ViCAR: Visualizing Categories for Automated Rewriting

Bhakti Shah, **William Spencer**, Laura Zielinski, Ben Caldwell, Adrian Lehmann, Robert Rand

THE UNIVERSITY OF CHICAGO

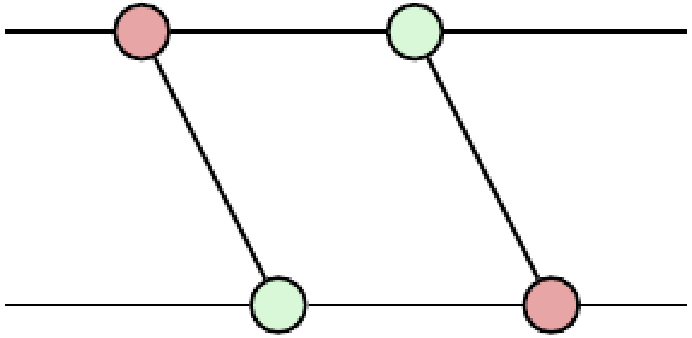
Coq

- Proof assistant
- Machine-checked proofs
  - High confidence, high detail
- Tactic-based, like written proof
- Automation for repetitive tasks

```
Theorem mul_comm : forall m n : nat,
  m * n = n * m.
Proof.
  assert (H:forall n k: nat, n*(S(k))= n + n * k).
  { intros n k. induction n.
  - reflexivity.
  - simpl. rewrite IHn. rewrite add_assoc. rewrite (add_assoc n k (n*k)).
    rewrite (add_comm k n). reflexivity.  }
  intros m n. induction n.
  - rewrite mul_0_r. reflexivity.
  - rewrite (H m n). simpl. rewrite IHn. reflexivity.
Qed.
```

VyZX

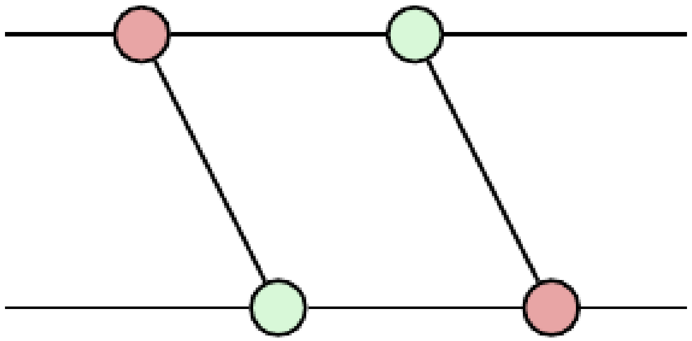
- Formalization of ZX Calculus in Coq
  - System of rewrite rules for ZX Diagrams
  - String diagram representation
  - ZX Diagrams form a monoidal category over $\mathbb{N}$
- Want to preserve visual nature of ZX Calculus



VyZX

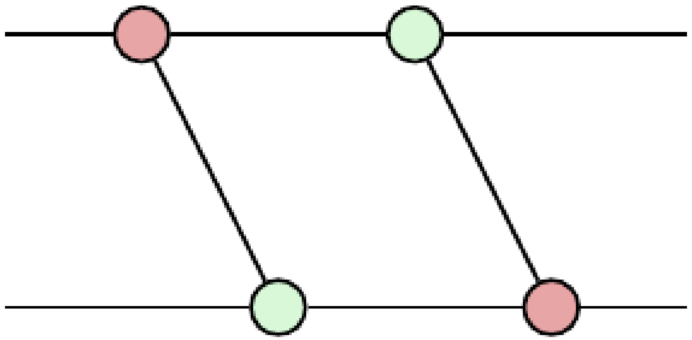
- Formalization of ZX Calculus in Coq
  - System of rewrite rules for ZX Diagrams
  - String diagram representation
  - ZX Diagrams form a monoidal category over ℕ
- Want to preserve visual nature of ZX Calculus
- Terms can get unwieldy!

```
Z (S n) (S m) α ⟷ (Z 1 2 0
↕ n_wire m) ∝ — ↕ Z n (S (S
 (S m))) α ⟷ (⊃ ↕ n_wire S
                  (S m))
```

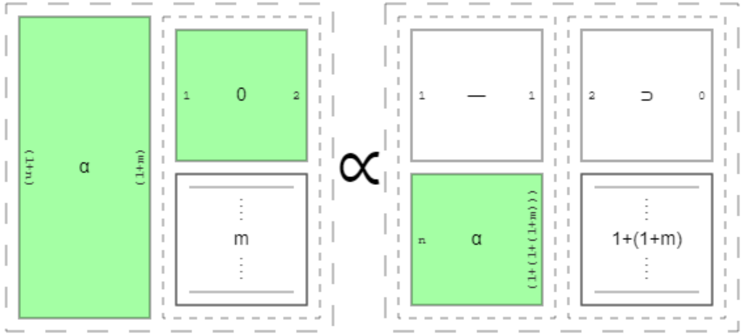


VyZX

- Formalization of ZX Calculus in Coq
  - System of rewrite rules for ZX Diagrams
  - String diagram representation
  - ZX Diagrams form a monoidal category over ℕ
- Want to preserve visual nature of ZX Calculus
- Terms can get unwieldy!



