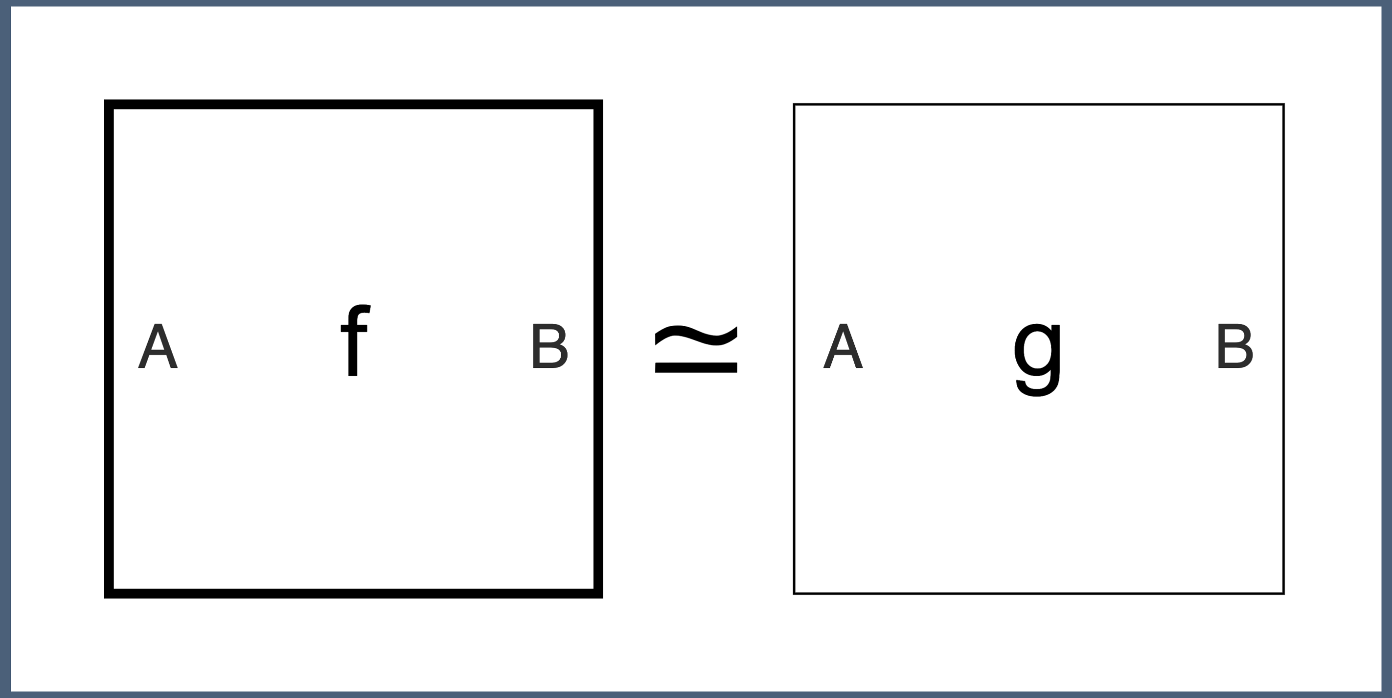
- Constructively defined categorical typeclasses, making use of Coq's inference.
- Certain uninteresting patterns emerge when dealing with proofs in Coq.
- We want these to be handled using automation.
- ViCAR provides automation tactics for several commonly encountered situations.

ViCAR

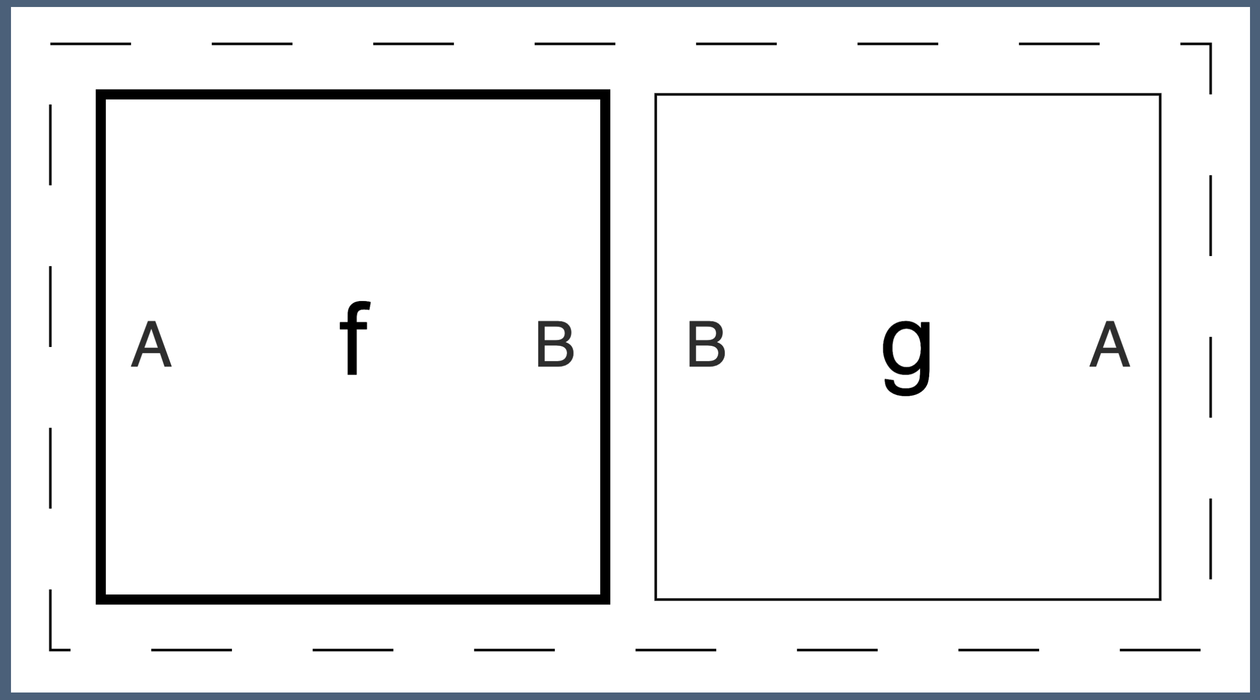
Visualizing Categories with Automated Rewriting



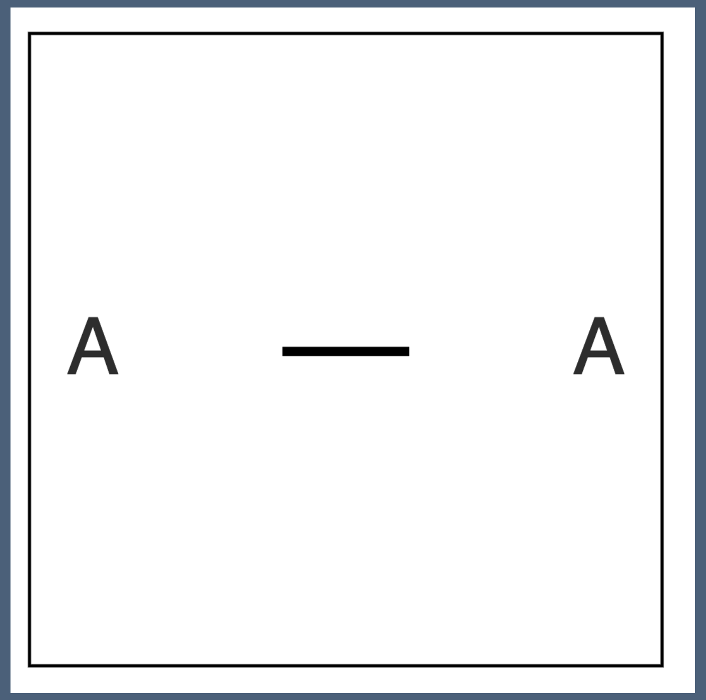
$f \otimes g$



$f \simeq g$



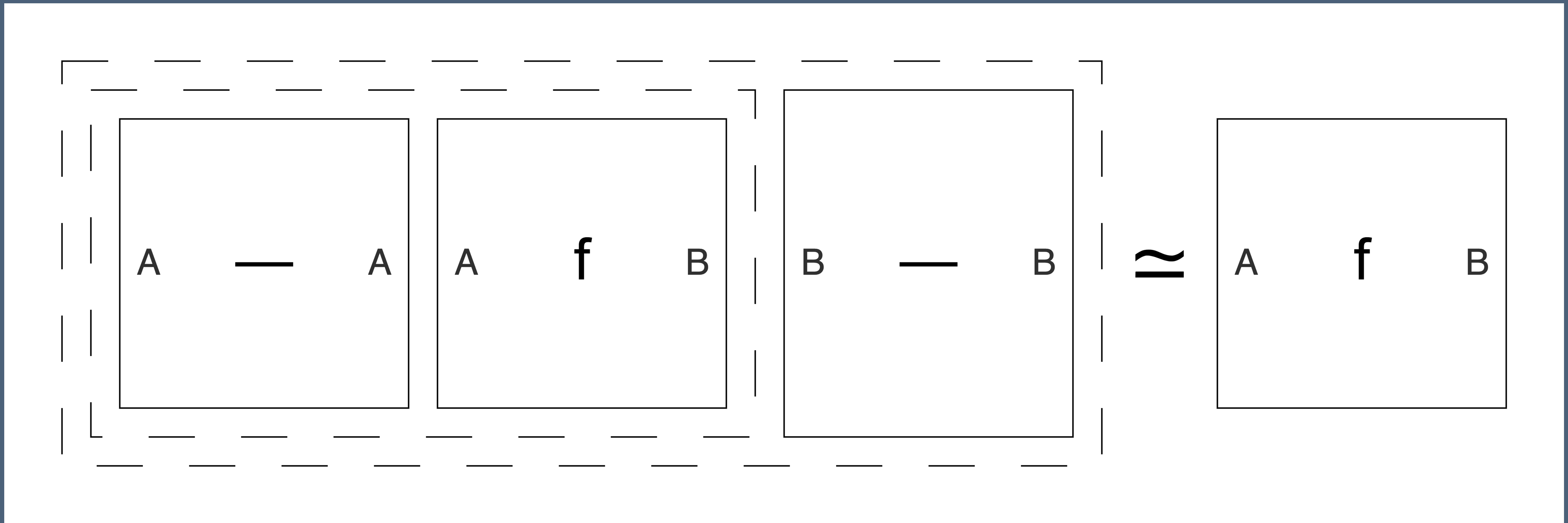
$f \circ g$



id_A

ViCAR

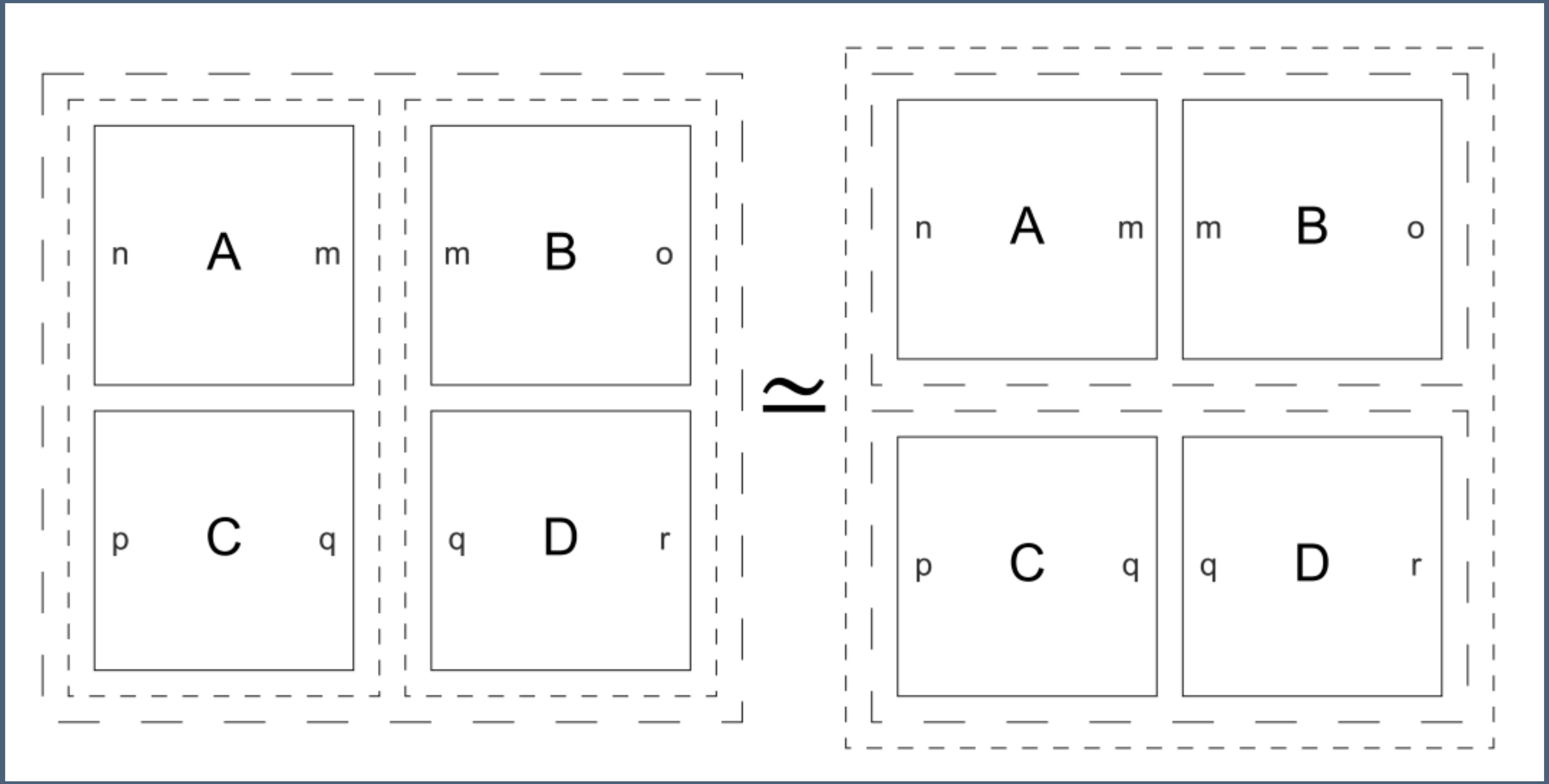
Visualizing Categories with Automated Rewriting



$(id_A \circ f) \circ id_B \simeq f$

ViCAR

Visualizing Categories with Automated Rewriting



$$(A \otimes C) \times (B \otimes D) \simeq (A \times B) \otimes (C \times D)$$

Categories get more complex.

