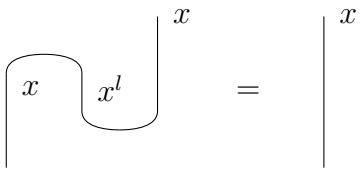
0.1 Adding extra structure

This section looks at the implementation of monoidal categories with extra structure: rigid, symmetric, hypergraph and cartesian closed categories.

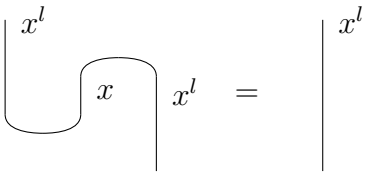
0.1.1 Rigid categories & wire bending

In sections **??** and **??** we discussed the fundamental notion of *adjunction* with the example of free-forgetful functors. The definition of left and right adjoints in terms of unit and counit natural transformations makes sense in **Cat**, but it can be translated in the context of any monoidal category C. An object $x^l \in C_0$ is the left adjoint of $x \in C_0$ whenever there are two arrows $\texttt{cup}(x) : x^l \otimes x \to 1$ and $\texttt{cap}(x) : 1 \to x \otimes x^l$ (also called counit and unit) such that:

- $\texttt{cap}(x) \otimes x \fatsemi x \otimes \texttt{cup}(x) = \texttt{id}(x)$,



- $x^l \otimes \texttt{cap}(x) \fatsemi \texttt{cup}(x) \otimes x^l = \texttt{id}(x^l)$.

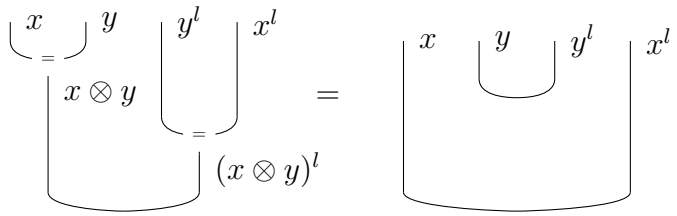


This is equivalent to the condition that the functor $x^l \otimes - : C \to C$ is the left adjoint of $x \otimes - : C \to C$. Symmetrically, $x^r \in C_0$ is the right-adjoint of $x \in C_0$ if x is its left adjoint. We say that C is *rigid* (also called *autonomous*) if every object has a left and right adjoint. From this definition we can deduce a number of properties:

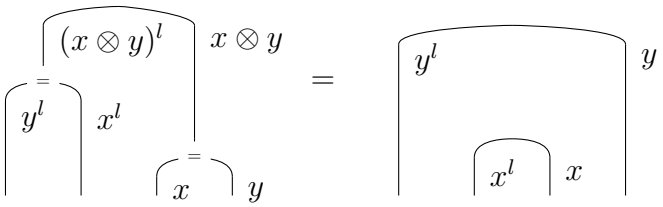
- adjoints are unique up to isomorphism,
- adjoints are monoid anti-homomorphisms, i.e. $(x \otimes y)^l \simeq y^l \otimes x^l$ and $1^l \simeq 1$,
- left and right adjoints cancel, i.e. $(x^l)^r \simeq x \simeq (x^r)^l$,

We say that C is strictly rigid whenever these isomorphisms are in fact equalities, again one can show that any rigid category is monoidally equivalent to a strict one. One can also show that cups and caps compose by nesting:

- $\texttt{cup}(x \otimes y) \;=\; y^l \otimes \texttt{cup}(x) \otimes y \;\fatsemi\; \texttt{cup}(y)$,



- $\texttt{cap}(x \otimes y) \;=\; \texttt{cap}(x) \;\fatsemi\; x \otimes \texttt{cap}(y) \otimes x^l$,

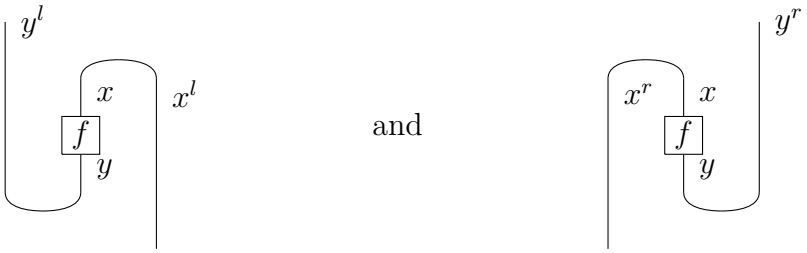


- $\texttt{cup}(1) \;=\; \texttt{cap}(1) \;=\; \texttt{id}(1)$, drawn as the equality of three empty diagrams.

The first two equations are drawn as diagrams in a non-foo monoidal category, i.e. with wires for composite types and explicit boxes for equality. This can be taken as an inductive definition, once we have defined the cups and caps for generating objects, we have defined them for all types. Thus, we can take the data for a (strictly) rigid category C to be that of a free-on-objects monoidal category together with:

- a pair of unary operators $(-)^l, (-)^r : C_0 \to C_0$ on generating objects,
- and a pair of functions $\texttt{cup}, \texttt{cap} : C_0 \to C_1$ witnessing that x^l and x^r are the left and right adjoints of each generating object $x \in C_0$.

Diagrams in rigid categories are more flexible than monoidal categories: we can bend wires. They owe their name to the fact that they are less flexible than *pivotal categories*. For any rigid category C, there are two contravariant endofunctors, called the left and right *transpose* respectively. They send objects to their left and right adjoints, and each arrow $f : x \to y$ to



A rigid category C is called *pivotal* when it has a monoidal natural isomorphism $x^l \sim x^r$ for each object x, which implies that the left and right transpose coincide: we can rotate diagrams by 360 degrees. We say C is strictly piotal when this isomorphism is an equality. This is the case for any rigid category C with a dagger structure: the dagger of the cup (cap) for an object x is the cap (cup) of its left-adjoint x^l. When this is the case, C is called †-pivotal. We say C is strictly piotal when left and right transpose are equal.

Example 0.1.1. *Recall from example* **??** *that for any category C, the category C^C of endofunctors and natural transformations is monoidal. Its subcategory with endofunctors that have both left and right adjoints is rigid. Its subcategory with endofunctors that have equal left and right adjoints is pivotal.*

Example 0.1.2. Tensor$_{\mathbb{S}}$ *is †-pivotal with left and right adjoints given by list reversal, cups and caps by the Kronecker delta* $\texttt{cup}(n)(i,j) = \texttt{cap}(n)(i,j) = 1$ *if* $i = j$ *else* 0. *Note that for tensors of order greater than 2, the* diagrammatic transpose *defined in this way differs from the usual* algebraic transpose*: the former reverses list order while the latter is the identity on objects.*

Example 0.1.3. Circ *is †-pivotal with the preparation of the Bell state as cap and the post-selected Bell measurement as cup (both are scaled by $\sqrt{2}$). The snake equations yield a proof of correctness for the (post-selected) quantum teleportation protocol.*

Example 0.1.4. *A rigid category which is also a preordered monoid (i.e. with at most one arrow between any two objects) is called a (quasi*[1]*)* pregroup*, their application to NLP will be discussed in section 1.1. A commutative pregroup is a (preordered) group: left and right adjoints coincide with the multiplicative inverse.*