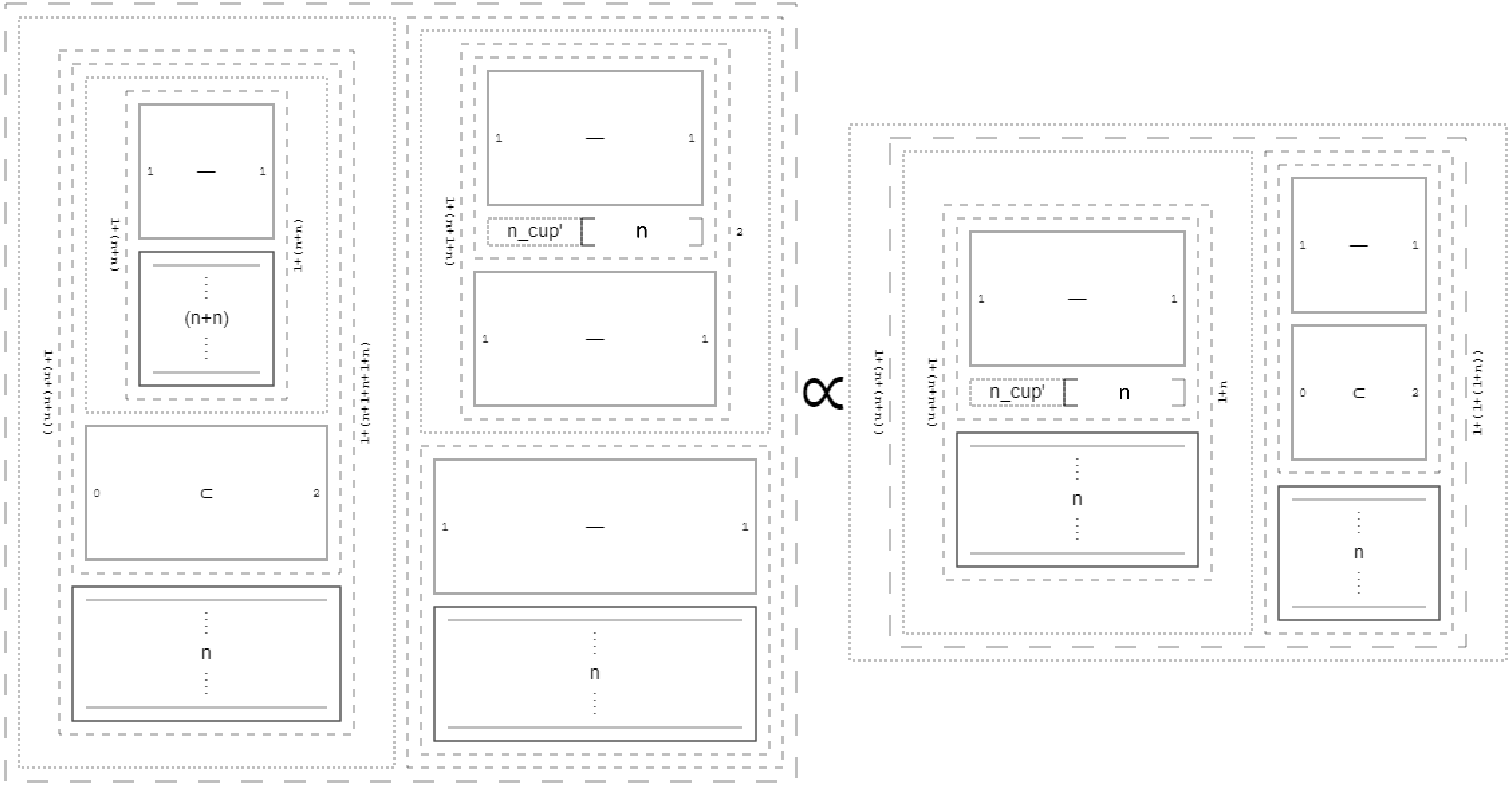
```
Z (S n) (S m) α ⟷ (Z 1 2 0
↕ n_wire m) ∝ — ↕ Z n (S (S
 (S m))) α ⟷ (⊃ ↕ n_wire S
              (S m))
```



Why Visualize?

```
$ S (n + (n + n)), S (n + S n + S n) ::: $ S (n +
n), S (n + n) ::: — ↕ (n_wire n ↕ n_wire n) $ ↕ ⊂ ↕
n_wire n $ ↔ ($ S (n + S n), 2 ::: — ↕ n_cup' n ↕ —
$ ↕ (— ↕ n_wire n)) ∝ $ S (n + (n + n)), S (S (S n))
::: — ↕ (n_wire n ↕ n_wire n) ↕ n_wire n ↔ $ S (n +
n + n), S n ::: — ↕ n_cup' n ↕ n_wire n $ ↔ (— ↕ ⊂
↕ n_wire n) $
```

Why Visualize?



Monoidal Categories in Proof Assistants

- Pervasive:
  - Matrices
  - STLC
  - Causal separation diagrams
  - Algebraic reasoning
- Awkward:
  - Need to keep around structural information
  - Don’t get nice string diagram representation

Overview

- What is ViCAR?
- Visualization
- Automation
- Diagrammatic Reasoning?

ViCAR

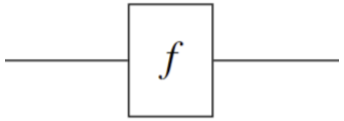
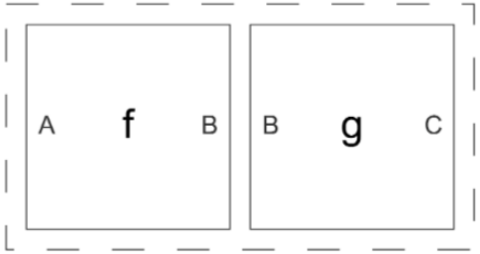
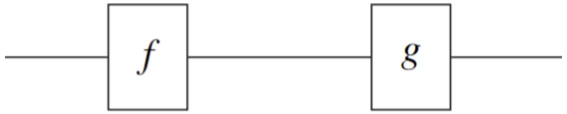
- Common framework for reasoning about monoidal categories in Coq
- Collection of typeclasses describing categorical structure
  - Instantiated with information describing particular (monoidal) category
  - Data and coherence split into separate typeclasses

Categories

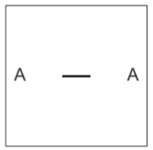
```
Class Category (C : Type) := {
    morphism (A B : C) : Type
      where "A ~> B" := (morphism A B);

    (* Morphism equivalence *)
    c_equiv {A B : C} : relation (A ~> B)
      where "f ≃ g" := (c_equiv f g);

    compose {A B M : C} :
      (A ~> B) -> (B ~> M) -> (A ~> M)
      where "f ∘ g" := (compose f g);
      (* Diagrammatic compose *)

    c_identity (A : C) : A ~> A
      where "id_ A" := (c_identity A);
}.
```