Braided Monoidal Category

- A *braiding* on a monoidal category consists of a natural family of isomorphisms,

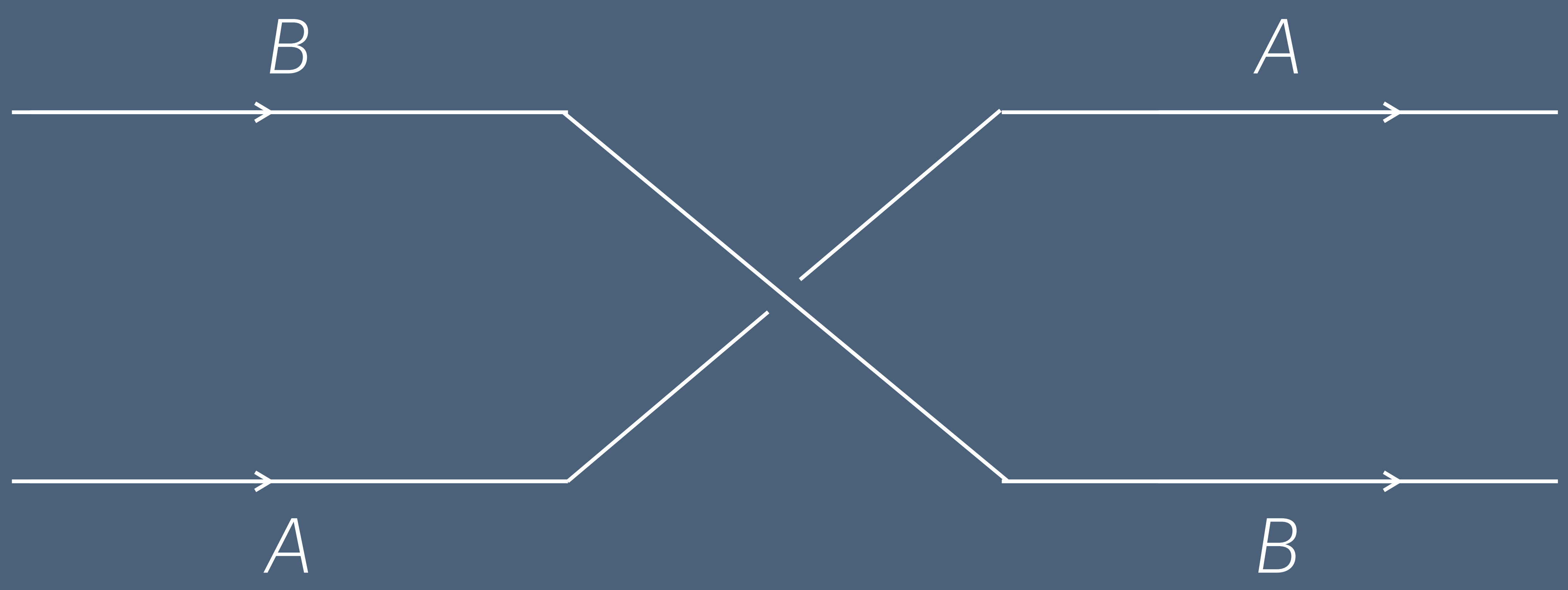
  $c_{A,B} : A \otimes B \simeq B \otimes A$

  such that the diagrams on the left commute.

$$
\begin{tikzcd}
 & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"] & B \otimes (A \otimes C) \arrow[dr, "id_B \otimes c_{A,C}"] & \\
(A \otimes B) \otimes C \arrow[ur, "c_{A,B} \otimes id_C"] \arrow[dr, "\alpha_{A,B,C}"'] & & & B \otimes (C \otimes A) \\
 & A \otimes (B \otimes C) \arrow[r, "c_{A,B \otimes C}"'] & (B \otimes C) \otimes A \arrow[ur, "\alpha_{B,C,A}"'] &
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"] & B \otimes (A \otimes C) \arrow[dr, "id_B \otimes c^{-1}_{C,A}"] & \\
(A \otimes B) \otimes C \arrow[ur, "c^{-1}_{B,A} \otimes id_C"] \arrow[dr, "\alpha_{A,B,C}"'] & & & B \otimes (C \otimes A) \\
 & A \otimes (B \otimes C) \arrow[r, "c^{-1}_{B \otimes C,A}"'] & (B \otimes C) \otimes A \arrow[ur, "\alpha_{B,C,A}"'] &
\end{tikzcd}
$$

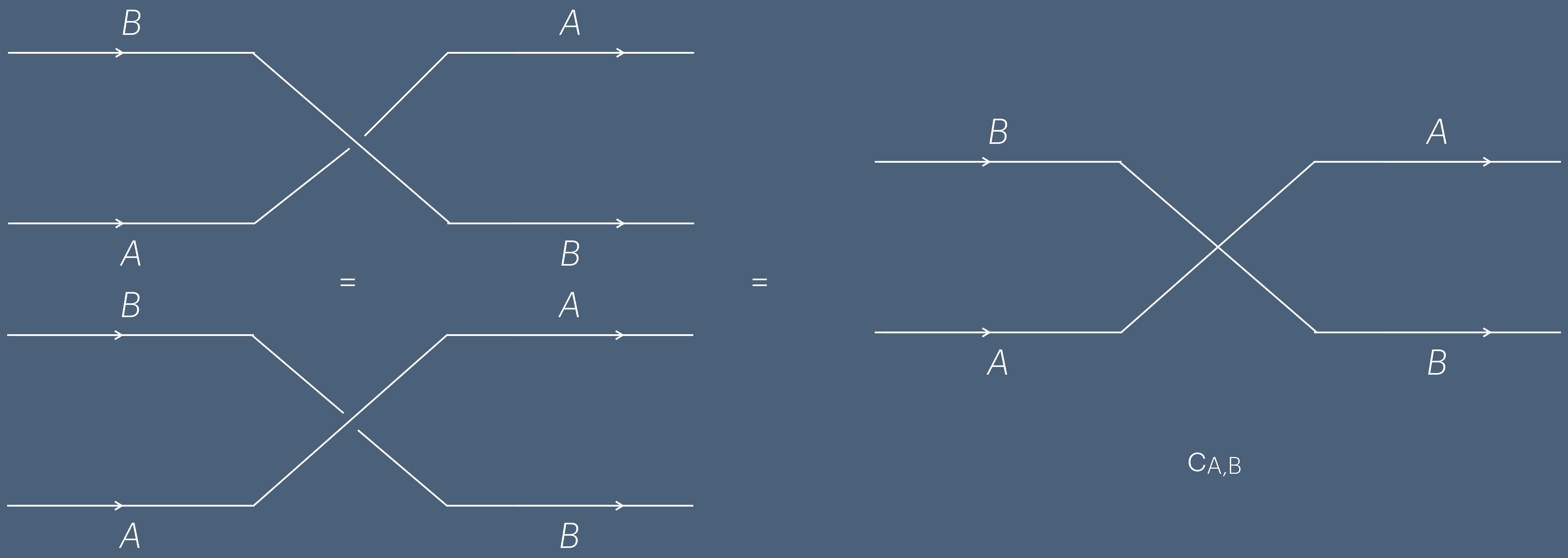
Braided Monoidal Category



Braiding $c_{A,B}$

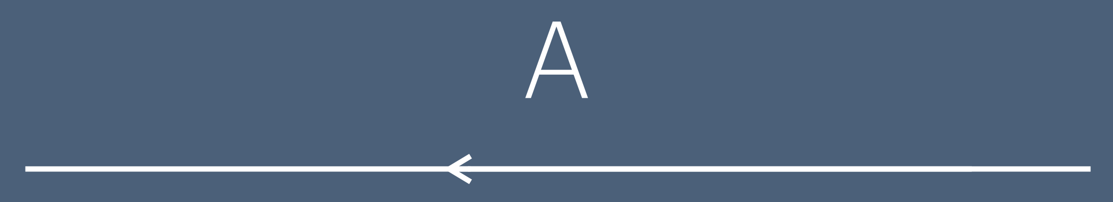
Symmetric Monoidal Category

- A symmetric monoidal category is a braided monoidal category with a self-inverse braiding, i.e. $c_{A,B} = c^{-1}_{B,A}$,