Natural examples of non-free non-commutative pregroups are hard to come by. One exception is the monoid of monotone unbounded integer functions *with composition as multiplication and pointwise order. The left adjoint of $f : \mathbb{Z} \to \mathbb{Z}$ is defined such that $f^l(m)$ is the minimum $n \in \mathbb{Z}$ with $m \leq f(n)$ and symmetrically $f^r(m)$ is the maximum $n \in \mathbb{N}$ with $f(n) \leq m$.*

Any monoidal functor $F : C \to D$ between two rigid categories C and D preserves left and right adjoints up to isomorphism, we say it is strict when it preserves

[1] In his original definition [Lam99], Lambek also requires that pregroups are *partial orders*, i.e. preorders with antisymmetry $x \leq y$ and $y \leq x$ implies $x = y$. This implies that pregroups are strictly rigid, but also that they cannot be free on objects: $\texttt{cup}(x) \otimes \texttt{id}(x) : x \otimes x^l \otimes x \to x$ and $\texttt{id}(x) \otimes \texttt{cap}(x) : x \to x \otimes x^l \otimes x$ together would imply $x = x \otimes x^l \otimes x$.

them up to equality. Thus, we have defined a subcategory **RigidCat** $\hookrightarrow$ **MonCat**. We define a *rigid signature* Σ as a monoidal signature where the generating objects have the form $\Sigma_0 \times \mathbb{Z}$. We identify $x \in \Sigma_0$ with $(x, 0) \in \Sigma_0 \times \mathbb{Z}$ and define the left and right adjoints $(x, z)^l = (x, z - 1)$ and $(x, z)^r = (x, z + 1)$. The objects Σ_0 are called *basic types*, their iterated adjoints $\Sigma_0 \times \mathbb{Z}$ are called *simple types*. The integer $z \in \mathbb{Z}$ is called the *adjunction number* of the simple type $(x, z) \in \Sigma_0 \times \mathbb{Z}$ by Lambek and Preller [PL07] and its *winding number* by Joyal and Street [JS88]. Again, a morphism of rigid signatures $f : \Sigma \to \Gamma$ is a pair of functions $f : \Sigma_0 \to \Gamma_0$ and $f : \Sigma_1 \to \Gamma_1$ which commute with domain and codomain.

There is a forgetful functor $U :$ **RigidCat** $\to$ **RigidSig** which sends any strictly-rigid foo-monoidal category to its underlying rigid signature. We now describe its left-adjoint $F^r :$ **RigidSig** $\to$ **RigidCat**. Given a rigid signature Σ, we define a monoidal signature $\Sigma^r = \Sigma \cup \{\texttt{cup}(x)\}_{x \in \Sigma_0} \cup \{\texttt{cap}(x)\}_{x \in \Sigma_0}$. The free rigid category is the quotient $F^r(\Sigma) = F(\Sigma^r)/R$ of the free monoidal category by the snake equations R. That is, the objects are lists of simple types $(\Sigma_0 \times \mathbb{Z})^\star$, the arrows are equivalence classes of diagrams with cup and cap boxes. This is implemented in the `rigid` module of DisCoPy as outlined below.

Listing 0.1.5. Implementation of objects and types of free rigid categories.

```
@dataclass
class Ob(cat.Ob):
    z: int

    l = property(lambda self: Ob(self.name, self.z - 1))
    r = property(lambda self: Ob(self.name, self.z + 1))

    @classmethod
    def upgrade(cls, old: cat.Ob) -> Ob:
        return old if isinstance(old, cls) else cls(str(old), z=0)

class Ty(monoidal.Ty, Ob):
    def __init__(self, objects=[]):
        monoidal.Ty.__init__(self, objects=map(Ob.upgrade, objects))

    l = property(lambda self: self.upgrade(Ty(*[x.l for x in self.objects[::-1]])))
    r = property(lambda self: self.upgrade(Ty(*[x.r for x in self.objects[::-1]])))
```

Example 0.1.6. *We can check the axioms for objects in rigid categories hold on the nose.*

```
x, y = Ty('x'), Ty('y')
assert Ty().l == Ty() == Ty().r
assert (x @ y).l == y.l @ x.l and (x @ y).r == y.r @ x.r
assert x.r.l == x == x.l.r
```

`rigid.Ob` and `rigid.Ty` are implemented as subclasses of `cat.Ob` and `monoidal.Ty` respectively, with `property` methods (i.e. attributes that are computed on the fly) `l` and `r` for the left and right adjoints. Thanks to the `upgrade` method, we do not need to override the `tensor` method inherited from `monoidal.Ty`. In turn, subclasses of `rigid.Ty` will not need to override `l` and `r`. Similarly, the `rigid.Diagram` class is a subclass of `monoidal.Diagram`, thanks to the `upgrade` we do not need to reimplement the identity, composition or tensor. `rigid.Box` is a subclass of `monoidal.Box` and `rigid.Diagram`, with `Box.upgrade = Diagram.upgrade`. `Cup` (`Cap`) is a subclass of `Box` initialised by a pair of types `x, y` such that `len(x) == len(y) == 1` `x == y.l` (`x.l == y`, respectively). The class methods `cups` and `caps` construct diagrams of nested cups and caps by induction, with `Cup` and `Cap` as a base case.

Listing 0.1.7. Implementation of the arrows of free rigid categories.

```
class Diagram(monoidal.Diagram):
    def transpose(self, left=True) -> Diagram:
        if left: ... # Symmetric to the right case.
        return self.caps(self.dom.r, self.dom) @ self.id(self.cod.r)\
            >> self.id(self.dom.r) @ self @ self.id(self.cod.r)\
            >> self.id(self.dom.r) @ self.cups(self.cod, self.cod.r)

class Box(monoidal.Box, Diagram):
    upgrade = Diagram.upgrade

class Cup(Box):
    def __init__(self, x: Ty, y: Ty):
        assert len(x) == 1 and x == y.l
        super().__init__("Cup({}, {})".format(x, y), x @ y, Ty())

class Cap(Box):
    def __init__(self, x: Ty, y: Ty):
        assert len(x) == 1 and x.l == y
        super().__init__("Cap({}, {})".format(x, y), Ty(), x @ y)

def nesting(factory):
    @classmethod
    def method(cls, x: Ty, y: Ty) -> Diagram:
```

```
        if len(x) == 0: return cls.id(Ty())
        if len(x) == 1: return factory(x, y)
        head = factory(x[0], y[-1])
        if head.dom:  # We are nesting cups.
            return x[0] @ method(x[1:], y[:-1]) @ y[-1] >> head
        return head >> x[0] @ method(x[1:], y[:-1]) @ y[-1]

Diagram.cups, Diagram.caps = nesting(Cup), nesting(Cap)
```

The *snake removal* algorithm listed below computes the normal form of diagrams in rigid categories. It is a concrete implementation of the abstract algorithm described in pictures by Dunn and Vicary [DV19, p. 2.12]. First, we implement a subroutine `follow_wire` takes a codomain node (given by the index `i` of its box and the index `j` of itself in the box's codomain) follows the wire till it finds either the domain of another box or the codomain of the diagram. When we follow a wire, we compute two lists of *obstructions*, the index of each box on its left and right. The `find_snake` function calls `follow_wire` for each `Cap` in the diagram until it finds one that is connected to a `Cup`, or returns `None` otherwise. A `Yankable` snake is given by the index of its cup and cap, the two lists of obstructions on each side and whether it is a left or right snake. `unsnake` applies `interchange` repeatedly to remove the obstructions, i.e. to make the cup and cap consecutive boxes in the diagram, then returns the diagram with the snake removed. Each snake removed reduces the length n of the diagram by 2, hence the `snake_removal` algorithm makes at most $n/2$ calls to `find_snake`. Finally, we call `monoidal.Diagram.normal_form` which takes at most cubic time. Finding a snake takes quadratic time (for each cap we need to follow the wire at each layer) as well as removing it (for each obstruction we make a linear number of calls to `interchange`). Thus, we can compute normal forms for diagrams in free rigid categories in cubic time. We conjecture that we can in fact solve the word problem (i.e. deciding whether two diagrams are equal) in quadratic time using the same reduction to planar map isomorphism as in theorem **??**.