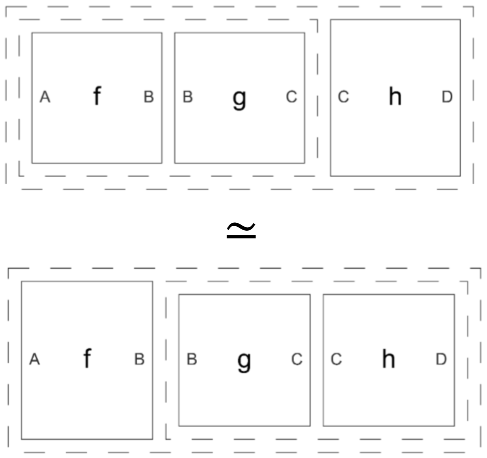
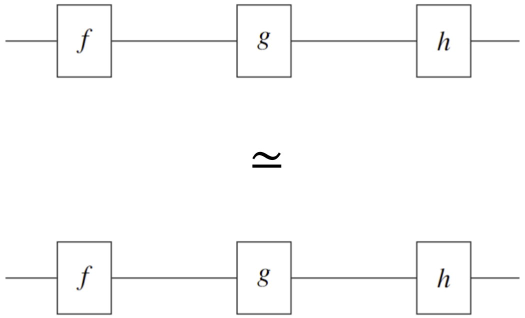
```
Class CategoryCoherence {C} (cC : Category C) := {
  c_equiv_rel {A B : C} :
    equivalence (A ~> B) cC.(c_equiv);

  compose_compat {A B M : C}
    (f g : A ~> B) (h j : B ~> M) :
      f ≃ g -> h ≃ j ->
      f ∘ h ≃ g ∘ j;

  assoc {A B M N : C} (f : A ~> B)
    (g : B ~> M) (h : M ~> N) :
      (f ∘ g) ∘ h ≃ f ∘ (g ∘ h);

  left_unit {A B : C} (f : A ~> B) :
    id_ A ∘ f ≃ f;
  right_unit {A B : C} (f : A ~> B) :
    f ∘ id_ B ≃ f;
}.
```

Graphical Language for Categories

Term	Visualization	String Diagram
`f : A ~> B`		
`f ∘ g`		

Graphical Language for Categories

Term	Visualization	String Diagram
`id_ A`	 	

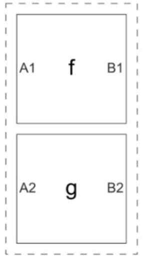
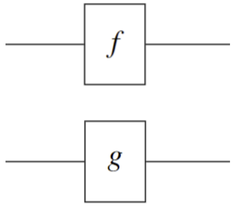
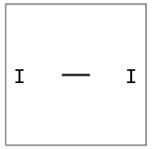
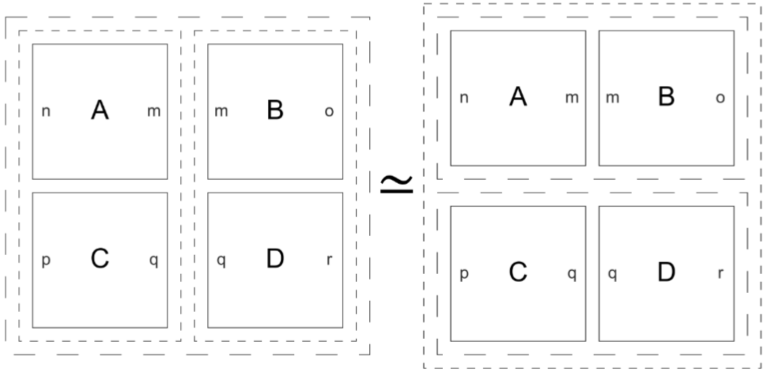
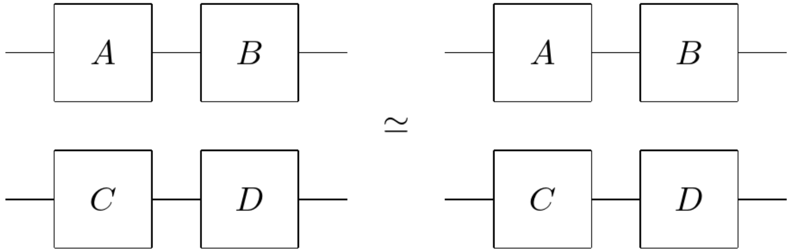
`f ∘ g ∘ h`
≃
`f ∘ (g ∘ h)`
	
	
		

Monoidal Categories

```
Class MonoidalCategory {C} (cC : Category C) := {
  obj_tensor : C -> C -> C
    where "x × y" := (obj_tensor x y);
  mor_tensor {A1 B1 A2 B2 : C}
    (f : A1 ~> B1) (g : A2 ~> B2) :
    A1 × A2 ~> B1 × B2
    where "f ⊗ g" := (mor_tensor f g);
  mon_I : C
    where "I" := (mon_I);

  associator (A B M : C) :
    (A × B) × M <~> A × (B × M)
    where "α_ A, B, M" := (associator A B M);

  left_unitor (A : C) : mon_I × A <~> A
    where "λ_ A" := (left_unitor A);

  right_unitor (A : C) : A × mon_I <~> A
    where "ρ_ A" := (right_unitor A);
}.
```

```
Class MonoidalCategoryCoherence {C} {cC : Category C}
  {cCh : CategoryCoherence cC} (mC : MonoidalCategory cC)  := {
  tensor_id (A1 A2 : C) : (id_ A1) ⊗ (id_ A2) ≃ id_ (A1 × A2);
  tensor_compose {A1 B1 M1 A2 B2 M2 : C}
    (f1 : A1 ~> B1) (g1 : B1 ~> M1)
    (f2 : A2 ~> B2) (g2 : B2 ~> M2) :
    (f1 ∘ g1) ⊗ (f2 ∘ g2) ≃ f1 ⊗ f2 ∘ g1 ⊗ g2;
  tensor_compat {A1 B1 A2 B2 : C}
    (f f' : A1 ~> B1) (g g' : A2 ~> B2) :
    f ≃ f' -> g ≃ g' -> f ⊗ g ≃ f' ⊗ g';

  (* Naturality conditions for α, λ, ρ *)
  associator_cohere {A B M N P Q : C}
    (f : A ~> B) (g : M ~> N) (h : P ~> Q) :
    α_ A, M, P ∘ (f ⊗ (g ⊗ h)) ≃ ((f ⊗ g) ⊗ h) ∘ α_ B, N, Q;
  left_unitor_cohere {A B : C} (f : A ~> B) :
    λ_ A ∘ f ≃ (id_ I ⊗ f) ∘ λ_ B;
  right_unitor_cohere {A B : C} (f : A ~> B) :
    ρ_ A ∘ f ≃ (f ⊗ id_ I) ∘ ρ_ B;

  (* Coherence conditions *)
  triangle (A B : C) :
    α_ A, I, B ∘ (id_ A ⊗ λ_ B) ≃ ρ_ A ⊗ id_ B;
  pentagon (A B M N : C) :
    (α_ A, B, M ⊗ id_ N) ∘ α_ A, (B × M), N ∘ (id_ A ⊗ α_ B, M, N)
    ≃ α_ (A × B), M, N ∘ α_ A, B, (M × N);
}.
```