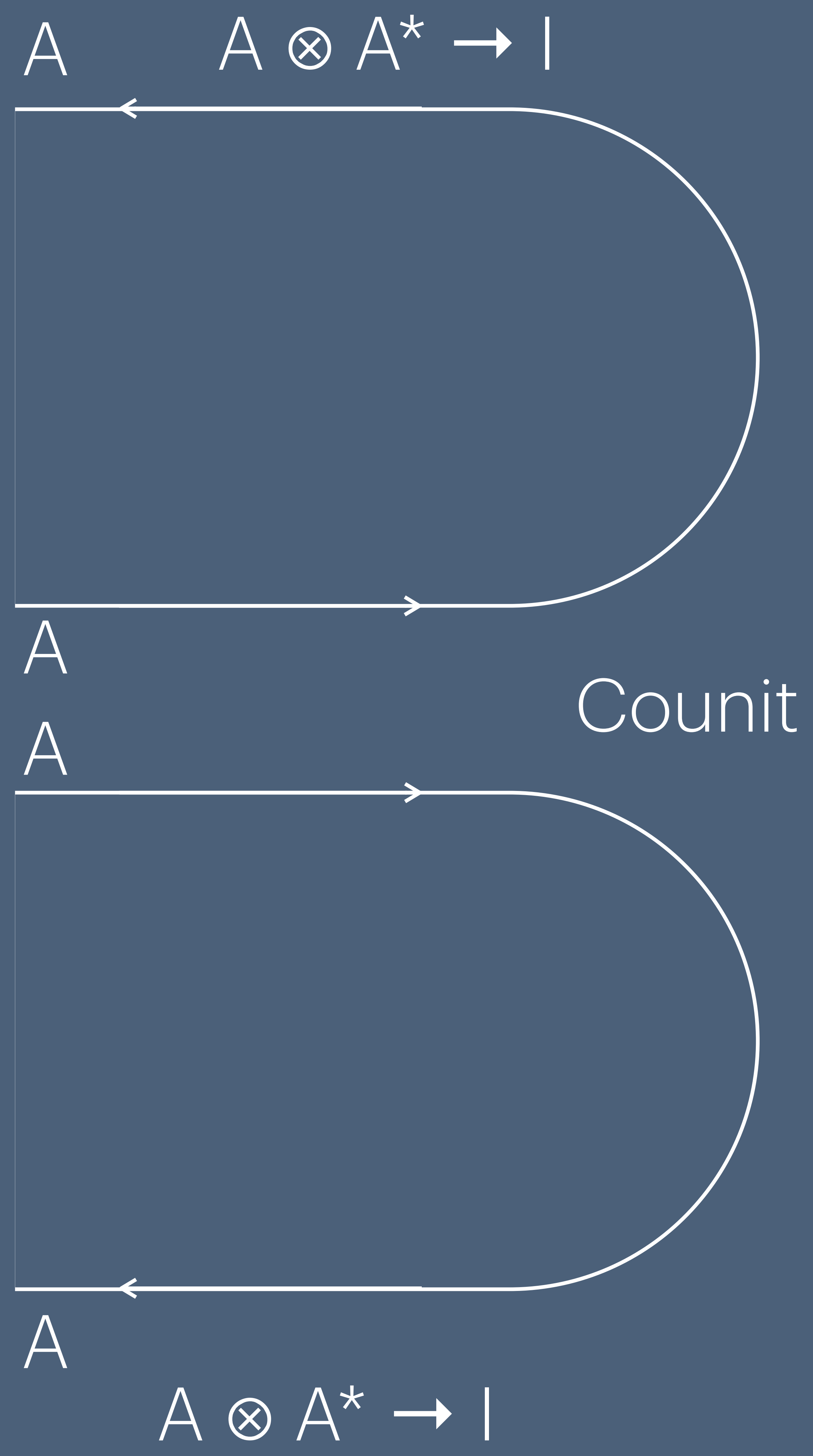


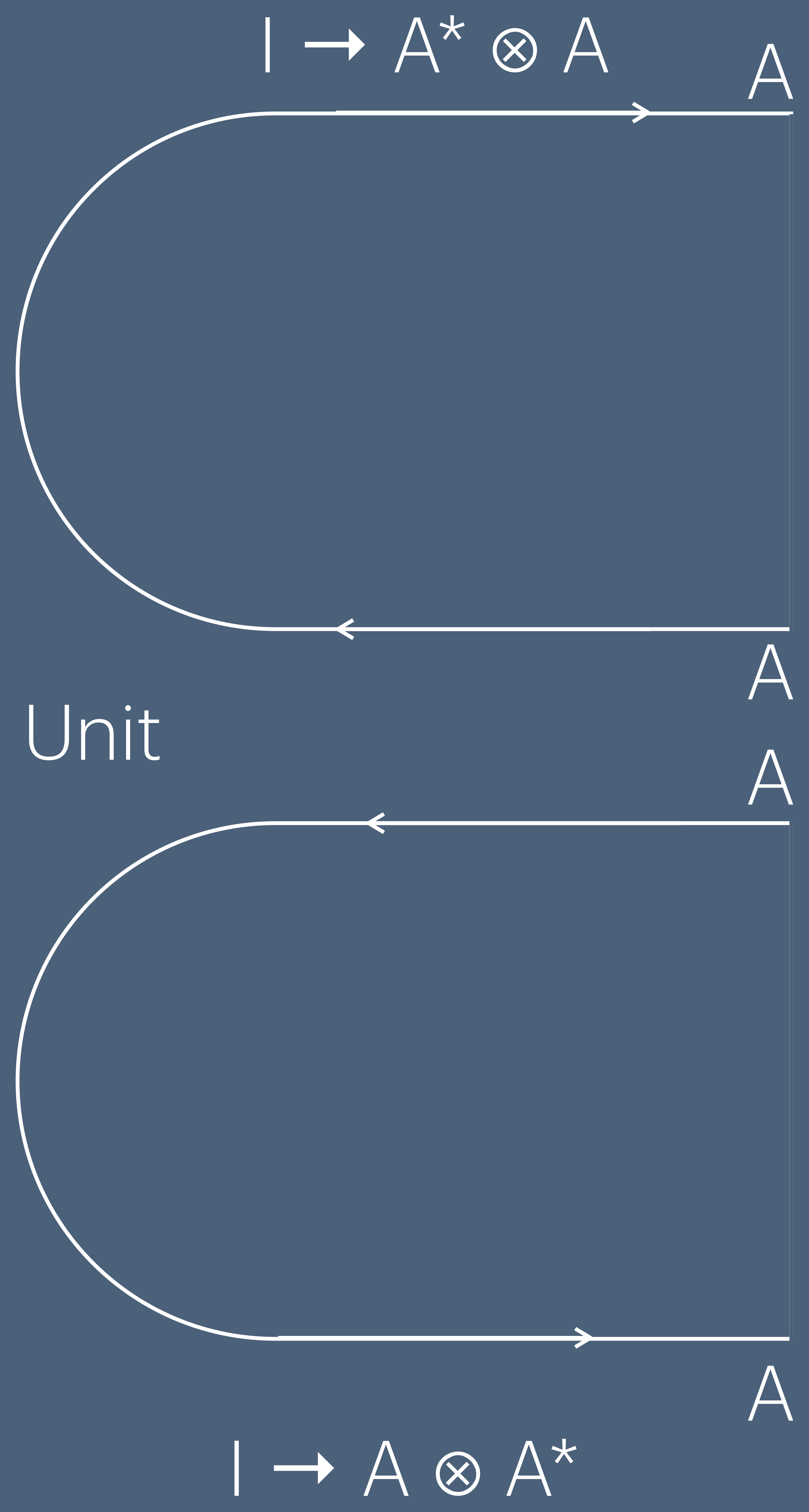
$c_{A,B}$

Autonomous Category



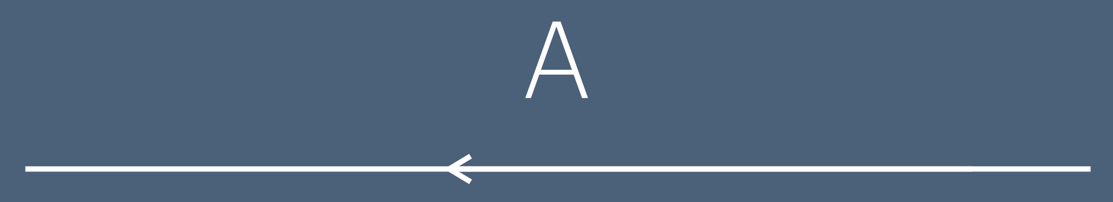
Every object has a *dual*, A*



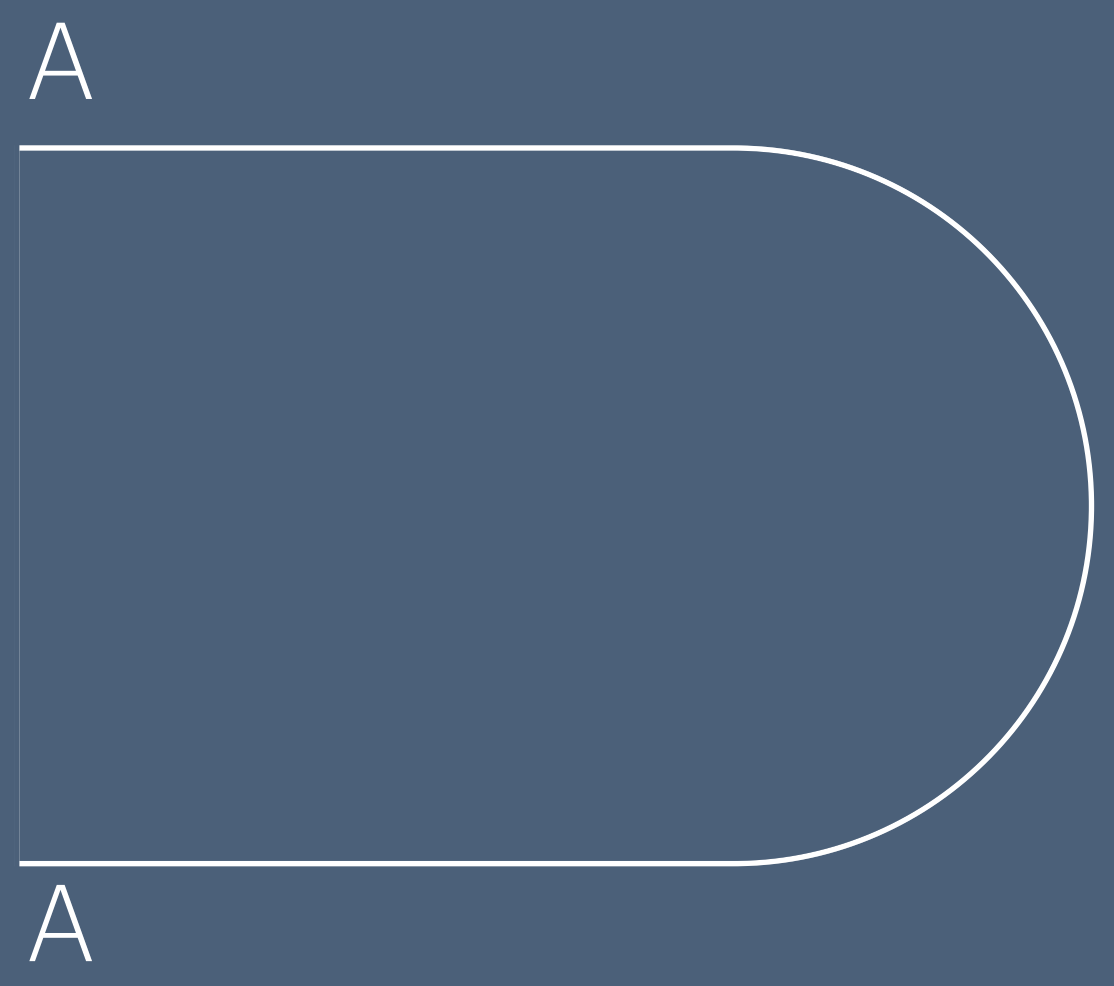


Compact Closed Category

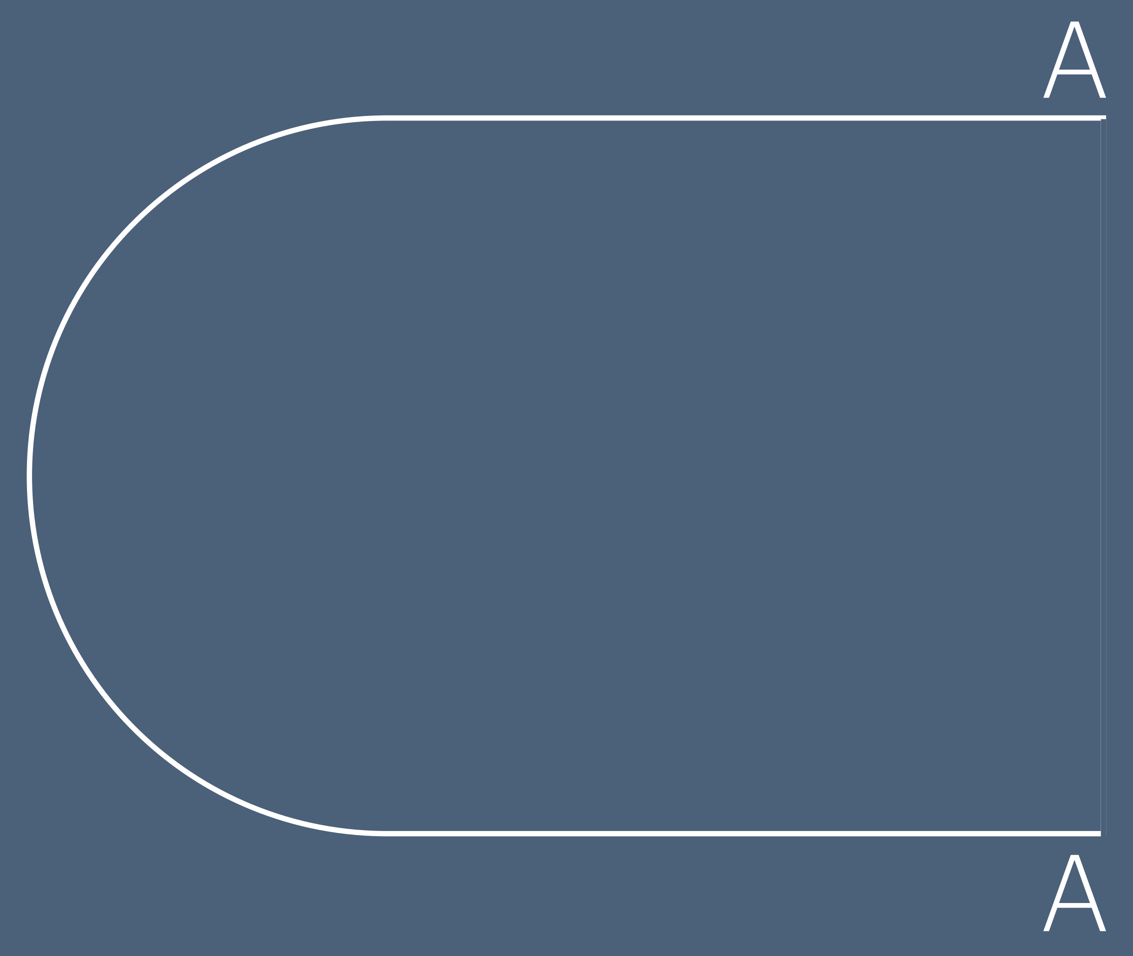
Symmetric Monoidal + Autonomous



Every object has a *dual*, A*



Counit

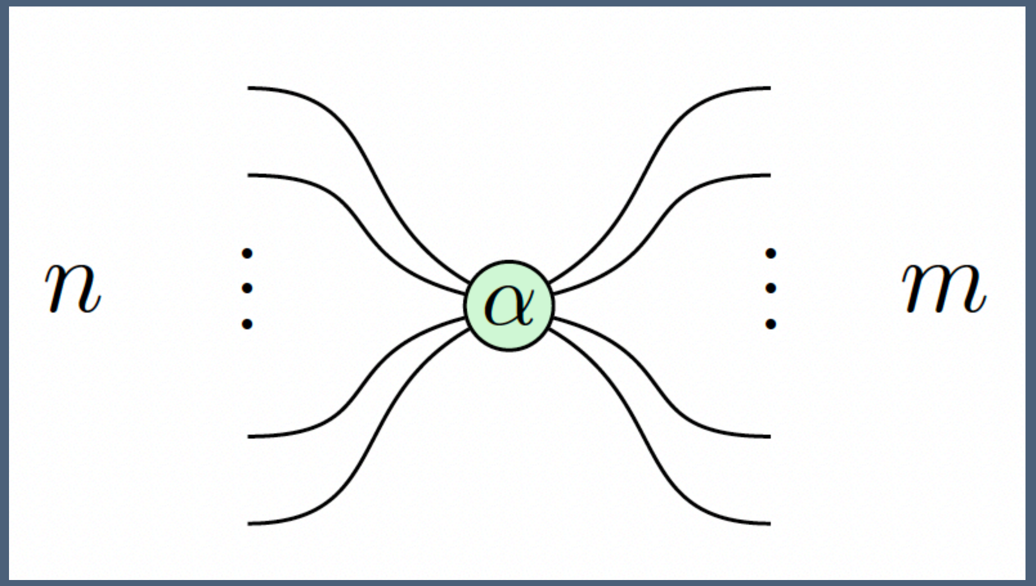


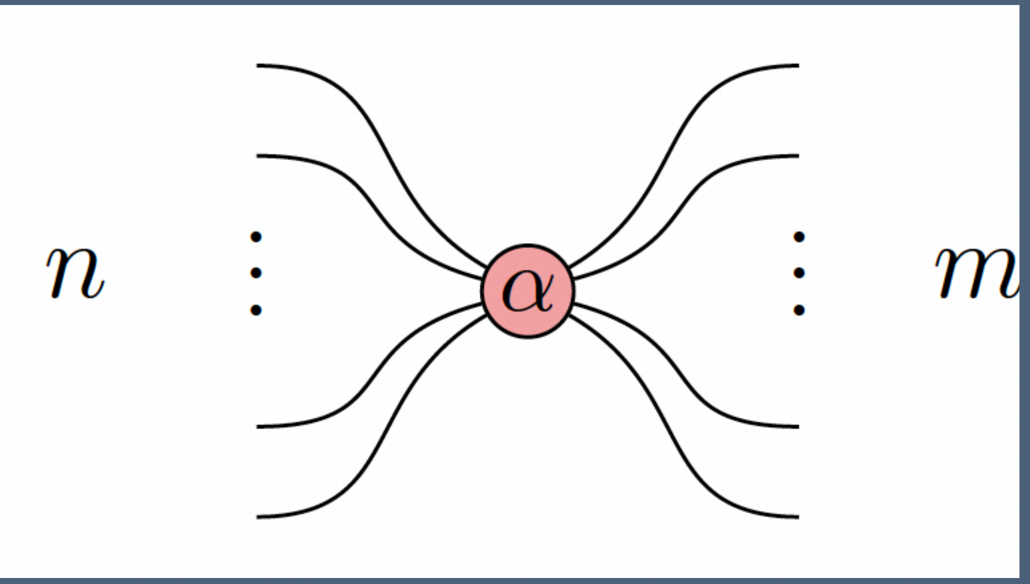
Unit

The ZX-calculus

The ZX-calculus

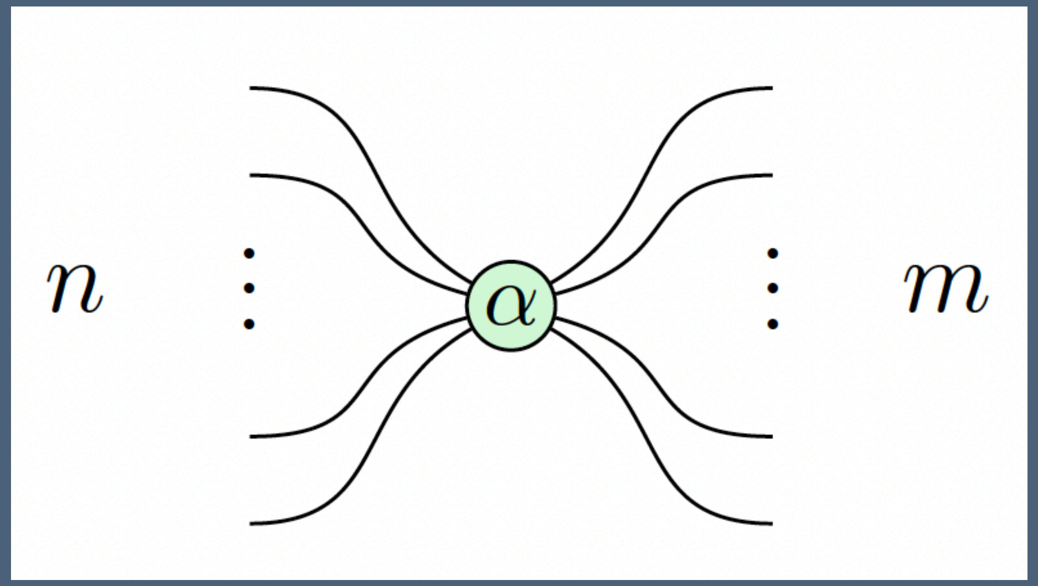
- A complete set of rewrite rules for the manipulation of ZX-diagrams, which are a graphical representation for quantum operations.
- Consists of red and green nodes known as *spiders*.
- A purely diagrammatic language with semantics corresponding to complex matrices.

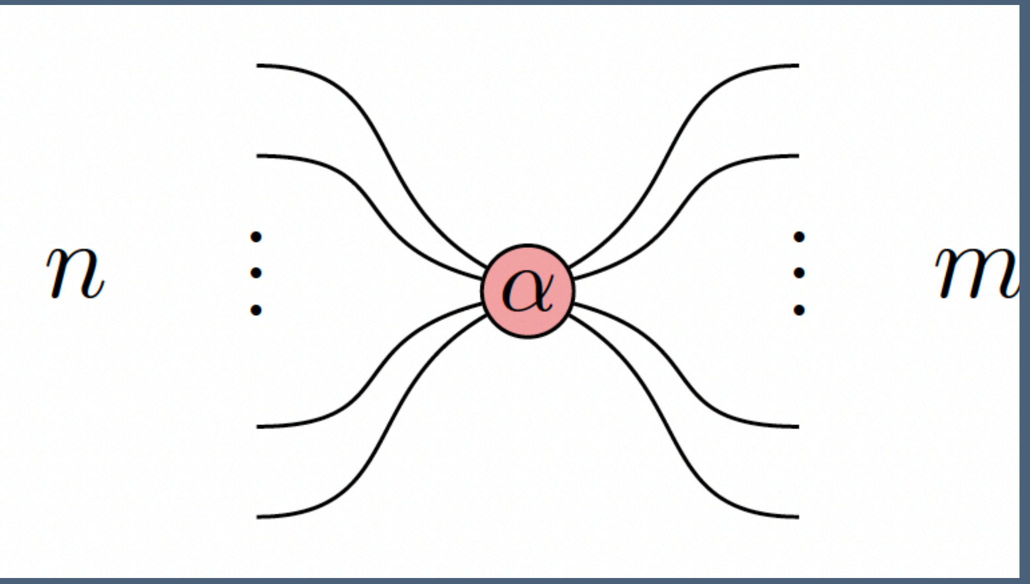




The ZX-calculus

- A complete set of rewrite rules for the manipulation of ZX-diagrams, which are a graphical representation for quantum operations.
- Consists of red and green nodes known as *spiders*.
- A purely diagrammatic language with semantics corresponding to complex matrices.
- <u>*The ZX-calculus forms a dagger compact category.*</u>





Symmetric monoidal + autonomous + dagger.

VyZX

VyZX

Verify the ZX Calculus

- A Coq formalization of the ZX-calculus.
- Uses *inductive* constructors for ZX-diagrams.

$$\frac{\mathrm{in}\ \mathrm{out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{Z}\ \mathrm{in}\ \mathrm{out}\ \alpha : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}} \qquad \frac{}{\mathtt{Cap} : \mathtt{ZX}\ 0\ 2} \qquad \frac{}{\mathtt{Cup} : \mathtt{ZX}\ 2\ 0} \qquad \frac{\mathrm{in}\ \mathrm{out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{X}\ \mathrm{in}\ \mathrm{out}\ \alpha : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}}$$

$$\frac{}{\mathtt{Wire} : \mathtt{ZX}\ 1\ 1} \qquad \frac{}{\mathtt{Box} : \mathtt{ZX}\ 1\ 1} \qquad \frac{}{\mathtt{Swap} : \mathtt{ZX}\ 2\ 2} \qquad \frac{}{\mathtt{Empty} : \mathtt{ZX}\ 0\ 0}$$