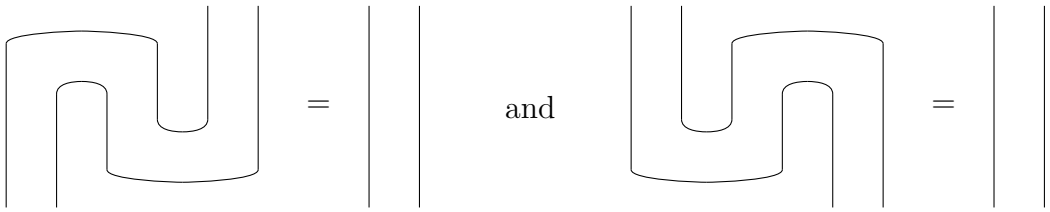
Listing 0.1.8. Outline of the snake removal algorithm.

```
Obstruction = tuple[list[int], list[int]]
Yankable = tuple[int, int, Obstruction, bool]

def follow_wire(self: Diagram, i: int, j: int) -> tuple[int, int, Obstruction]: ...
def find_snake(self: Diagram) -> Optional[Yankable]: ...
def unsnake(self: Diagram, yankable: Yankable) -> Diagram: ...
def snake_removal(self: Diagram) -> Diagram:
```

```
    yankable = find_snake(diagram)
    return snake_removal(unsnake(diagram, yankable)) if yankable else diagram

Diagram.normal_form = lambda self:\
    monoidal.Diagram.normal_form(snake_removal(self))
```

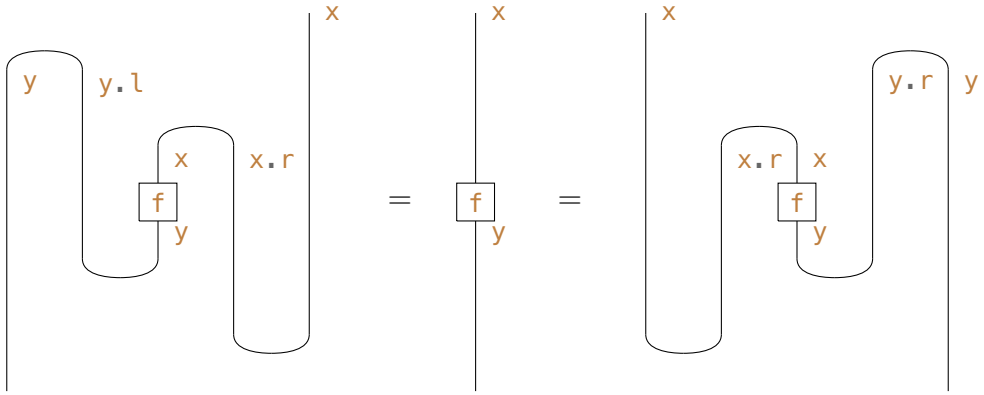
Example 0.1.9. *We can check that the snake equations hold up to normal form.*

```
t = x @ y

left_snake = Diagram.id(t.l).transpose(left=False)
right_snake = Diagram.id(t).transpose(left=True)

assert left_snake.normal_form() == Diagram.id(t)\
    and right_snake.normal_form() == Diagram.id(t.l)

drawing.equation(
    drawing.Equation(left_snake, Diagram.id(t)),
    drawing.Equation(right_snake, Diagram.id(t.l)),
    symbol='and', space=2, draw_type_labels=False)
```



Example 0.1.10. *We can check that left and right transpose cancel up to normal form.*

```
f = Box('f', x, y)

left_right_transpose = f.transpose(left=True).transpose(left=False)
right_left_transpose = f.transpose(left=False).transpose(left=True)

assert left_right_transpose.normal_form() == f == right_left_transpose.normal_form()
drawing.equation(left_right_transpose, f, right_left_transpose)
```



Listing 0.1.11. Implementation of **Circ** as a pivotal category.

```
class Qubits(monoidal.Qubits, Ty):
    l = r = property(lambda self: self)

class Circuit(monoidal.Circuit, Diagram):
    cups = nesting(lambda _, _: sqrt2 @ Ket(0, 0) >> H @ Id)
    caps = lambda x, y: Circuit.cups(x, y).dagger()
```

Example 0.1.12. *We can verify the teleportation protocol for two qubits.*

```
two_qubit_Bell_state = Circuit.caps(Qubits(2))
two_qubit_Bell_effect = Circuit.cups(Qubits(2))

assert (two_qubit_Bell_state @ Id @ Id >> Id @ Id @ two_qubit_Bell_effect).eval()\
    == (Id @ Id).eval()\
    == (Id @ Id @ two_qubit_Bell_state >> two_qubit_Bell_effect @ Id @ Id).eval()
```

`rigid.Functor` is implemented as a subclass of `monoidal.Functor` with the `__call__` method overriden. The image on types and on objects `x` with `x.z == 0` remains unchanged. The image on objects `x` with `x.z < 0` is defined by `F(x) = F(x.r).l` and symmetrically for `x.z > 0`. Indeed, when defining a strict rigid functor we only need to define the image of basic types, the image of their iterated adjoints is completely determined. The only problem arises when the objects in the codomain do not have `l` and `r` attributes, such as the implementation of $\mathbf{Tensor}_{\mathbb{S}}$ with `list[int]` as objects. In this case, we assume that the left and right adjoints are given by list reversal.

Listing 0.1.13. Implementation of strict rigid functors.

```
class Functor(monoidal.Functor):
    dom = cod = Category(Ty, Diagram)

    def __call__(self, other):
        if isinstance(other, Ty) or isinstance(other, Ob) and other.z == 0:
            return super().__call__(other)
        if isinstance(other, Ob):
            if not hasattr(self.cod.ob, 'l' if other.z < 0 else 'r'):
                return self(Ob(other.name, z=0))[::-1]
            return self(other.r).l if other.z < 0 else self(other.l).r
        if isinstance(other, Cup):
            return self.cod.ar.cups(self(other.dom[:1]), self(other.dom[1:]))
        if isinstance(other, Cap):
            return self.cod.ar.caps(self(other.dom[:1]), self(other.dom[1:]))
        return super().__call__(other)
```

Listing 0.1.14. Implementation of $\mathbf{Tensor}_{\mathbb{S}}$ as a pivotal category.

```
Tensor.cups = classmethod(lambda cls, x, y: cls(cls.id(x).inside, x + y, []))
Tensor.caps = classmethod(lambda cls, x, y: cls(cls.id(x).inside, [], x + y))
```