Graphical Language for Monoidal Categories

<table>
<tr><th>Term</th><th>Visualization</th><th>String Diagram</th></tr>
<tr><td>$f \otimes g$</td><td>
</td><td>
</td></tr>
<tr><td>id_ I</td><td>
</td><td></td></tr>
<tr><td>$A \otimes C \circ B \otimes D$
$\simeq$
$(A \circ B) \otimes (C \circ D)$</td><td>
</td><td>
</td></tr>
</table>

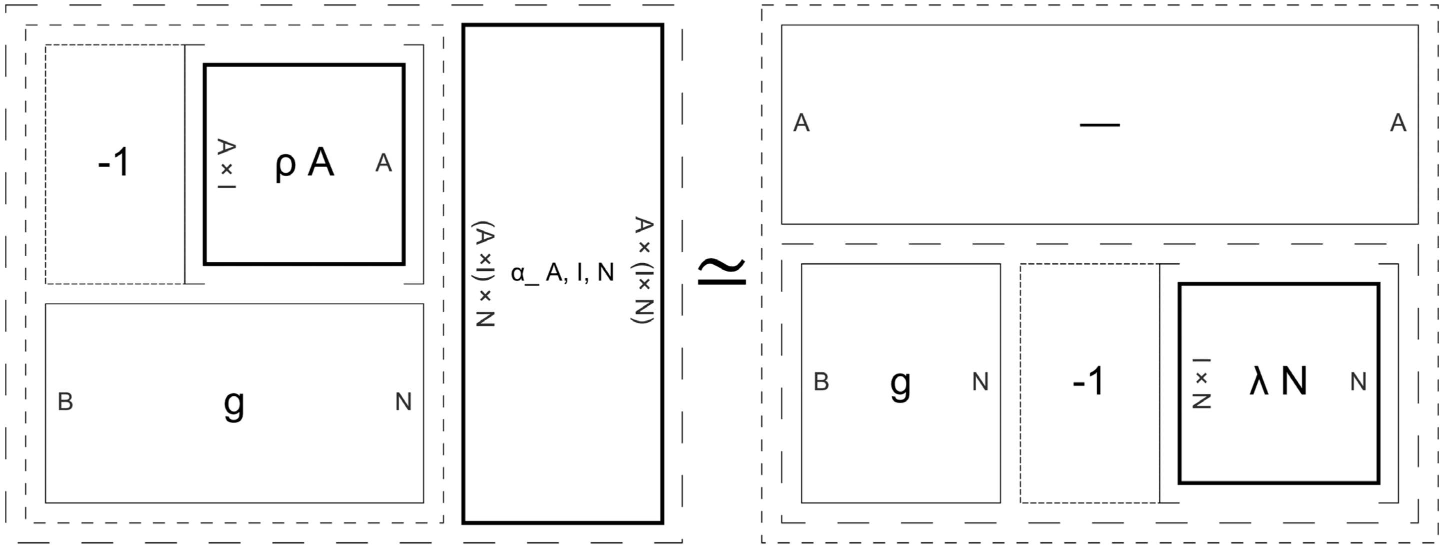
VisCAR

- With typeclass instances declared, can visualize a term in any monoidal category in this style

VisCAR

- With typeclass instances declared, can visualize a term in any monoidal category in this style

ρ_ A ^-1 ⨂ g ∘ α_ A, I, N ≃ id_ A ⨂ (g ∘ λ_ N ^-1)



VisCAR

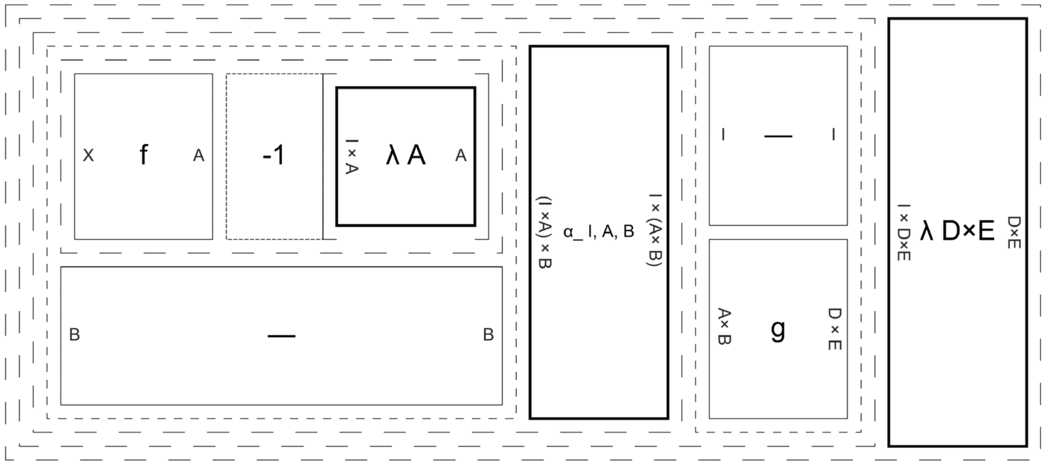
- With typeclass instances declared, can visualize a term in any monoidal category in this style

```
(f ∘ λ_ A ⁻¹) ⨂ id_ B ∘ α_ I,
A, B ∘ id_ I ⨂ g ∘ λ_ (D × E)
```

VisCAR

- With typeclass instances declared, can visualize a term in any monoidal category in this style