$$\frac{\mathtt{zx}_0 : \mathtt{ZX}\ \mathrm{in}\ \mathrm{mid} \qquad \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{mid}\ \mathrm{out}}{\mathtt{Compose}\ \mathtt{zx}_0\ \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}} \qquad \frac{\mathtt{zx}_0 : \mathtt{ZX}\ \mathrm{in}_0\ \mathrm{out}_0 \qquad \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{in}_1\ \mathrm{out}_1}{\mathtt{Stack}\ \mathtt{zx}_0\ \mathtt{zx}_1 : \mathtt{ZX}\ (\mathrm{in}_0 + \mathrm{in}_1)\ (\mathrm{out}_0 + \mathrm{out}_1)}$$

Categorical Concept	Inductive Constructor	Symbol
id_A	Wire	–
I	Empty	∅
∘	Compose	↔
⊗	Stack	↕
Symmetric braid	Swap 1 1	×
Unit	Cap	⊂
Counit	Cup	⊃

Proof assistant shenanigans

Cast

- Dependently typed terms

- The type *ZX 1 2* is not automatically equal to the type *ZX 1 (1 + 1)*.

$$\frac{\mathrm{in}\ \mathrm{out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{Z}\ \mathrm{in}\ \mathrm{out}\ \alpha : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}} \qquad \frac{}{\mathtt{Cap} : \mathtt{ZX}\ 0\ 2} \qquad \frac{}{\mathtt{Cup} : \mathtt{ZX}\ 2\ 0} \qquad \frac{\mathrm{in}\ \mathrm{out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{X}\ \mathrm{in}\ \mathrm{out}\ \alpha : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}}$$

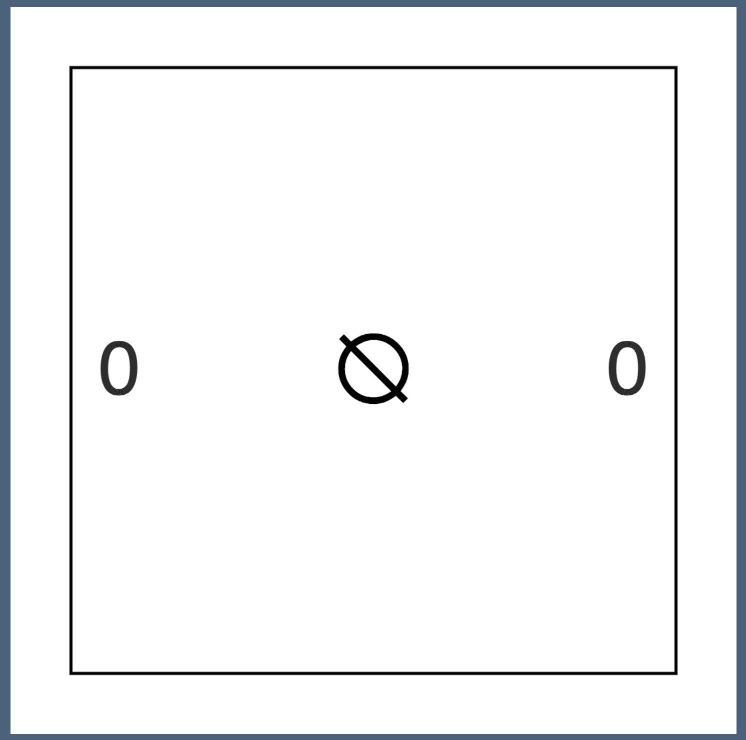
$$\frac{}{\mathtt{Wire} : \mathtt{ZX}\ 1\ 1} \qquad \frac{}{\mathtt{Box} : \mathtt{ZX}\ 1\ 1} \qquad \frac{}{\mathtt{Swap} : \mathtt{ZX}\ 2\ 2} \qquad \frac{}{\mathtt{Empty} : \mathtt{ZX}\ 0\ 0}$$

$$\frac{\mathtt{zx}_0 : \mathtt{ZX}\ \mathrm{in}\ \mathrm{mid} \qquad \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{mid}\ \mathrm{out}}{\mathtt{Compose}\ \mathtt{zx}_0\ \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{in}\ \mathrm{out}} \qquad \frac{\mathtt{zx}_0 : \mathtt{ZX}\ \mathrm{in}_0\ \mathrm{out}_0 \qquad \mathtt{zx}_1 : \mathtt{ZX}\ \mathrm{in}_1\ \mathrm{out}_1}{\mathtt{Stack}\ \mathtt{zx}_0\ \mathtt{zx}_1 : \mathtt{ZX}\ (\mathrm{in}_0 + \mathrm{in}_1)\ (\mathrm{out}_0 + \mathrm{out}_1)}$$

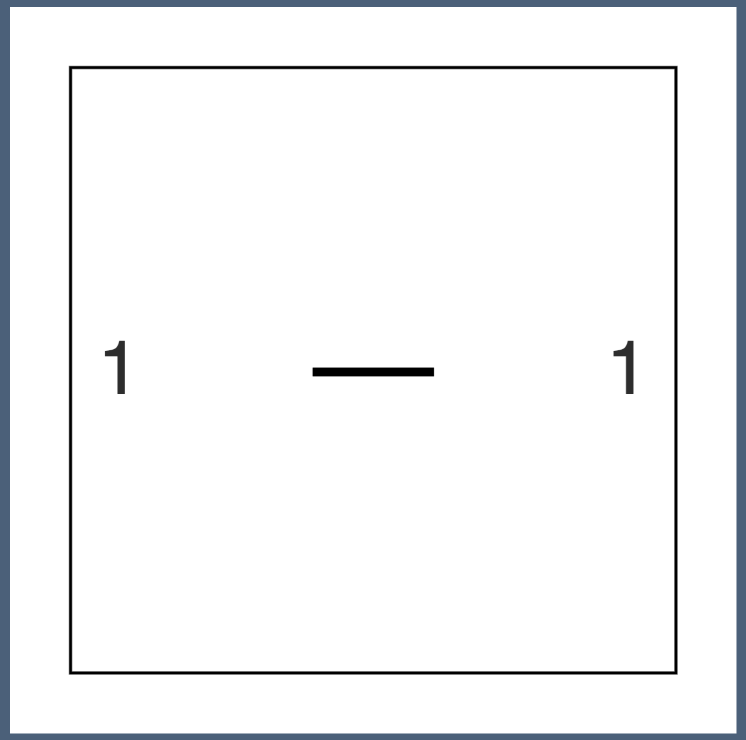
cast (n m : ℕ) {n' m' : ℕ} (prfn : n = n') (prfm : m = m') (zx : ZX n' m') : ZX n m.