Example 0.1.15. *We can check that* $\mathbf{Tensor}_{\mathbb{S}}$ *is indeed pivotal.*

```
F = Functor(
    ob={x: 2, y: 3}, ar={f: [[1, 2, 3], [4, 5, 6]]}
    cod=Category(list[int], Tensor[int]))

assert F(left_snake) == F(Diagram.id(x)) == F(right_snake)
assert F(f.transpose(left=True)) == F(f).transpose() == F(f.transpose(left=False))

# Diagrammatic and algebraic transpose differ for tensors of order >= 2.
assert F(f @ f).transpose() != F((f @ f).transpose())
```

Free pivotal categories are defined in a similar way to free rigid categories, with the two-element field $\mathbb{Z}/2\mathbb{Z}$ instead of the integers $\mathbb{Z}$, i.e. simple types with adjunction numbers of the same parity are equal. In this case, we usually write $x^l = x^r = x^\star$ with $(x^\star)^\star = x$. Given a pivotal signature Σ with objects of the form $\Sigma_0 \times (\mathbb{Z}/2\mathbb{Z})$, the free pivotal category is the quotient $F^p(\Sigma) = F^r(\Sigma)/R$ of the free rigid category by the relation R equating the left and right transpose of the identity for each generating object. While the diagrams of free rigid categories can have snakes, those of free pivotal categories can have circles: we can compose $\mathsf{cap}(x) : 1 \to x^l \otimes x$ then $\mathsf{cup}(x^\star) : x^l \otimes x \to 1$ to form a scalar diagram called the *dimension* of the system x. We also draw the wires with an orientation: the wire for x is labeled with an arrow going down, the one for $x^\star$ with an arrow going up.

To the best of our knowledge, the word problem for pivotal categories is still open. When defining the normal form of pivotal diagrams, we would need to make a choice between the diagrams for left or right transpose of a box. Another solution is to add a new box $f^T : y^\star \to x^\star$ for the transpose of every box $f : x \to y$ in the signature, and set it as the normal form of both diagrams. We can add some asymmetry to the drawing of the box f, and draw f^T as its 180° degree rotation. If the category is also †-pivotal, we get a four-fold symmetry: the box, its dagger, its transpose and its dagger-transpose (also called its conjugate). This is still being developed by the DisCoPy community.

Listing 0.1.16. Implementation of free †-pivotal categories.

```
class Ob(rigid.Ob):
    l = r = property(lambda self: self.upgrade(Ob(self.name, (self.z + 1) % 2)))
```

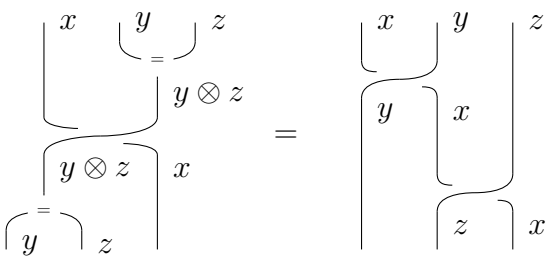
```
class Ty(rigid.Ty, Ob):
    def __init__(self, objects=[]):
        rigid.Ty.__init__(self, objects=map(Ob.upgrade, objects))

class Diagram(rigid.Diagram): pass

class Box(rigid.Box, Diagram):
    upgrade = Diagram.upgrade

class Cup(rigid.Cup, Box):
    def dagger(self):
        return Cap(self.dom[0], self.dom[1])

class Cap(rigid.Cap, Box):
    def dagger(self):
        return Cup(self.cod[0], self.cod[1])

Diagram.cups, Diagram.caps = nesting(Cup), nesting(Cap)

class Functor(rigid.Functor):
    dom = cod = Category(Ty, Diagram)
```

0.1.2 Braided categories & wire crossing

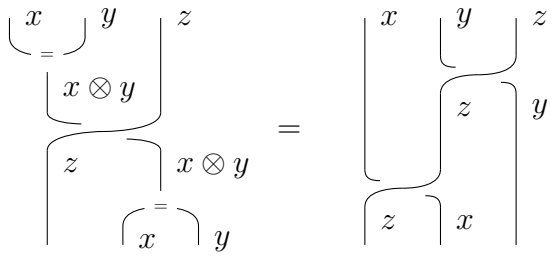
With rigid and pivotal categories, we have removed the assumption that diagrams are progressive: we can bend wires. With braided and symmetric monoidal categories, we now remove the planarity assumption: wires can cross.

The data for a *braided category* is that of a monoidal category C together with a *brading* natural isomorphism $B(x, y) : x \otimes y \to y \otimes x$ (and its inverse B^{-1}) drawn as a wire for x crossing under (over) a wire y. Braidings are subject to the following *hexagon equations*:

- $B(x, y \otimes z) = B(x, y) \otimes z \fatsemi y \otimes B(x, z)$,



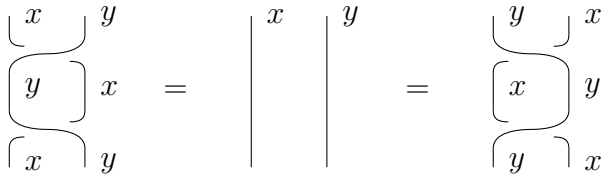
- $B(x \otimes y, z) = x \otimes B(y, z) \fatsemi B(x, z) \otimes y$



which owe their name to the shape of the corresponding commutative diagrams when C is non-strict monoidal. We also require that $B(x, 1) = \mathtt{id}(x) = B(1, x)$[1], i.e. braiding a wire x with the unit 1 does nothing, we do not need to draw it. The hexagon equations may be taken as an inductive definition: we can decompose the braiding $B(x, y \otimes z)$ of an object with a tensor in terms of two simpler braids $B(x, y)$ and $B(x, z)$. Thus, we can take the data for a braided category to be that of a foo-monoidal category together with a pair of functions $B, B^{-1} : C_0 \times C_0 \to C_1$ which send a pair of generating objects to their braiding and its inverse. Once we have specified the braids of generating objects, the braids of any type (i.e. list of objects) is uniquely determined. A monoidal functor $F : C \to D$ between two braided categories C and D is braided when $F(B(x, y)) = B(F(x), F(y))$. Thus, we get a category **BraidCat** with a forgetful functor $U : \mathbf{BraidCat} \to \mathbf{MonSig}$, we now describe its left adjoint.

Given a monoidal signature Σ, the free braided category is a quotient $F^B(\Sigma) = F(\Sigma^B)/R$ of the free monoidal category generated by $\Sigma^B = \Sigma \cup B \cup B^{-1}$ for the braiding $B(x, y) : x \otimes y \to y \otimes x$ and its inverse $B^{-1}(x, y) : y \otimes x \to x \otimes y$ for each pair of generating objects $x, y \in \Sigma_0$. The relation R is given by the following axioms for a natural isomorphism:

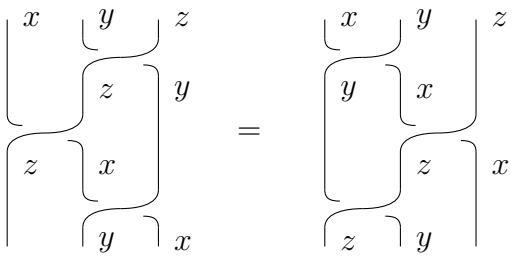
- $B(x, y) \fatsemi B^{-1}(x, y) \;=\; \mathtt{id}(x \otimes y) \;=\; B^{-1}(x, y) \fatsemi B(x, y)$,



- $f \otimes x \fatsemi B(b, x) \;=\; B(a, x) \fatsemi x \otimes f$ and $x \otimes f \fatsemi B(x, b) \;=\; B(x, a) \fatsemi f \otimes x$.