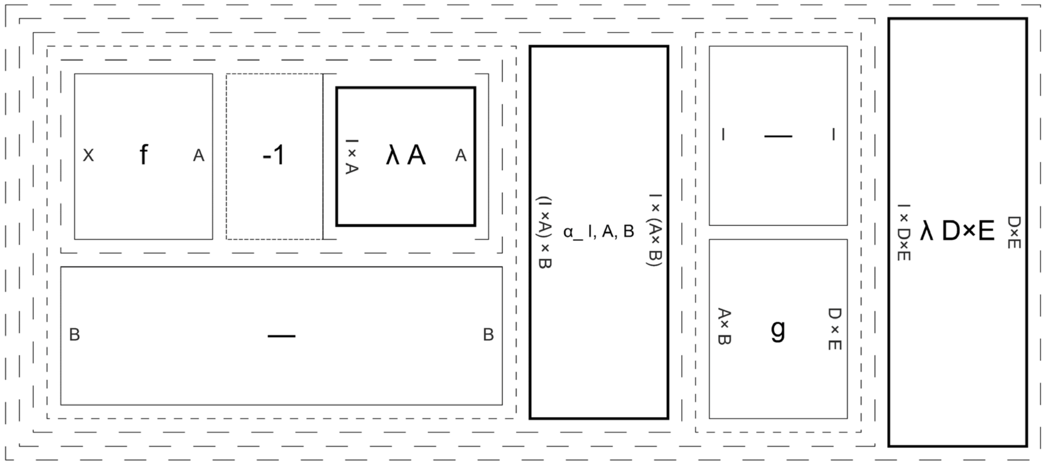
```
(f ∘ λ_ A ⁻¹) ⨂ id_ B ∘ α_ I,
A, B ∘ id_ I ⨂ g ∘ λ_ (D × E)
```



VisCAR

- With typeclass instances declared, can visualize a term in any monoidal category in this style
- … but lots of visual noise required by structure

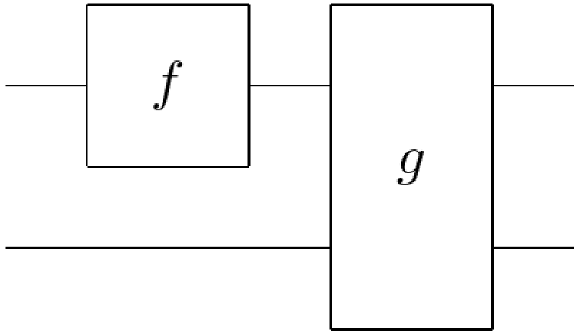
```
(f ∘ λ_ A ⁻¹) ⨂ id_ B ∘ α_ I,
A, B ∘ id_ I ⨂ g ∘ λ_ (D × E)
```



VisCAR

- With typeclass instances declared, can visualize a term in any monoidal category in this style
- … but lots of visual noise required by structure

```
(f ∘ λ_ A ⁻¹) ⨂ id_ B ∘ α_ I,
A, B ∘ id_ I ⨂ g ∘ λ_ (D × E)
```



Automation

- Coq “tactics” — mini-programs that try to advance proof
- Bridge the gap between structural definition and diagrammatic reasoning

Associativity

- `rassoc, lassoc`
- `cancel_isos` : cancel isomorphisms with inverses, remove units
- `assoc_rw` : reassociate to rewrite with an existing lemma

Associativity

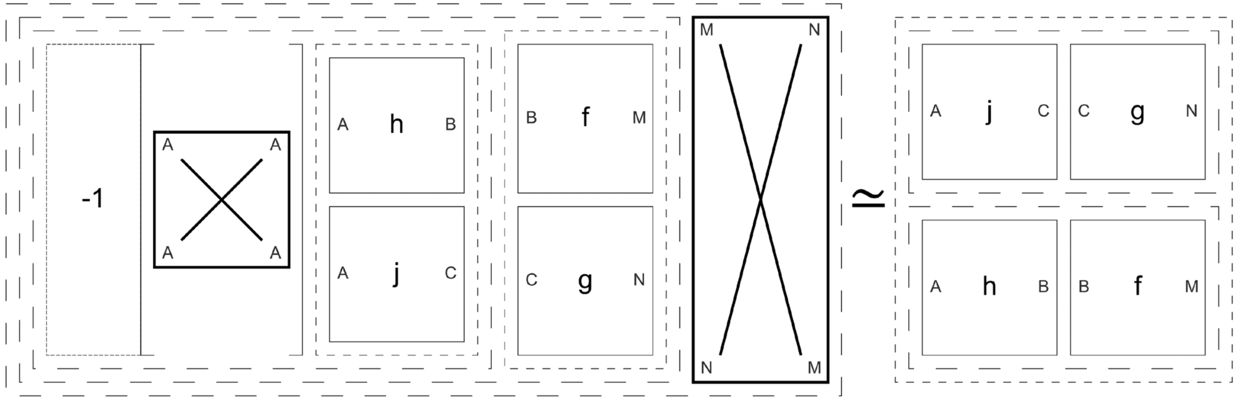
- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```

Associativity

- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

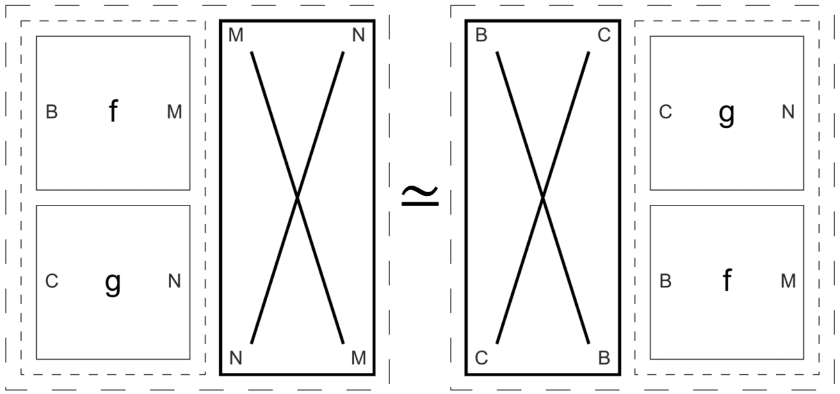
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Associativity

- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

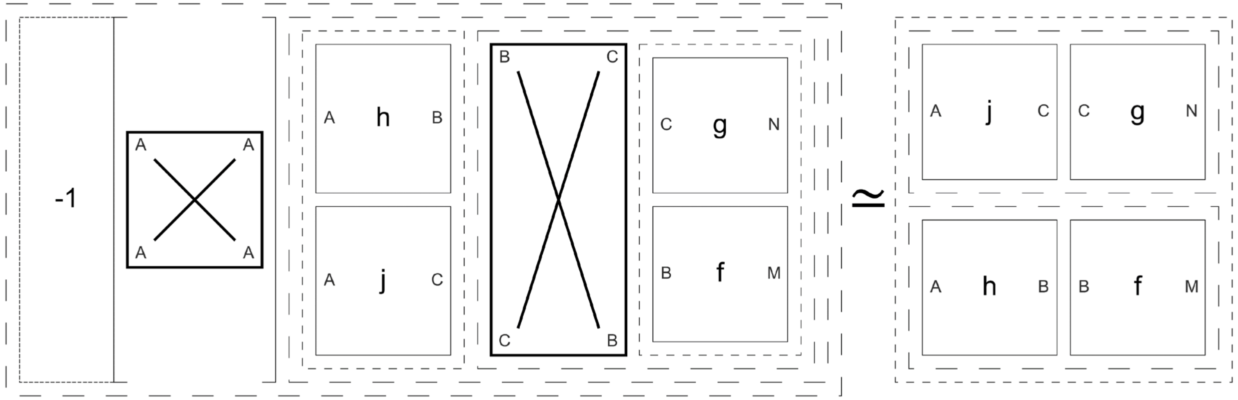
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Associativity

- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

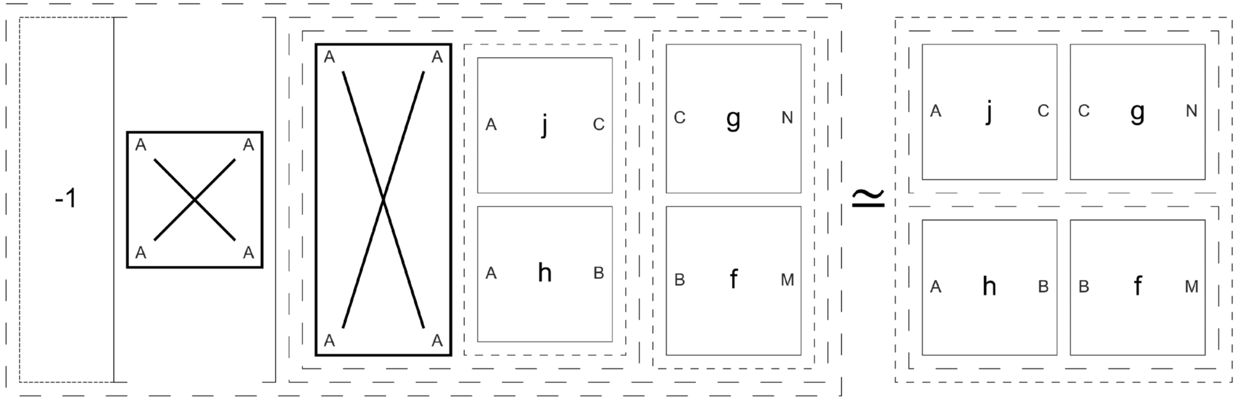
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Associativity

- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

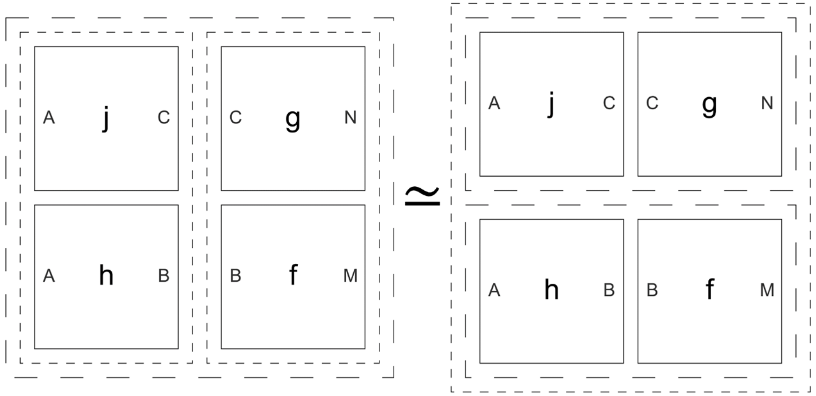
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Associativity

- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

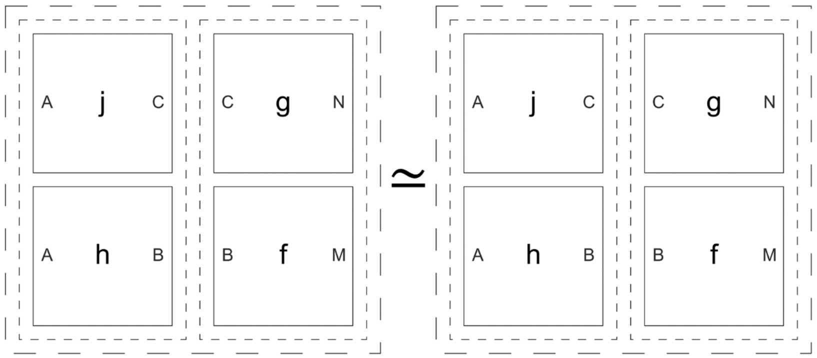
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Associativity

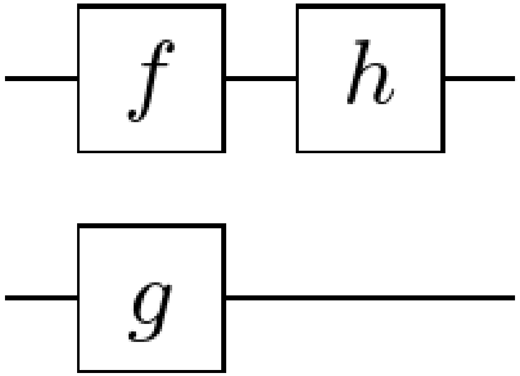
- `rassoc, lassoc`
- `assoc_rw` : reassociate to rewrite with an existing lemma
- `cancel_isos` : cancel isos with inverses, remove units

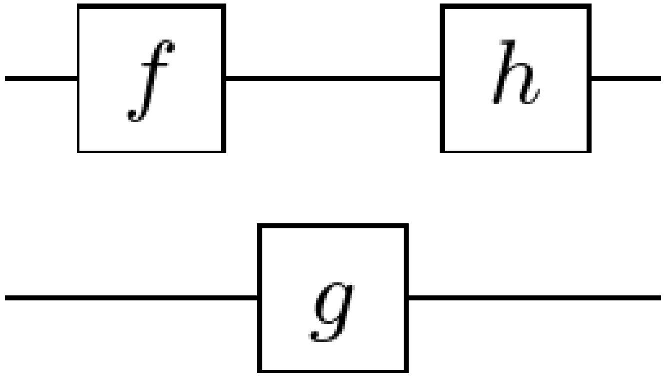
```
Lemma assoc_rw_example {A B C M N : CC}
  (h : A ~> B) (j: A ~> C) (f : B ~> M) (g : C ~> N) :
  (β_ _, _)^-1 ∘ h ⊗ j ∘ f ⊗ g ∘ β_ _, _
  ≃ (j ∘ g) ⊗ (h ∘ f).
Proof.
  assoc_rw braiding_natural.
  assoc_rw braiding_natural.
  cancel_isos.
  rewrite tensor_compose.
  reflexivity.
Qed.
```