VyZX

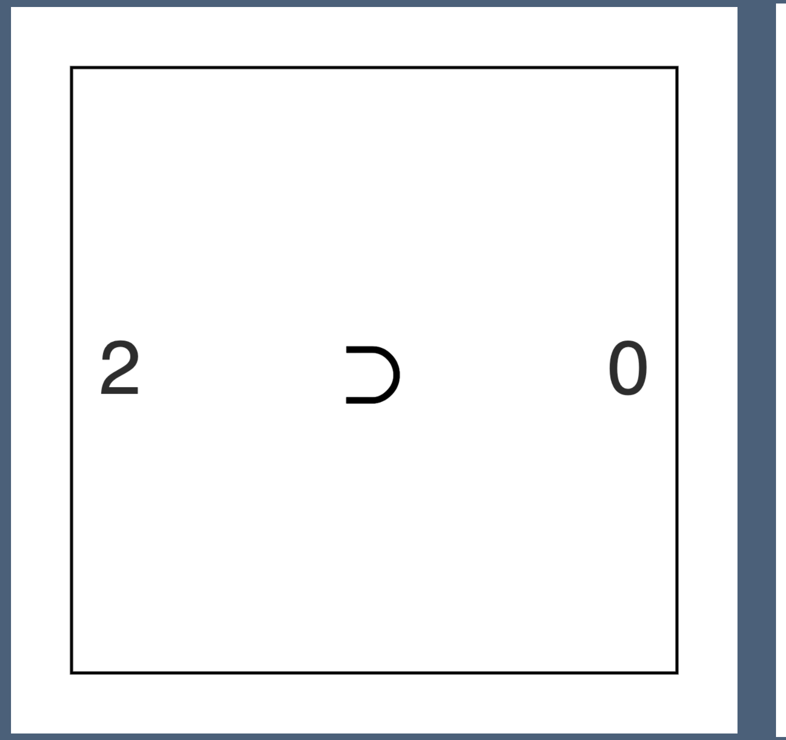
Basic constructors



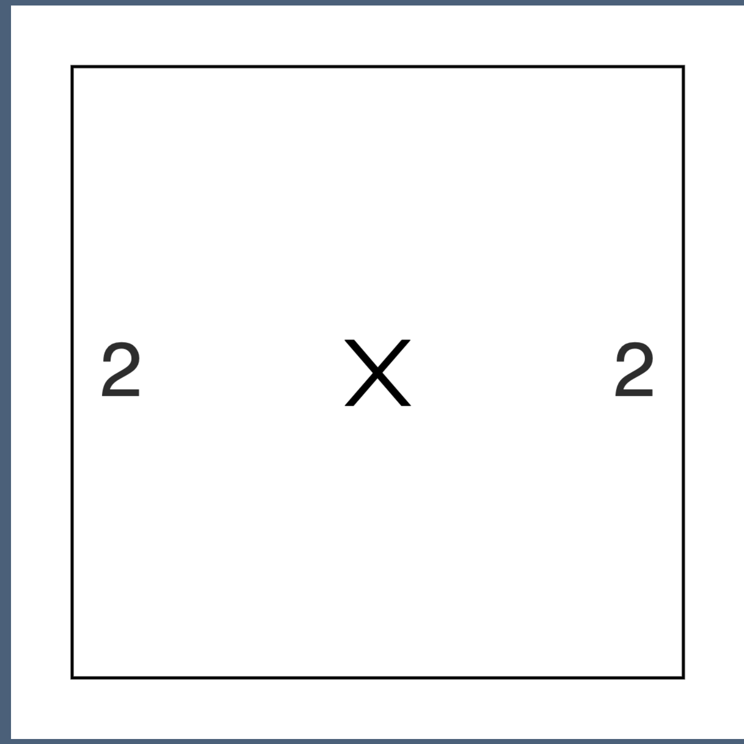
Empty



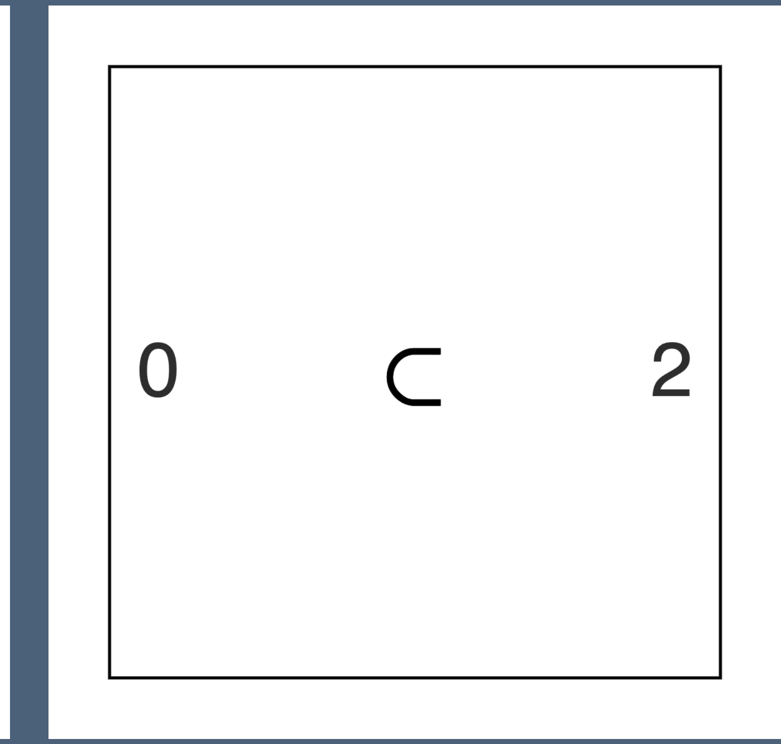
Wire



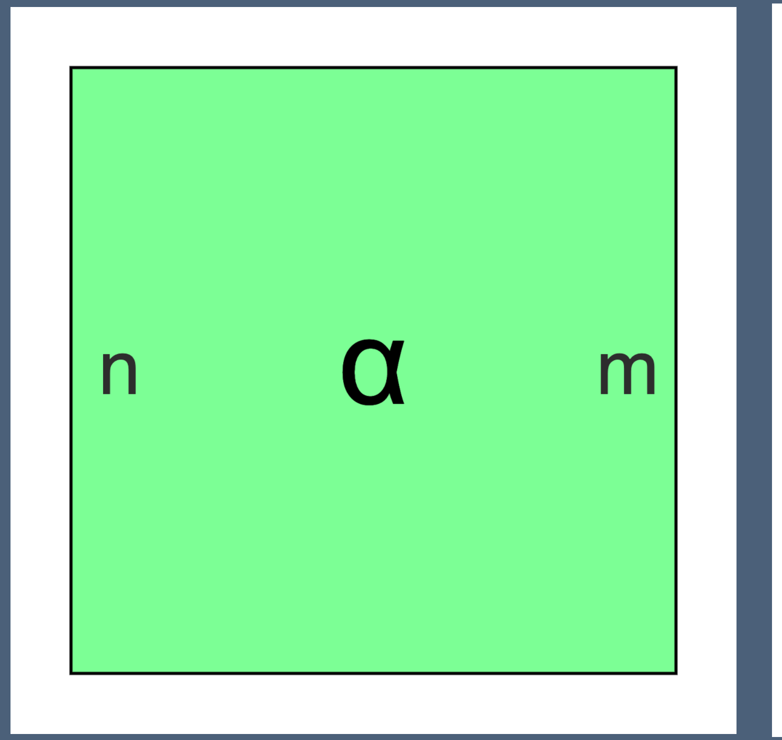
Cup



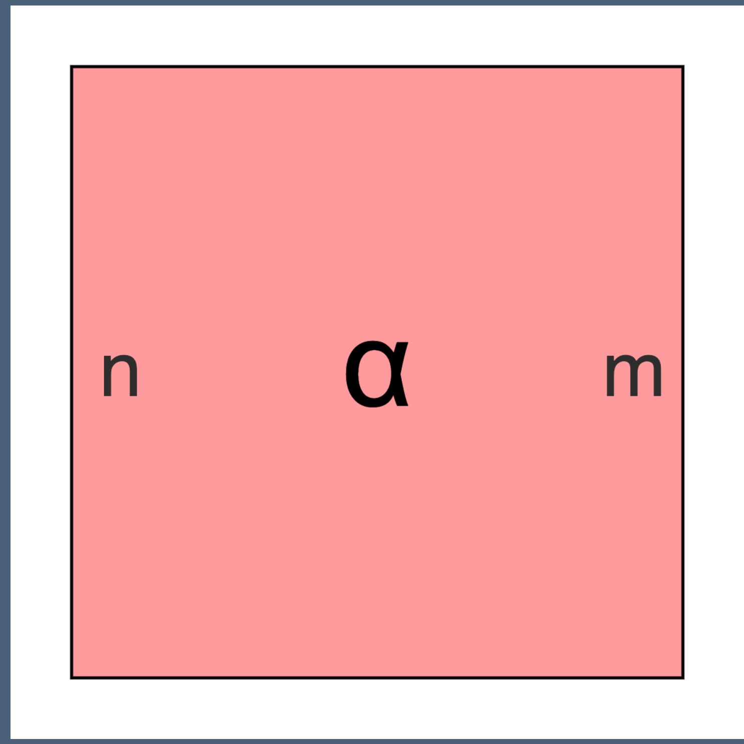
Swap



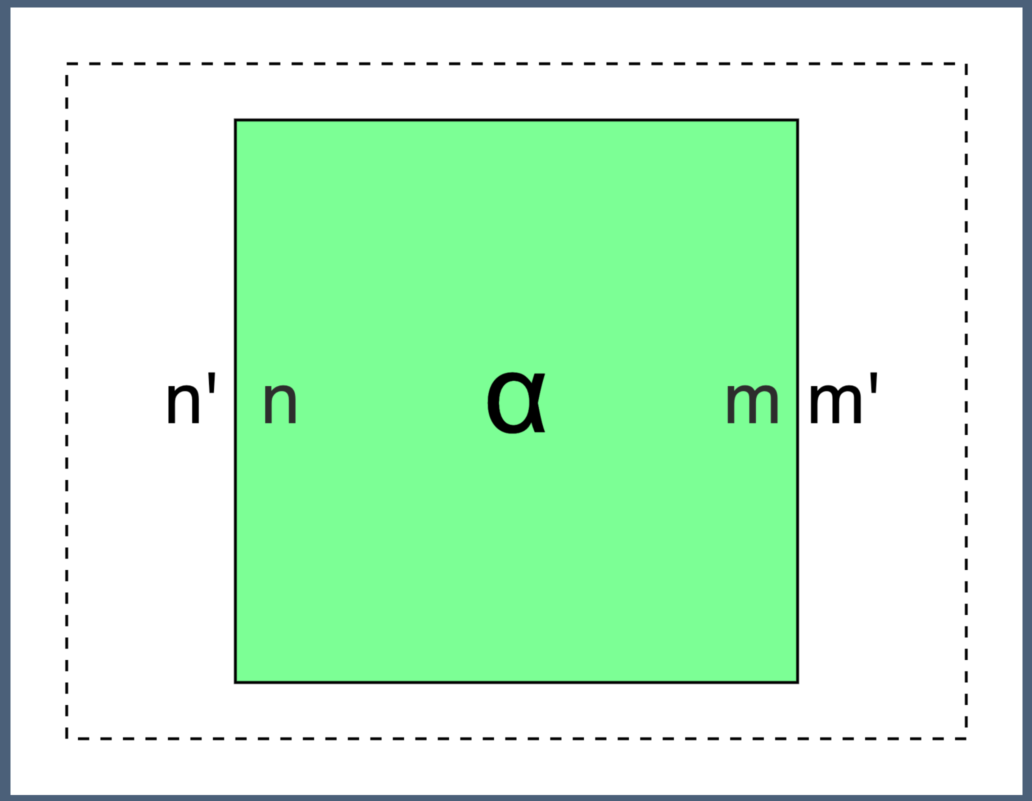