[1]Note that in a non-strict monoidal category this axiom is unnecessary, it follows from the coherence conditions.

for all generating objects $x, y \in \Sigma_0$ and boxes (including braidings) $f : a \to b$ in Σ^B. From B being an isomorphism on generating objects, we can prove it is self-inverse on any type by induction. Similarly, from B being natural on the left and right for each box, we can prove by induction that it is in fact natural for any diagram. Note that the naturality axiom holds for boxes with domains and codomains of arbitrary length. In particular, it holds for $f = B(y, z)$ in which case we get the following Yang-Baxter equation:



It also holds for any scalar $f : 1 \to 1$, which allows to pass them through a wire:

A braided category C is *symmetric* if the braiding B is its own inverse $B = B^{-1} = S$, in this case it is called a *swap* and drawn as the intersection of two wires. A symmetric functor is a braided functor between symmetric categories. A †-braided category is a braided category with a dagger structure, such that the braidings are unitaries, i.e. their inverse is also their dagger. A †-symmetric category is a †-braided category that is also symmetric.

Remark 0.1.17. *A symmetric (braided) category with one generating object is called a PROP (PROB) for PROduct and Permutation (Braid). Indeed, the arrows of the free PROP with no generating boxes (i.e. only swaps) are* permutations*, the arrows of the free braided PRO with no boxes are called* braids*. Both are* groupoids*, i.e. all their arrows are isomorphisms, which also implies that they are †-braided with the dagger given by the inverse. For every $n \in \mathbb{N}$, the arrows $f : x^n \to x^n$ in the free PROP (PROB) are the elements of the n-th symmetric group S_n (braid group B_n).*

DisCoPy implements free †-symmetric (†-braided) categories with a class `Swap` (`Braid`) initialised by types of length one and a class method `swap` (`braid`) for types of arbitrary length. The method `simplify` cancels every braid followed by its inverse. The `naturality` method applies the naturality axiom to the box at a given index `i: int`. The optional argument `left: bool` allows to choose between left and

right naturality axioms, `down: bool` allows to move the box either up or down the braid and `braid: Callable` allows to apply naturality to any subclass of `Braid`.

Listing 0.1.18. Implementation of free †-braided and †-symmetric categories.

```
class Diagram(monoidal.Diagram):
    def simplify(self):
        for i, (x, f, _), (y, g, _) in enumerate(zip(self.layers, self.layers[1:])):
            if x == y and isinstance(f, Braid) and f == g[::-1]:
                layers = self.layers[:i] + self.layers[i + 2:]
                return simplify(self.upgrade(Diagram(self.dom, self.cod, layers)))
        return self

class Box(monoidal.Box, Diagram):
    upgrade = Diagram.upgrade

class Braid(Box):
    def __init__(self, x: Ty, y: Ty, is_dagger=False):
        assert len(x) == len(y) == 1
        name = "Braid({}, {})[::-1]".format(y, x)\
            if is_dagger else "Braid({}, {})".format(y, x)
        super().__init__(name, x @ y, y @ x, is_dagger)

    def dagger(self): return Braid(*self.cod, is_dagger=not self.is_dagger)

class Swap(Braid):
    def __init__(self, x: Ty, y: Ty):
        super().__init__(x, y); self.name = self.name.replace("Braid", "Swap")

    def dagger(self): return Swap(*self.cod)

def hexagon(factory) -> Callable:
    @classmethod
    def method(cls, x: Ty, y: Ty) -> Diagram:
        if len(x) == 0: return cls.id(y)
        if len(x) == 1:
            if len(y) == 1: return factory(x[0], y[0])
            return method(cls, x, y[:1]) @ cls.id(y[1:])\
                >> cls.id(y[:1]) @ method(cls, x, y[1:])  # left hexagon equation.
        return cls.id(x[:1]) @ method(cls, x[1:], y)\
            >> method(cls, x[:1], y) @ cls.id(x[1:])  # right hexagon equation.
    return method

Diagram.braid, Diagram.swap = hexagon(Braid), hexagon(Swap)
```

```
def naturality(self: Diagram, i: int, left=True, down=True, braid=None):
    braid = braid or self.braid
    layer, box = self.layers[i], self.layers[i].box
    def pattern(left, down):
        if left and down: return layer.left[-1] @ box >> braid(layer.left[-1], box.cod)
        if down: return box @ layer.right[0] >> braid(box.cod, layer.right[0])
        if left: return braid(box.dom, layer.right[0]) >> layer.right[0] @ box
        return braid(layer.left[-1], box.dom) >> box @ layer.left[-1]
    source, target = pattern(left, down), pattern(not left, not down)
    match = Match(top=self[:i] if down else self[:i - len(source) + 1],
                  bottom=self[i + len(source):] if down else self[i + 1:],
                  left=layer.left[-1] if left == down else layer.left,
                  right=layer.right if left == down else layer.right[1:])
    assert self == match.subs(source)
    return match.subs(target)

Diagram.naturality = naturality

class Functor(monoidal.Functor):
    def __call__(self, other):
        if isinstance(other, Swap):
            return self.cod.ar.swap(self(other[0]), self(other[1]))
        if isinstance(other, Braid) and not other.is_dagger:
            return self.cod.ar.braid(self(other[0]), self(other[1]))
        return super().__call__(other)
```

Example 0.1.19. *We can check the hexagon equations hold on the nose.*

```
x, y, z = map(Ty, "xyz")

assert Diagram.braid(x, y @ z) == Braid(x, y) @ z >> y @ Braid(x, z)
assert Diagram.braid(x @ y, z) == x @ Braid(y, z) >> Braid(x, z) @ y
```

We can check that `Braid` *is an isomorphism up to a* `simplify` *call.*

```
assert (Diagram.braid(x, y @ z) >> Diagram.braid(x, y @ z)[::-1]).simplify()\
    == Diagram.id(x @ y @ z)\
    == (Diagram.braid(x, y @ z)[::-1] >> Diagram.braid(x, y @ z)).simplify()
```

We can check that `Braid`*, its inverse and* `Swap` *are all natural.*

```
a, b = Ty('a'), Ty('b')
f = Box('f', a, b)

for braid in [Diagram.braid, (lambda x, y: Diagram.braid(y, x)[::-1]), Diagram.Swap]:
```

```
    source, target = x @ f >> braid(x, b), braid(x, a) >> f @ x
    assert source.naturality(0, braid=braid) == target
    assert target.naturality(1, left=False, down=False, braid=braid) == source
```

Listing 0.1.20. Implementation of **Pyth** and $\mathbf{Tensor}_{\mathbb{S}}$ as symmetric categories.

```
def function_swap(x: list[type], y: list[type]) -> Function:
    def inside(*xs):
        assert len(xs) == len(x + y)
        return untuple(xs[len(x):] + xs[:len(x)])
    return Function(inside, dom=x + y, cod=y + x)

Function.swap = Function.braid = function_swap

@classmethod
def tensor_swap(cls, x: list[int], y: list[int]) -> Tensor:
    inside = [[(i0, j0) == (i1, j1)
        for j0 in range(product(y)) for i0 in range(product(x))]
        for i1 in range(product(x)) for j1 in range(product(y))]
    return cls(inside, dom=x + y, cod=y + x)

Tensor.swap = Tensor.braid = tensor_swaps
```