Foliation

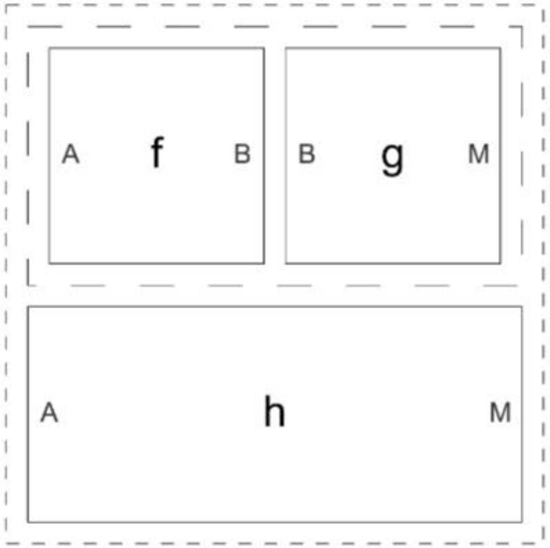
- Splitting into “layers” with one non-identity morphism each (“normal-ish” form)



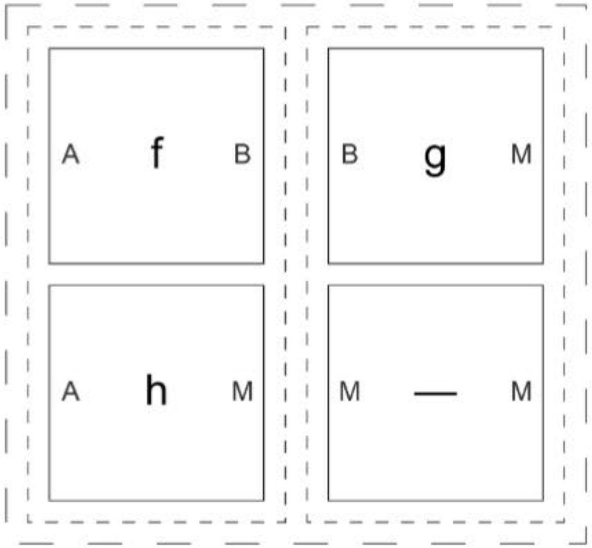


Foliation

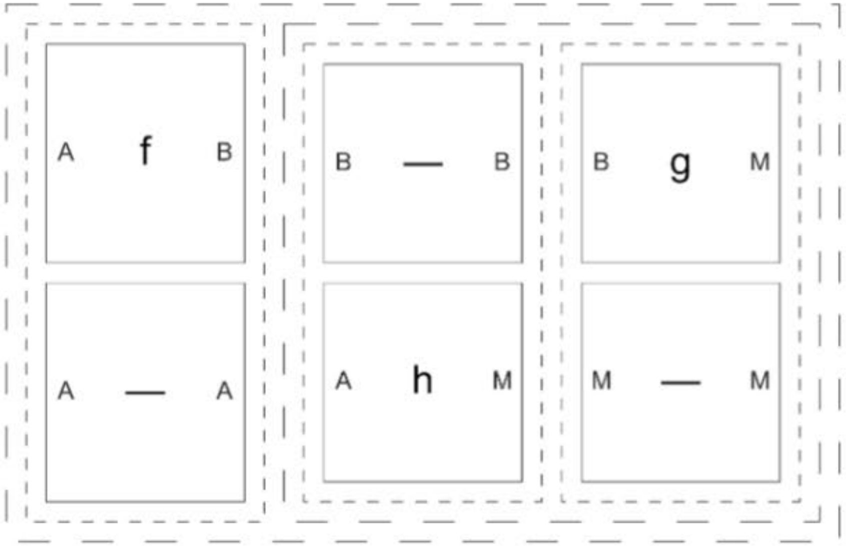
- Splitting into “layers” with one non-identity morphism each (“normal-ish” form)
- Weak foliation: stacks of non-identity morphisms without compositions
- `foliate, weak_foliate [_LHS | _RHS | _LRHS]`



(a) Initial state.



(b) Weak foliation.



(c) Foliation.

Diagrammatic Reasoning?

Diagrammatic Reasoning?

- Automatically handle coherence

Diagrammatic Reasoning?

- Automatically handle coherence
- For categories:
  - `cancel_isos` + `rassoc` solves categorical diagrammatic reasoning

Diagrammatic Reasoning?

- Automatically handle coherence
- For categories:
  - `cancel_isos` + `rassoc` solves categorical diagrammatic reasoning
- For monoidal categories… less clear
- Starting point is monoidal coherence
  - Only one structural morphism to reassociate and cancel unit

Monoidal Coherence Tactics

- Proved monoidal coherence (for types with UIP)
- To apply to diagrams with non-structural morphisms, construct intermediate inductive representation directly encoding structure
    - So, we can write functions on morphisms to manipulate structure and prove these functions correct

Monoidal Coherence Tactics

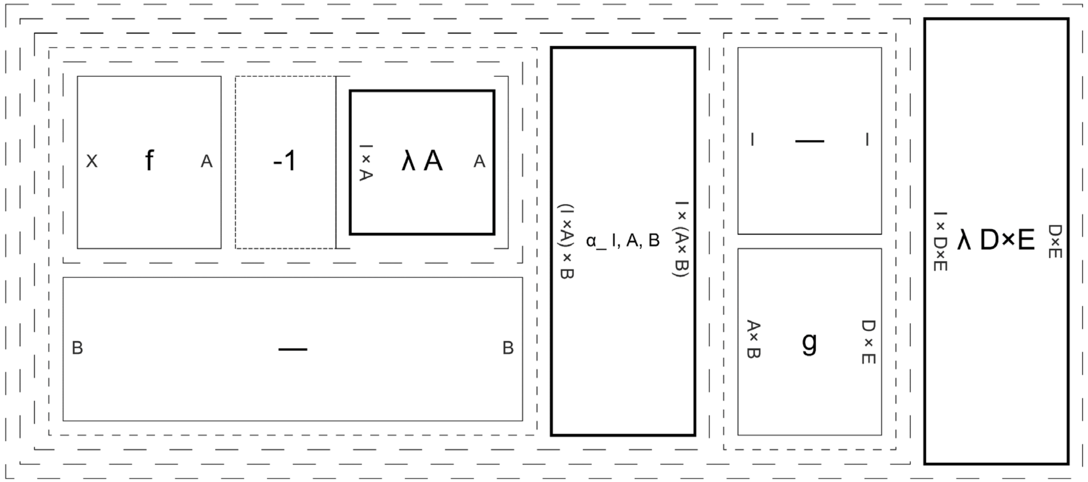
- `monoidal` tactic: shows morphisms equivalent by computing and comparing their representations in "string diagram form"