Cap



Green spider



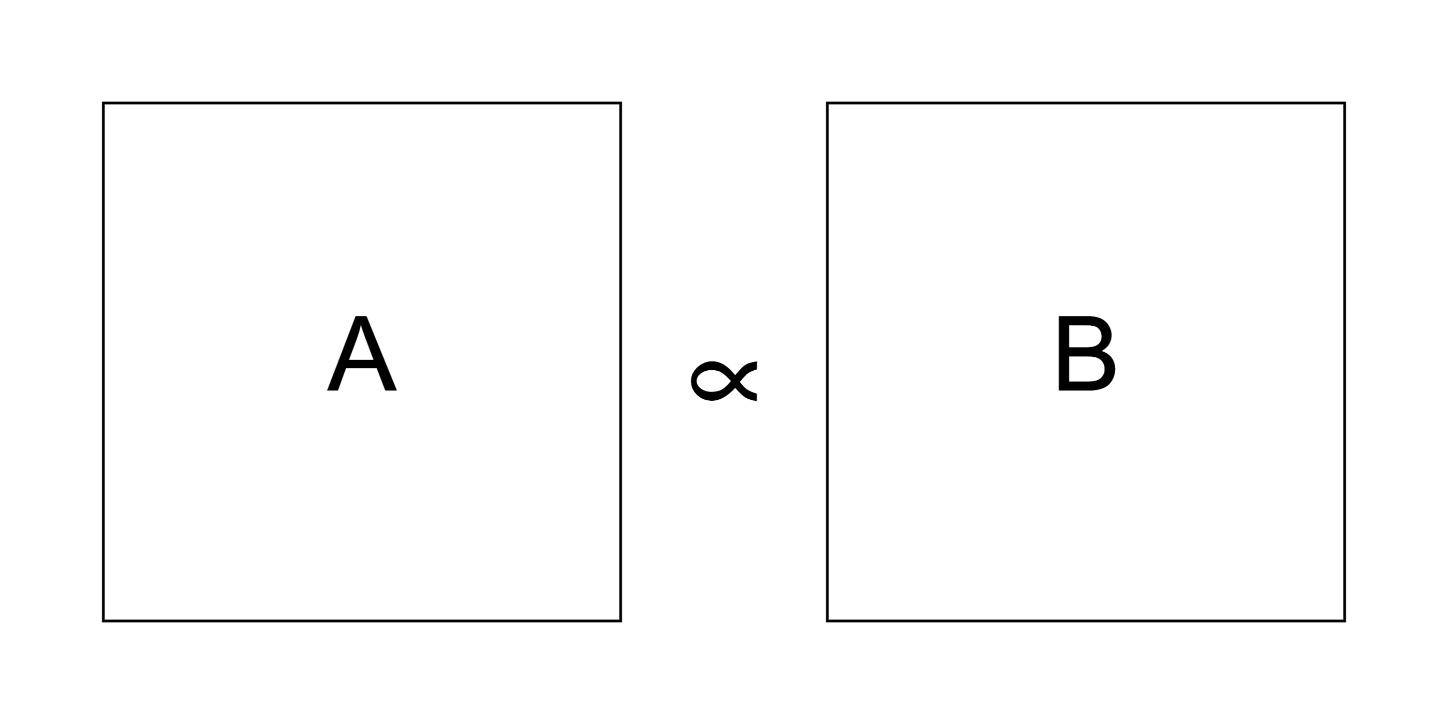
Red spider



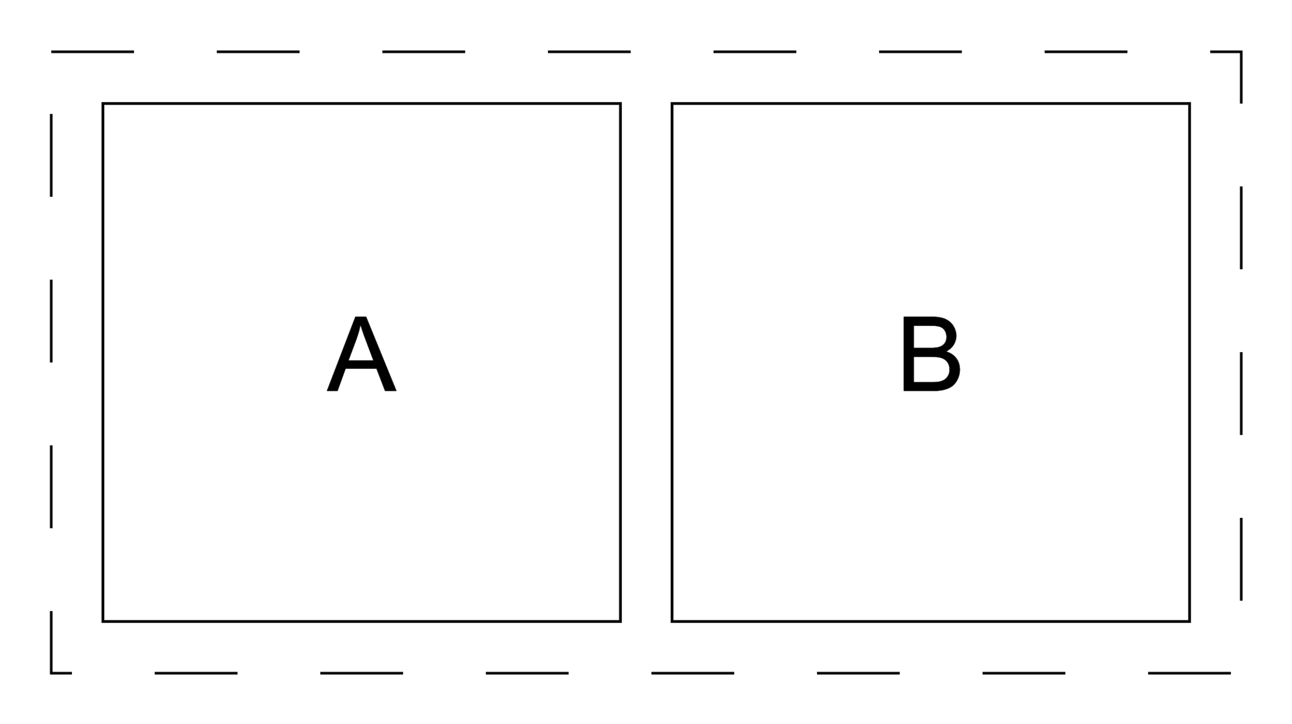
Cast

VyZX

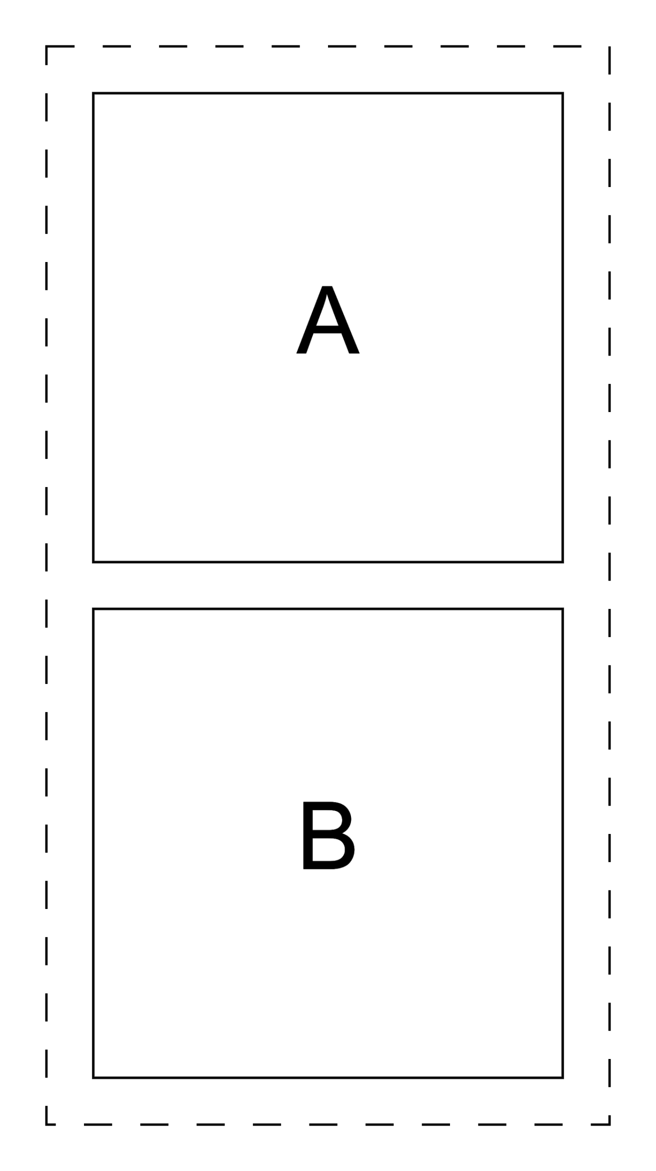
More constructors



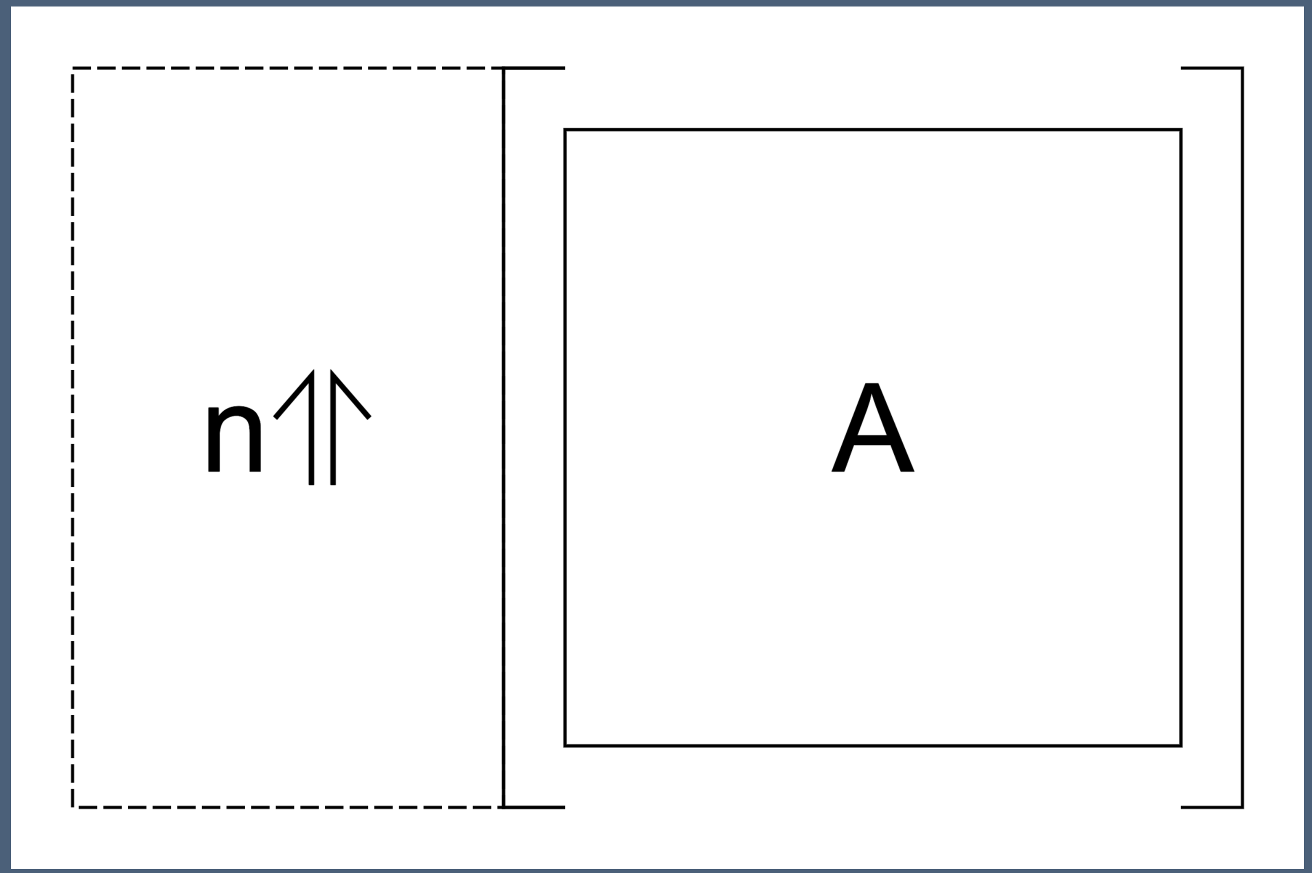
Proportionality



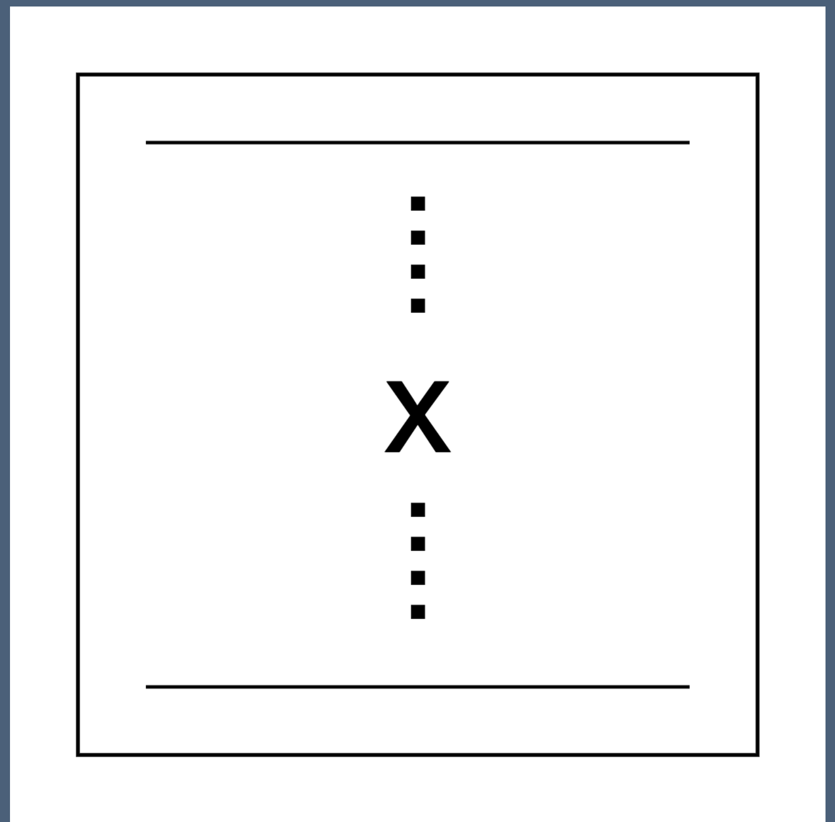
Compose (○)