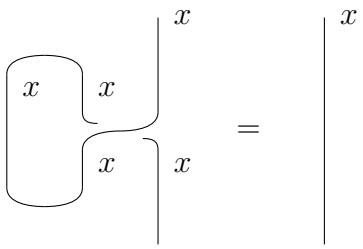
Example 0.1.21. *We can check the axioms for symmetric categories hold in* $\mathbf{Tensor}_{\mathbb{S}}$ *and* **Pyth**.

```
swap_twice = Diagram.swap(x, y @ z) >> Diagram.swap(y @ z, x)

F = Functor(
    ob={a: 2, b: 3, x: 4, y: 5, z: 6},
    ar={f: [[1-2j, 3+4j]]},
    cod=Category(list[int], Tensor[complex]))

assert F(f @ x >> Swap(b, x)) == F(Swap(a, x) >> x @ f)
assert F(x @ f >> Swap(x, b)) == F(Swap(x, a) >> f @ x)
assert F(swap_twice) == Tensor.id(F(x @ y @ z))

G = Functor(
    ob={a: complex, b: real, x: int, y: bool, z: str},
    ar={f: lambda z: abs(z) ** 2},
    cod=Category(list[type], Function))

assert G(f @ x >> Swap(b, x))(1j, 2) == G(Swap(a, x) >> x @ f)(1j, 2) == (2, f(1j))
assert G(x @ f >> Swap(x, b))(2, 1j) == G(Swap(x, a) >> f @ x)(2, 1j) == (f(1j), 2)
assert G(swap_twice)(42, True, "meaning of life") == (42, True, "meaning of life")
```

Remark 0.1.22. *Note that the naturality axioms in* **Pyth** *hold only for its subcategory of pure functions, as we will see in section 0.2* **Pyth** *is in fact a symmetric* premonoidal *category. This is also the case for* $\mathbf{Tensor}_{\mathbb{S}}$ *when the rig* $\mathbb{S}$ *is non-commutative.*

A *compact closed category* is one that is both rigid and symmetric, which implies that it is also pivotal, a †-compact closed category is both †-pivotal and †-symmetric. The arrows of free †-compact closed categories (i.e. equivalence classes of diagrams with cups, caps and swaps) are also called *tensor networks*, a graphical equivalent to *Einstein notation* and *abstract index notation*, first introduced by Penrose [Pen71]. Unlike the computer scientists however, physicists tend to identify the diagram (syntax) with its image under some interpretation functor to the category of tensors (semantics).

A *tortile category*, also called a *ribbon category*[1], is a braided, pivotal category which furthermore satisfies the following *untwisting* equation:



The scalars of the free tortile category with no boxes (i.e. equivalence classes of diagrams with only cups, caps and braids) are called *links* in general and *knots* when they are connected. Untwisting, the self-inverse equation and the Yang-Baxter equation (i.e. naturality with respect to braids) are called the three *Reidemeister moves*, they completely characterise the continuous deformations of circles embedded in three-dimensional space [Rei13].

The *unknotting problem* (given a knot, can it be untied, i.e. continuously deformed to a circle?) is a candidate NP-intermediate problem: it is decidable [Hak61] and in NP [Lac15], but there is neither a proof of it being NP-complete nor a polynomial-time algorithm. Delpeuch and Vicary [DV21] proved that the word problem for free braided categories is unknotting-hard. Hence, there is little hope of finding a simple polynomial-time algorithm for computing normal forms of braided diagrams. It is not known whether it is even decidable.

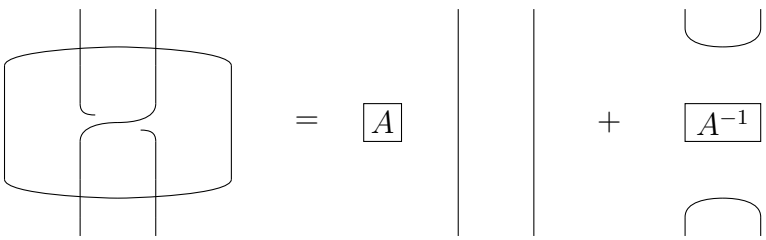
[1]Here again we take a *strict* definition, where the twist is an identity rather than an isomorphism. In a non-strict tortile category, the wires would be drawn as ribbons, i.e. two wires side by side. The twist isomorphism would be drawn as the two wires being braided twice. In a strict tortile category, the ribbon has no width thus the twist is invisible.

The word problem for free symmetric categories reduces to the *graph isomorphism problem* [PSV21], another potential NP-intermediate problem. The word problem for free compact closed categories also reduces to graph isomorphism [Sel07]. To the best of our knowledge, it is not known whether they are graph-isomorphism-hard, i.e. whether there is a reduction the other around that sends any graph to a diagram with swaps (and cups and caps) so that graphs are isomorphic if and only their diagrams are equal. Thus, there could be a simple polynomial-time algorithm for computing normal forms of diagrams in symmetric and compact closed categories. In any case, DisCoPy does not implement any normal forms for diagrams with braids yet.

Of course, we can also enrich rigid and braided categories in commutative monoids, i.e. we can take formal sums of diagrams with cups, caps and braids in the same way as any other box. We can also define bubbles and draw them in the same way as for monoidal diagrams.

Example 0.1.23. *We can define knot polynomials such as the Kauffman bracket using pivotal functors into a category where braids are defined as a weighted sum of diagrams.*

```
x = pivotal.Ty('$x$')
A, A.inverse = Box('$A$', Ty(), Ty()), Box('$A^{-1}$', Ty(), Ty())

class Polynomial(pivotal.Diagram):
    def braid(x, y):
        assert x == y and len(x) == len(y) == 1
        return (A @ x @ y) + (Cup(x, y) >> A.inverse >> Cap(x, y))

Kauffman = Functor(
    ob={x: x}, ar={}, cod=Category(pivotal.Ty, Polynomial))

drawing.equation(Braid(x, x).bubble(), Kauffman(Braid(x, x)))
```



0.1.3 Hypergraph categories & wire splitting

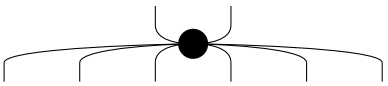
With compact closed and tortile categories, we have removed both the progressivity and the planarity assumptions: wires can bend and cross. With *hypergraph*

categories we remove the assumption that diagrams are graphs: wires can split and merge, they need not be homeomorphic to an open interval. A hypergraph category is a symmetric category with *coherent special commutative spiders*, let's spell out what this means.

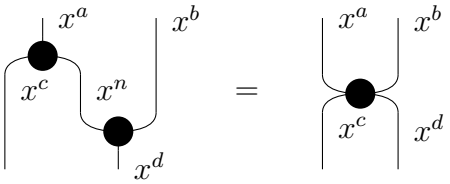
An object x in a monoidal category C has *spiders* with *phases* in a monoid $(\Phi, +, 0)$ if it comes equipped with a family of arrows $\texttt{spider}_{\phi,a,b}(x) : x^a \to x^b$ for every phase $\phi \in \Phi$ and pair of natural numbers $a, b \in \mathbb{N}$, such that the following *spider fusion* equation holds for all $a, b, c, d, n \in \mathbb{N}$.

$$\texttt{spider}_{\phi,a,c+n}(x) \otimes x^b \fatsemi x^c \otimes \texttt{spider}_{\phi',n+c,d}(x) \quad = \quad \texttt{spider}_{\phi+\phi',a+b,c+d}(x)$$

We also require that our spiders satisfy the *special* condition $\texttt{spider}_{0,1,1}(x) = \texttt{id}(x)$. Spiders owe their name to their arachnomorphic drawing, for example $\texttt{spider}_{\phi,2,6}$ is drawn as a node (the head) and its wires (the eight legs of the spider, two of them menacing us):



Once drawn, the spider fusion equation has the intuitive graphical meaning that if two spiders touch, they fuse and add up their phase.