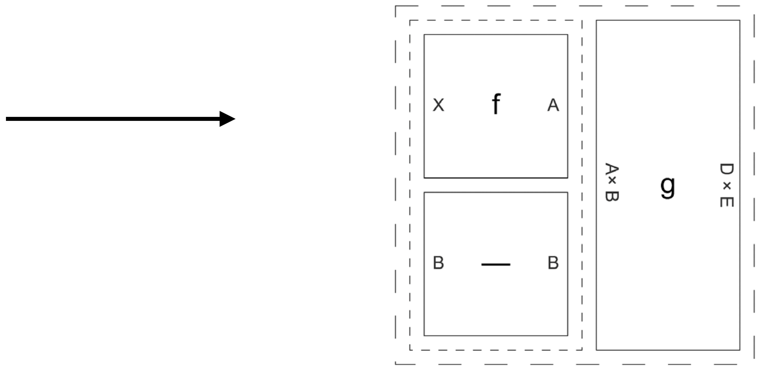
Monoidal Coherence Tactics

- `monoidal` tactic: shows morphisms equivalent by computing and comparing their representations in "string diagram form"

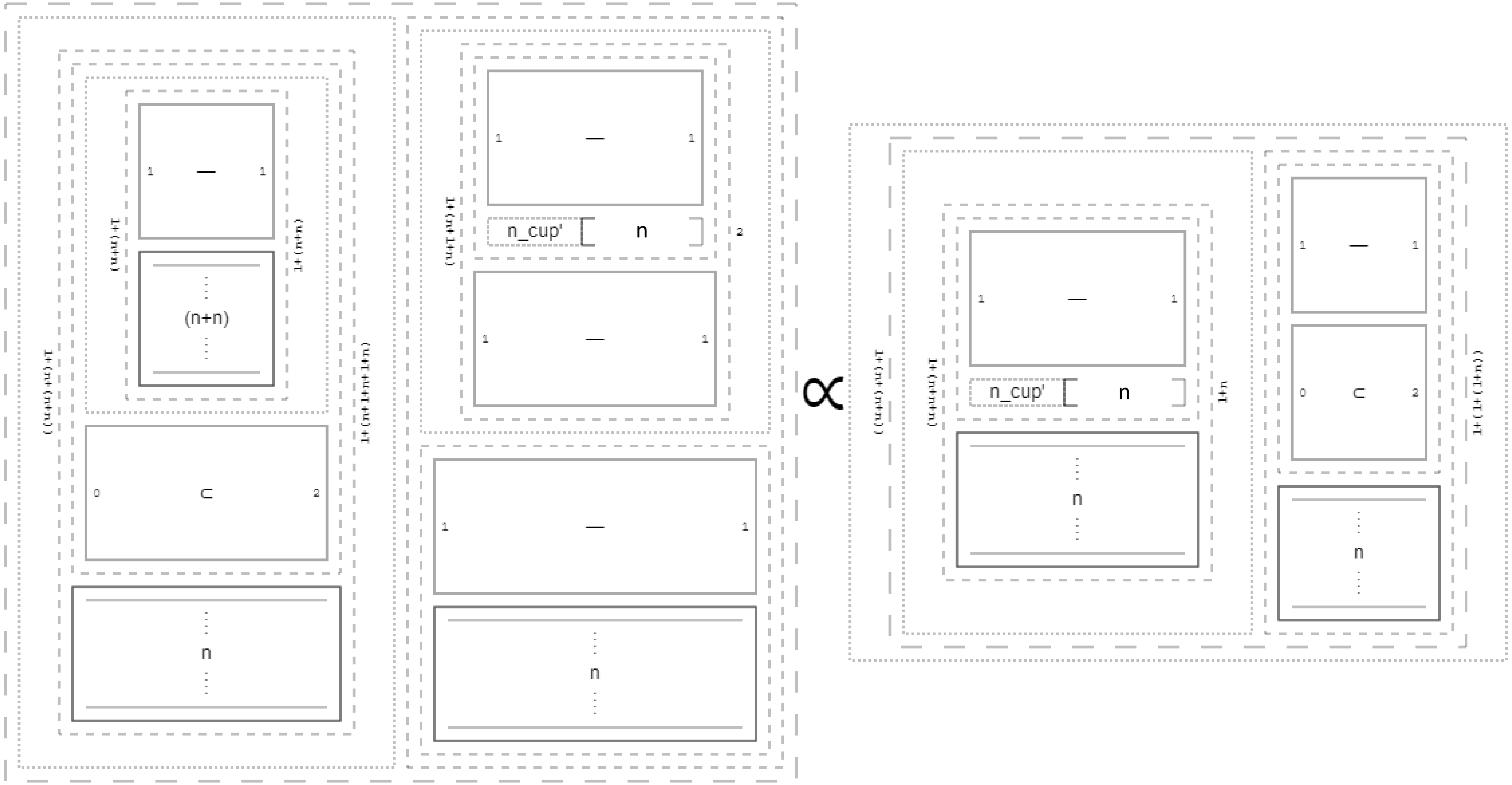
```
(f ∘ λ_ A ⁻¹) ⊗ id_ B ∘ α_ I, A, B ∘
      id_ I ⊗ g ∘ λ_ (D × E)
```

$$[[f\,,\mathrm{id}_B],[g]]$$





Why ViCAR?



Future Directions

- `monoidal_rw` : rewrite up to monoidal structure
  - Visualization hiding structural morphisms
- More category theory, more coherence (braided, symmetric, etc.)
- Support for cast-based developments (likely through automation)
- Interactive graphical proof for true diagrammatic reasoning