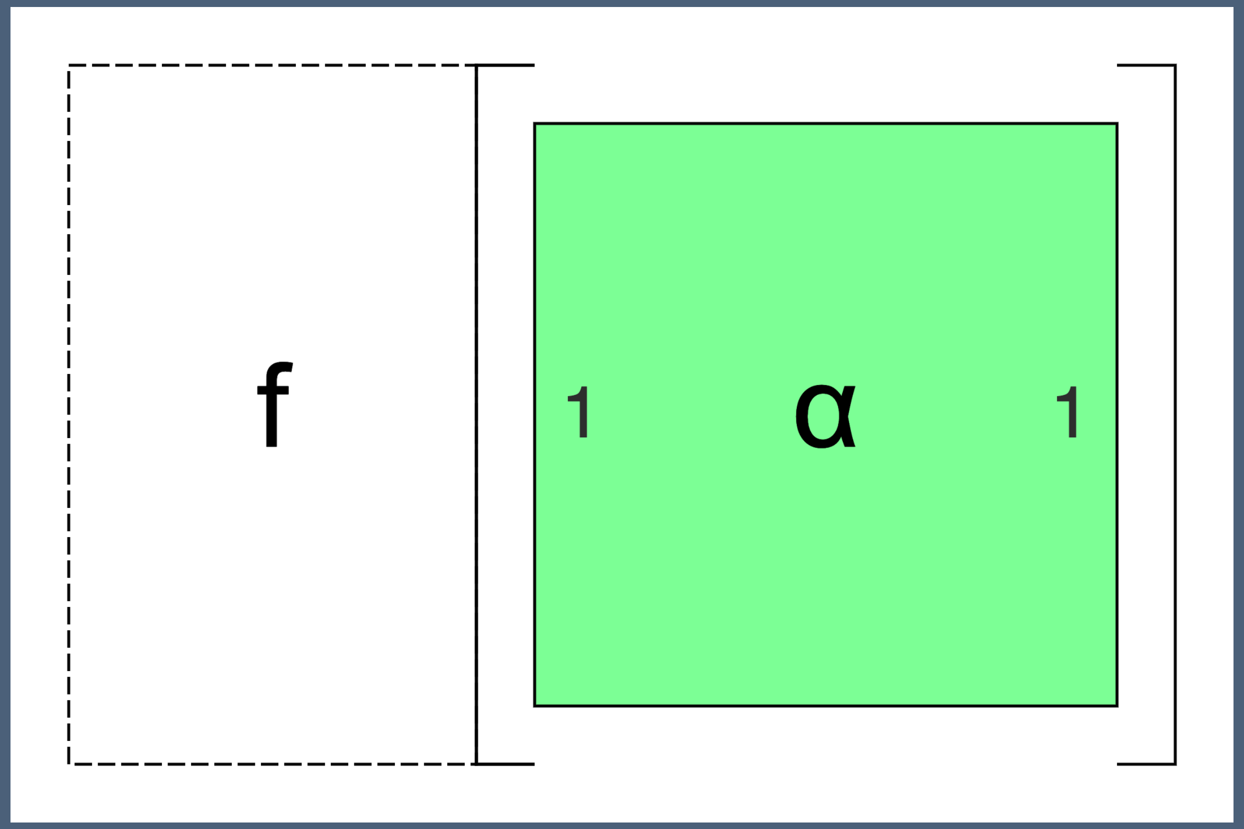
Stack (⊗)