From spider fusion, we can deduce the following properties:

- $\texttt{merge}(x) = \texttt{spider}_{0,2,1}(x)$ and $\texttt{unit}(x) = \texttt{spider}_{0,0,1}(x)$ form a commutative monoid,
- $\texttt{split}(x) = \texttt{spider}_{0,1,2}(x)$ and $\texttt{counit}(x) = \texttt{spider}_{0,1,0}(x)$ form a cocommutative comonoid,
- $\texttt{split}(x) \fatsemi \texttt{merge}(x) = \texttt{id}(x)$, called the *special* condition,
- $\texttt{merge}(x) \otimes x \fatsemi x \otimes \texttt{split}(x) \quad = \quad x \otimes \texttt{merge}(x) \fatsemi \texttt{split}(x) \otimes x$, called the *Frobenius law*.

In fact, when the phases are trivial $\Phi = \{0\}$ these four axioms are sufficient to deduce spider fusion, spiders are also called *special Frobenius algebras*. Indeed,

given a commutative monoid $\texttt{merge}(x) : x \otimes x \to x$, $\texttt{unit}(x) : 1 \to x$ and a cocommutative comonoid $\texttt{split}(x) : x \to x \otimes x$, $\texttt{counit}(x) : x \to 1$ subject to the Frobenius law, we can construct $\texttt{spider}_{a,b}(x) : x^a \to x^b$ by induction on the number of legs. The base case is given by the special condition $\texttt{spider}_{1,1}(x) = \texttt{id}(x)$. Then we define spiders with $a \in \mathbb{N}$ input legs for $a \neq 1$:

- $\texttt{spider}_{0,b}(x) \;=\; \texttt{unit}(x) \fatsemi \texttt{spider}_{1,b}(x)$,
- $\texttt{spider}_{a+2,b}(x) \;=\; \texttt{merge}(x) \otimes x^a \;\fatsemi\; \texttt{spider}_{a+1,b}(x)$,

Finally we define spiders with one input leg by induction on the output legs $b \in \mathbb{N}$:

- $\texttt{spider}_{1,0}(x) \;=\; \texttt{counit}(x)$,
- $\texttt{spider}_{1,b+2}(x) \;=\; \texttt{spider}_{1,b+1}(x) \;\fatsemi\; \texttt{split}(x) \otimes x^b$.

One can show that this satisfies the spider fusion law, again by induction on the legs [HV19, Lemma 5.20]. In this way, we can construct an infinite family of spiders from just the four boxes $\texttt{merge}(x), \texttt{unit}(x), \texttt{split}(x), \texttt{counit}(x)$ and a finite set of equations. A spider is nothing more than a big product followed by a big coproduct. As for the phases, we can recover them from a family of *phase shifts* $\{\texttt{shift}_\phi(x) : x \to x\}_{\phi \in \Phi}$ such that:

- $\texttt{shift}_{\_}(x)$ is a monoid homomorphism $\Phi \to C(x,x)$, i.e. $\texttt{shift}_0(x) = \texttt{id}(x)$ and $\texttt{shift}_\phi(x) \fatsemi \texttt{shift}_{\phi'} = \texttt{shift}_{\phi+\phi'}(x)$,
- $\texttt{shift}_\phi(x) \otimes x \;\fatsemi\; \texttt{merge}(x) \;=\; \texttt{merge}(x) \fatsemi \texttt{shift}_\phi(x) \;=\; x \otimes \texttt{shift}_\phi(x) \;\fatsemi\; \texttt{merge}(x)$,
- $\texttt{split}(x) \;\fatsemi\; \texttt{shift}_\phi(x) \otimes x \;=\; \texttt{shift}_\phi(x) \fatsemi \texttt{split}(x) \;=\; x \otimes \texttt{split}(x) \;\fatsemi\; \texttt{shift}_\phi(x)$.

We can then define $\texttt{spider}_{\phi,a,b}(x) = \texttt{spider}_{a,1}(x) \fatsemi \texttt{shift}_\phi(x) \fatsemi \texttt{spider}_{1,b}(x)$ and check that indeed, spiders fuse up to addition of their phase. Thus when the monoid is finite, we get a finite number of boxes and equations, i.e. a finite presentation of the spiders. In fact instead of taking it as data, we could have equivalently defined the monoid of phases Φ as the set of endomorphisms $x \to x$ that satisfy the last two conditions.

Remark 0.1.24. *Given any (non-special) Frobenius algebra on an object x, we can show that x is its own left and right adjoint. Indeed, take* $\texttt{cup}(x) = \texttt{unit}(x) \fatsemi \texttt{split}(x)$ *and* $\texttt{cap}(x) = \texttt{merge}(x) \fatsemi \texttt{counit}(x)$*, then the Frobenius law and the (co)unit law of the (co)monoid implies the snake equations. Thus, a category with (not-necessarily special) spiders on every object is automatically a pivotal category.*

Example 0.1.25. *In any pivotal category, there is a (non-special) Frobenius algebra for every object of the form $x^\star \otimes x$ given by:*

- $\texttt{merge}(x^\star \otimes x) = x^\star \otimes \texttt{cup}(x^\star) \otimes x$ *and* $\texttt{unit}(x) = \texttt{cup}(x)$,
- $\texttt{split}(x^\star \otimes x) = x^\star \otimes \texttt{cap}(x) \otimes x$ *and* $\texttt{counit}(x) = \texttt{cap}(x^\star)$.

Due to the drawing of its comonoid, it is also called the pair of pants *algebra. The special condition requires the dimension of the system x to be the unit, i.e. the circle is equal to the empty diagram. Non-special Frobenius algebras can still be drawn as spiders, they satisfy a modifed version of spider fusion where we keep track of the number of circles, i.e. the number of splits followed by a merge. We can extend our inductive definition so that all the circles are in between the product and coproduct, see [HV19, Theorem 5.21].*

Example 0.1.26. *The category* $\mathbf{Tensor}_{\mathbb{S}}$ *has spiders for every dimension $n \in \mathbb{N}$ with phases in any submonoid of $\phi \in (\mathbb{S}, \times, 1)^n$. They are given by $\texttt{spider}_{\phi,a,b}(n) = \sum_{i \leq n} \phi_i |i\rangle^{\otimes a} \langle i|^{\otimes b}$ where $|i\rangle$ ($\langle i|$) is the i-th basis row (column) vector.*

```
class Tensor:
    ...
    @classmethod
    def spider(cls, a: int, b: int, n: int, phase=None) -> Tensor:
        phase = phase or n * [1]
        inside = [[sum(phase)]] if not a and not b\
            else [[phase[xs[0]] for xs in itertools.product(*b * [range(n)])
                   if all(x == xs[0] for x in xs)]]\
            if not a else cls.spider([], a + b, n).inside
        return cls(inside, dom=a * [n], cod=b * [n])
```