n stack, general



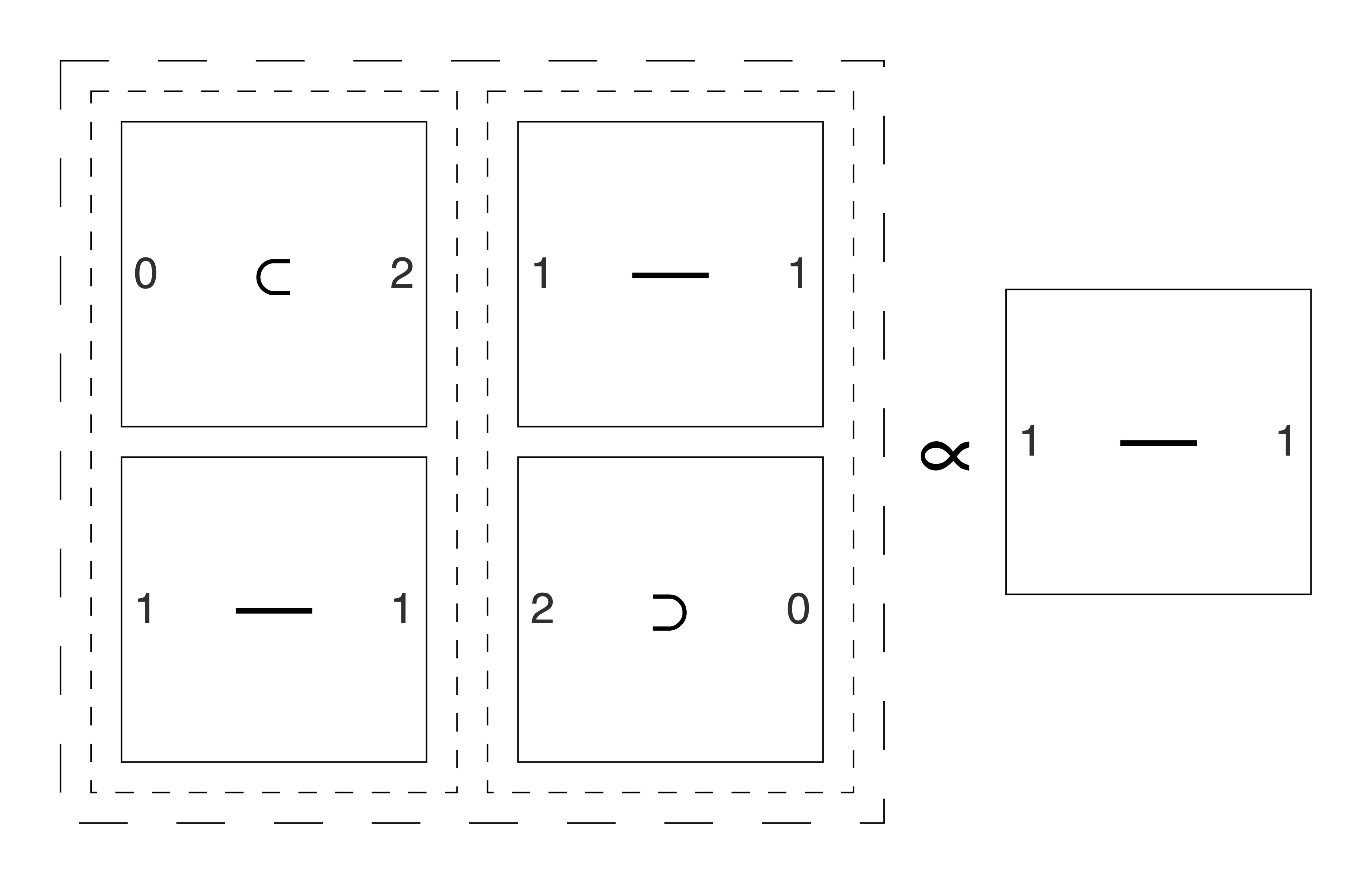
n wires stacked

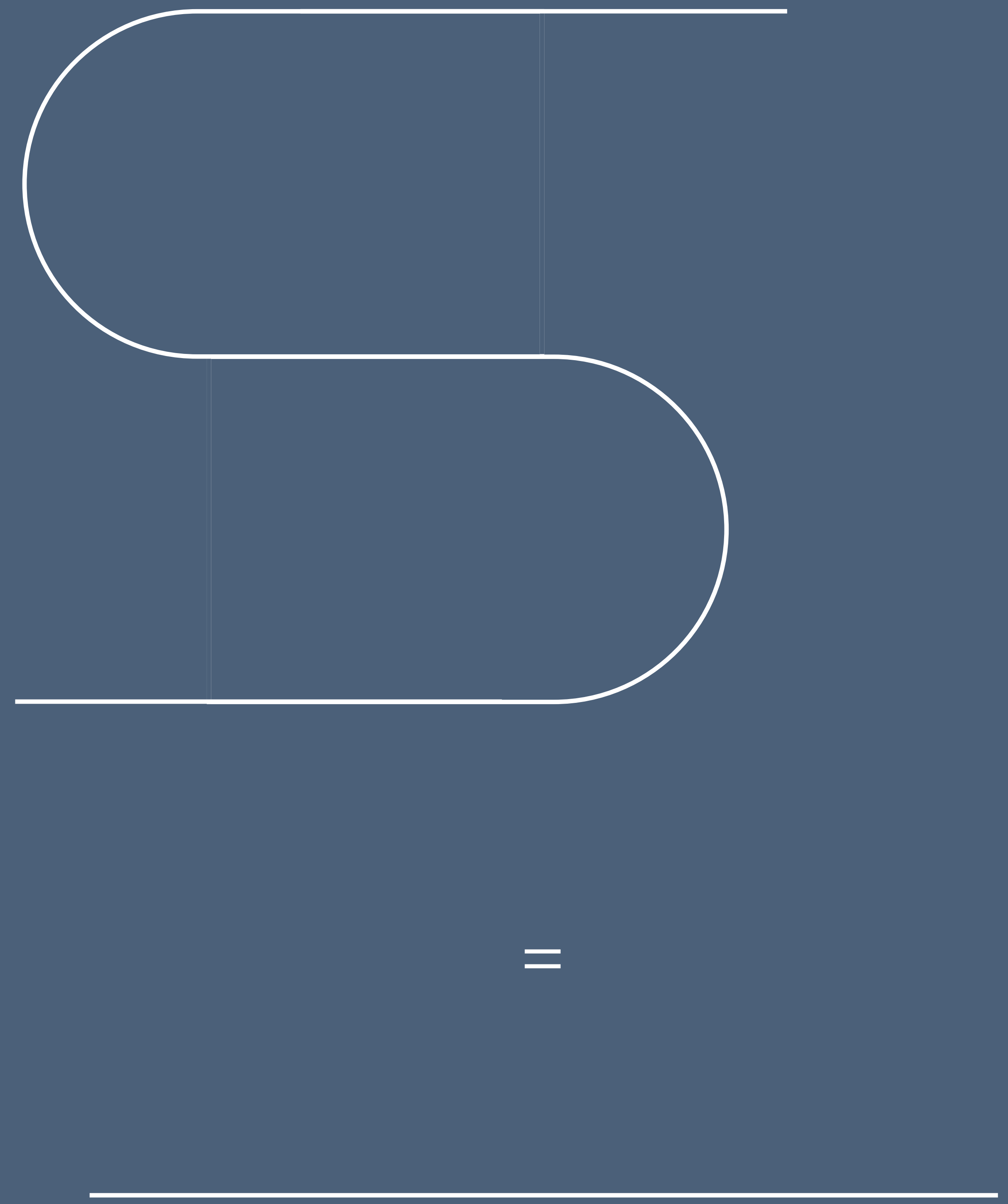


Function / transform

More structure, more explicitly

Foliation

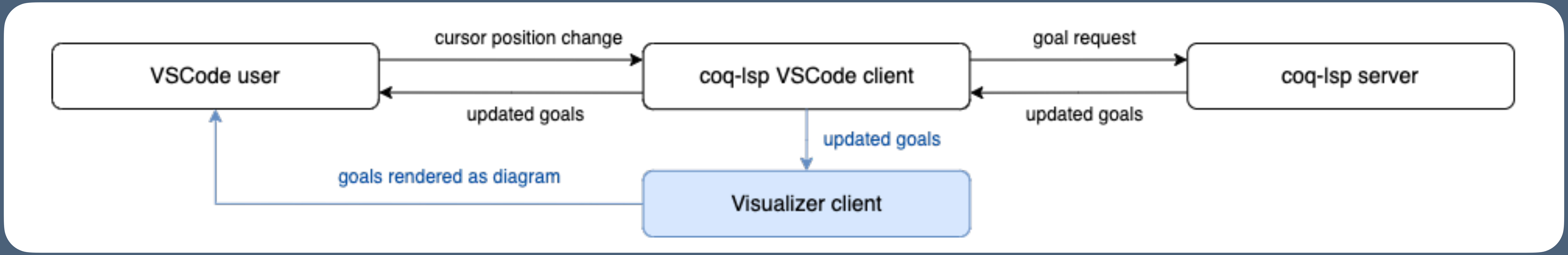


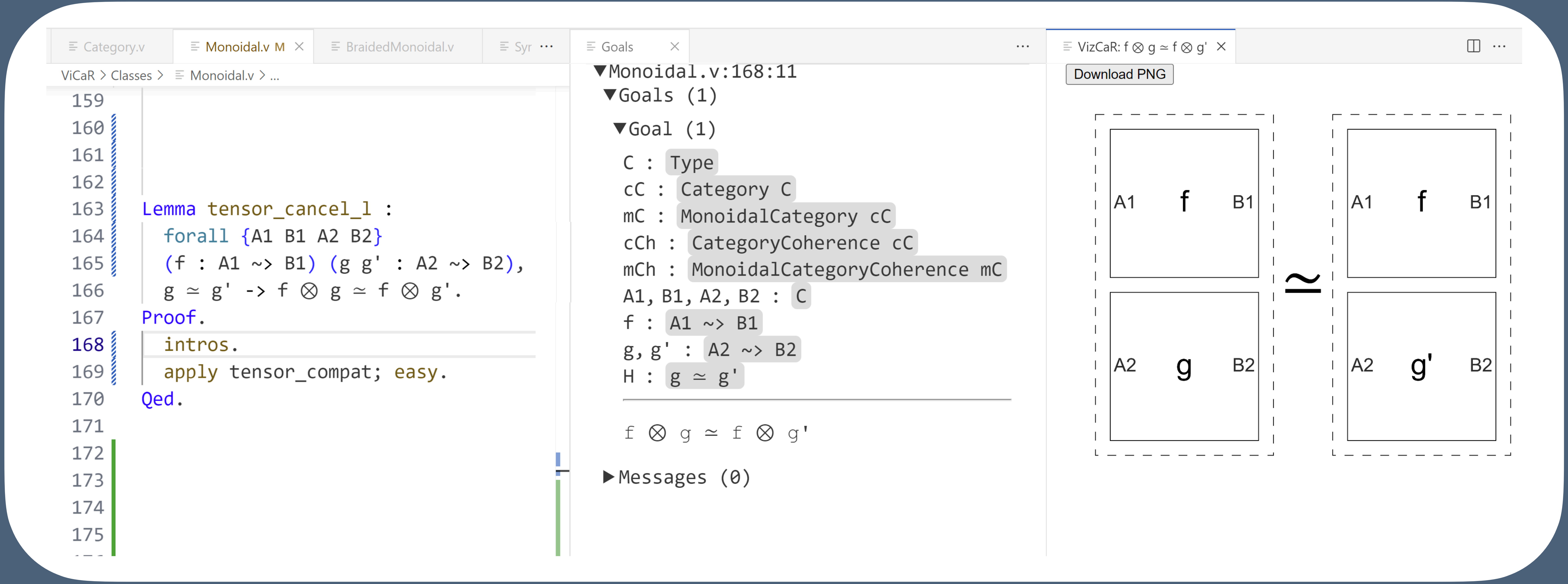


Visualization Workflow

Workflow

Using coq-lsp

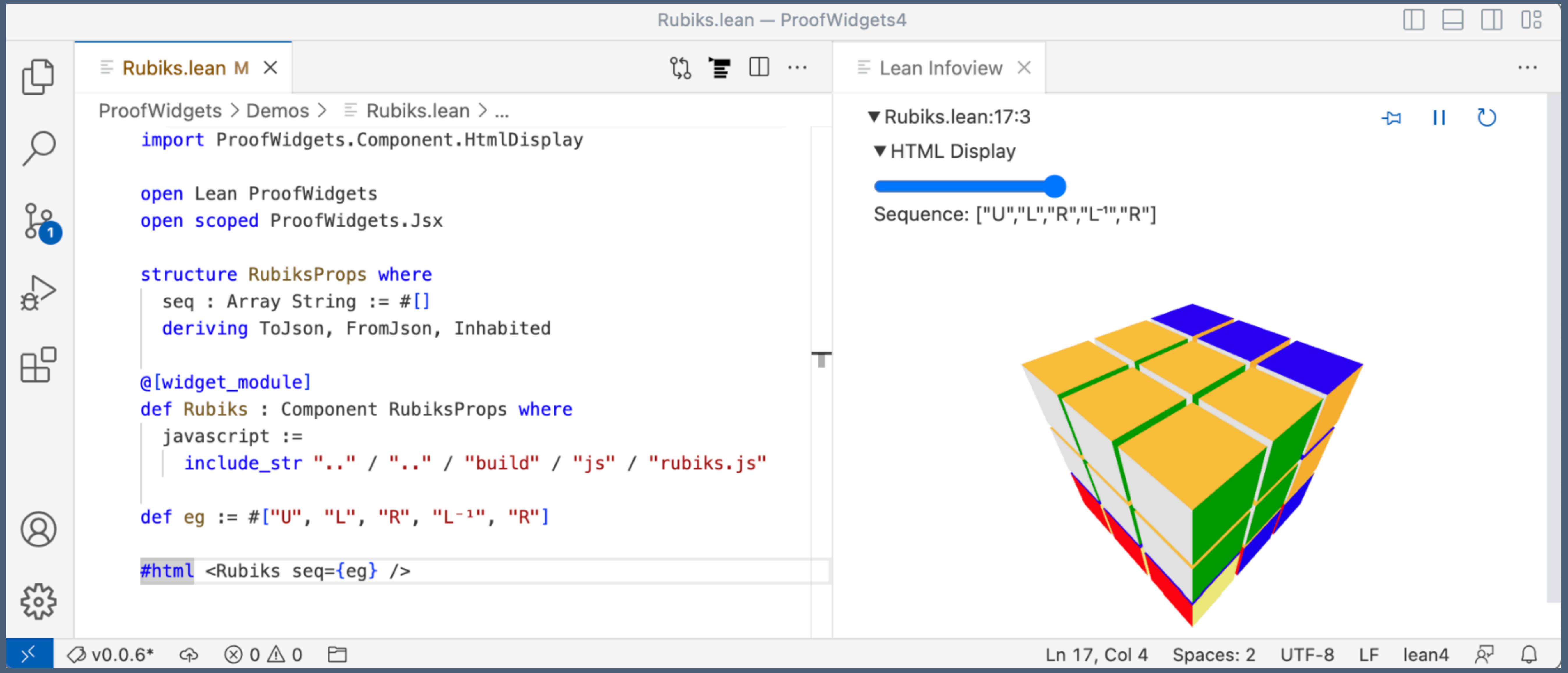


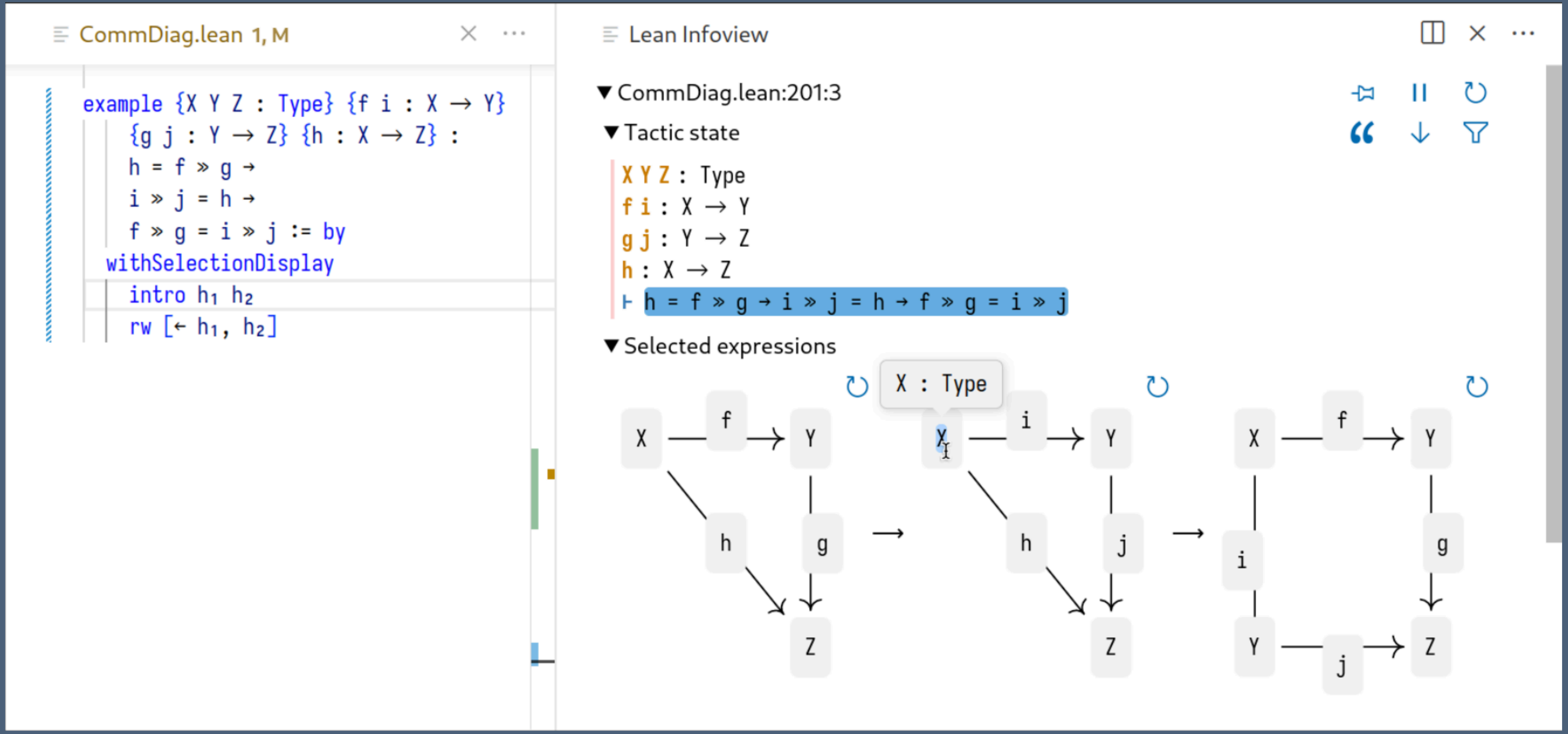


Related Work

ProofWidgets

Nawrocki et al.

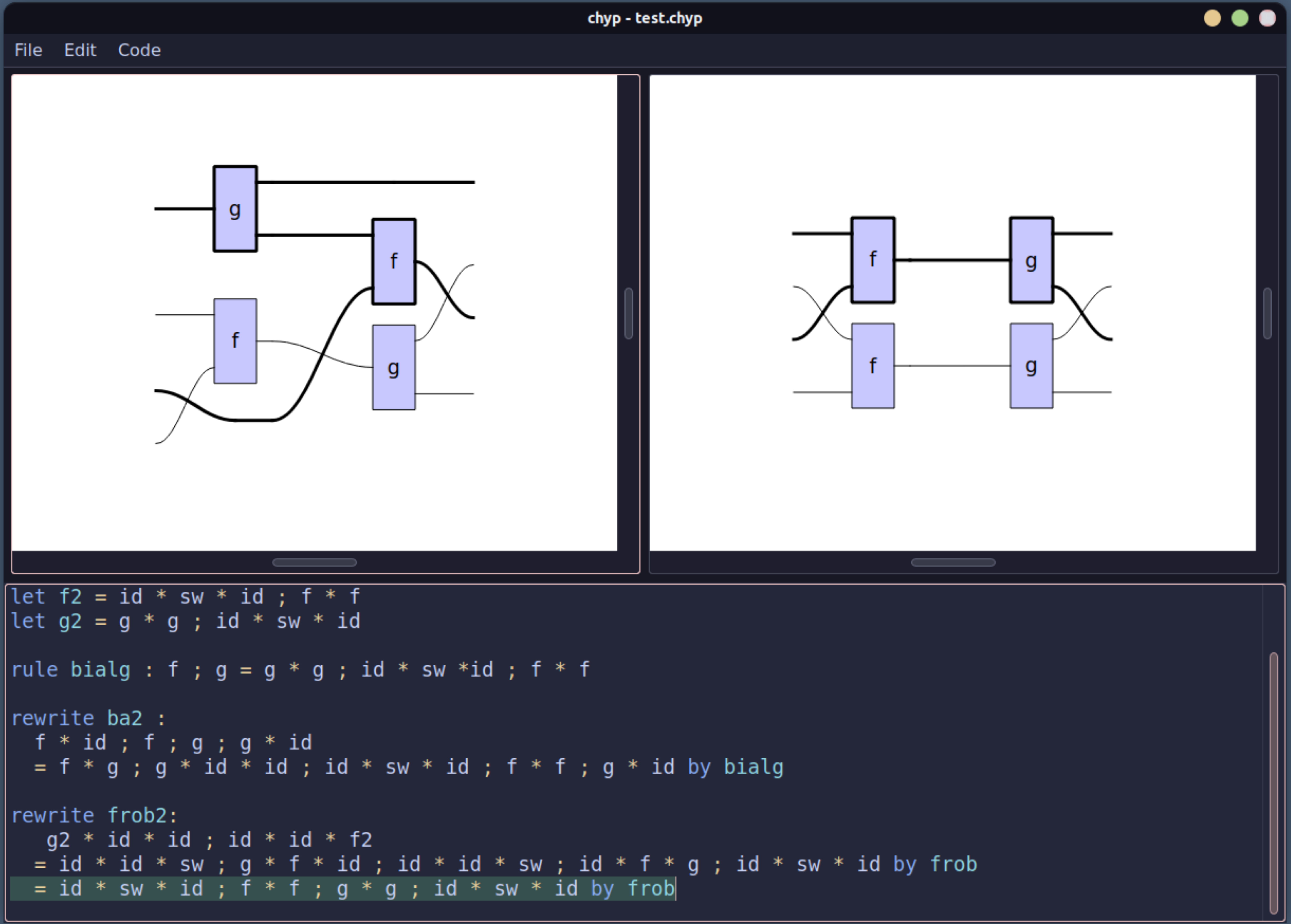





```
import ProofWidgets.Component.HtmlDisplay

open Lean ProofWidgets
open scoped ProofWidgets.Jsx

structure RubiksProps where
  seq : Array String := #[]
  deriving ToJson, FromJson, Inhabited

@[widget_module]
def Rubiks : Component RubiksProps where
  javascript := include_str ".." / ".." / ".lake" / "build" / "js" / "rubiks.js"

def eg := #["L", "L", "D⁻¹", "U⁻¹", "L", "D", "D", "L", "U⁻¹", "R", "D", "F", "F", "D"]

#html <Rubiks seq={eg} />
```

Chyp

Kissinger et al.



Future Work

Customizable visualization

ViZX++

- ZX-calculus visualizer = specialized, distinct implementation,
- Intractable method for future instantiations.
- User-specified custom directives for a set of structural constructs.

Interactive visualization

- Diagrammatic rewriting, graphically
- Bidirectional text/graphic system

Conclusion

Conclusion

Proof Visualization for Graphical Structures

- A methodology for working with graphical constructs in a proof assistant,
- An implementation of visualizations for the graphical language of string diagrams associated with classes of categories,
- An instantiation for the ZX-calculus, a symmetric monodical autonomous category,
- An integration with the proof assistant Coq.