When $\mathbb{S}$ is a field, we can divide every ϕ_i by ϕ_0, or equivalently require that $\phi_0 = 1$. Indeed, we can represent any spider with $\phi_0 \neq 1$ as a spider with $\phi_0 = 1$ multiplied by the scalar ϕ_0, which is called a global phase. *When $\mathbb{S} = \mathbb{C}$ and $n = 2$, we usually take the monoid of phases to be the unit circle and write it in terms of addition of angles.*

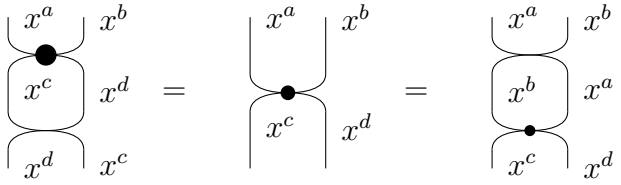
Example 0.1.27. *In the category* **Circ** *of quantum circuits, if we allow post-selected measurements then we can construct spiders with the unit circle as phases. The spiders with no inputs legs are called the (generalised) GHZ states:*

$$\texttt{spider}_{\alpha,0,b} = |0\rangle^{\otimes b} + e^{i\alpha}|1\rangle^{\otimes b}$$

Note that we need to scale by $\frac{1}{\sqrt{2}}$ *to make this a normalised quantum state. The spiders with* $a > 0$ *input legs can be thought of as measuring a qubits, post-selecting on all of them giving the same result and then preparing* b *copies of this result. The evaluation functor* $\mathbf{Circ} \to \mathbf{Tensor}_{\mathbb{C}}$ *sends spiders to spiders.*

Spiders allow us to draw diagrams where wires can split and merge, connecting an arbitrary number of boxes. The PRO of Frobenius algebras (without the special condition), i.e. diagrams with only spider boxes, defines a notion of "well-behaved" 1d subspaces of the plane, up to continuous deformation. Indeed, it is equivalent to the category of *planar thick tangles* [Lau05]. Intuitively, planar thick tangles can be thought of as planar wires with a width, i.e. that we can draw with pens or pixels. The inductive definition of spiders in terms of monoids and comonoids has the topological interpretation that any wire can be deformed so that all its singular points (i.e. where the wire crosses itself) are binary splits and merges. The special condition has the non-topological consequence that we can contract the holes in the wires, splitting a wire then merging it back does nothing. If the monoidal category C is braided, we can remove the planarity assumption and define *commutative spiders* as those where the monoid and comonoid are commutative, i.e.

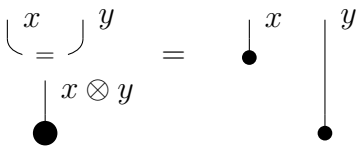
$$\texttt{spider}_{\phi,a+b,c+d}(x) \fatsemi B(x^c, x^d) \;=\; \texttt{spider}_{\phi,a+b,c+d}(x) \;=\; B(x^a, x^b) \fatsemi \texttt{spider}_{\phi,a+b,c+d}(x)$$



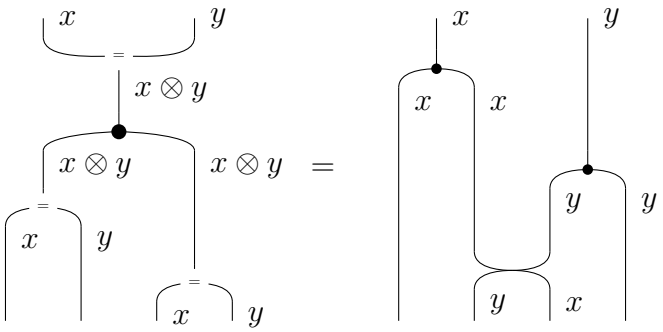
Together with spider fusion, this implies that the monoid of phases is also commutative. The PROB of commutative Frobenius algebras (without the special condition), i.e. diagrams with only spiders and braids, defines a notion of "well-behaved" 1d subspaces of 3d space, up to continuous deformation. When the category is furthermore symmetric, the PROP of commutative spiders defines a notion of "well-behaved" 1d spaces up to diffeomorphism, or equivalently 1d subspaces of 4d space, i.e. one where wires can pass through each other and all knots untie. It is equivalent to the category of two-dimensional *cobordisms* [Abr96], i.e. oriented 2d manifolds with a disjoint union of circles as boundary. Intuitively, a 2d cobordism can be thought of as a (non-planar) wire with a width, i.e. one that we can draw.

If C is braided, we can also give an inductive definition of spiders for tensors. Indeed, given the spiders for x and y we can construct the following comonoid:

- $\texttt{spider}_{1,0}(x \otimes y) \;=\; \texttt{spider}_{1,0}(x) \otimes \texttt{spider}_{1,0}(y)$,



- $\texttt{spider}_{1,2}(x \otimes y) \;=\; \texttt{spider}_{1,2}(x) \otimes \texttt{spider}_{1,2}(y) \fatsemi x \otimes S(x,y) \otimes y$



and a construct a monoid in a symmetric way, then show that they satisfy the spider fusion equations for $x \otimes y$. We can also show that the identity of the unit defines a family of spiders, i.e. $\texttt{spider}_{a,b}(1) = \texttt{id}(1)$. If we take them as axioms rather than definitions, these are called the *coherence conditions* for spiders.

Thus we get to our definition: a *hypergraph category* is a symmetric category with coherent special commutative spiders on each object. We can take the data to be that of a foo-monoidal category C together with a function $\texttt{spider} : \mathbb{N} \times \mathbb{N} \times C_0 \to C_1$ or equivalenty, with four functions $\texttt{merge}, \texttt{unit}, \texttt{split}, \texttt{counit} : C_0 \to C_1$. Once we fix the spiders for generating objects, we get spiders for any type (i.e. list of objects). A hypergraph functor is a symmetric functor $F : C \to D$ between hypergraph categories such that $F \fatsemi \texttt{spider}_{a,b} = \texttt{spider}_{a,b} \fatsemi F$. Thus we get a category **HypCat** with a forgetful functor $U : \mathbf{HypCat} \to \mathbf{MonSig}$. Its left adjoint $F^H : \mathbf{MonSig} \to \mathbf{HypCat}$ is defined as a quotient $F^S(\Sigma^H)/R$ of the free symmetric category generated by $\Sigma^H = \Sigma\texttt{spider}$ and the relation R given by the equations for commutative spiders. Equivalently, we can take $\Sigma^H = \bigcup\{\Sigma, \texttt{merge}, \texttt{unit}, \texttt{split}, \texttt{counit}\}$ and R given by the equations for special commutative Frobenius algebras. A †-*hypergraph category* is a †-symmetric category (i.e. the swaps are unitaries) where the dagger is a hypergraph functor. We also require that the monoid of phases is in fact a group with the dagger as inverse or equivalently, that phase shifts are unitaries.

Example 0.1.28. *Every commutative rig* $\mathbb{S}$ *induces a* †*-hypergraph category* $\mathbf{Tensor}_{\mathbb{S}}$ *with the transpose as dagger. Arguably, special commutative Frobenius algebras were*

first defined by Peirce [Pei06] with their interpretation in the category of relations, or equivalently $\mathbf{Tensor}_{\mathbb{B}}$. *Indeed, they correspond to what Peirce calls* lines of identity*: they express in two dimensions what one-dimensional first-order logic would express with equality symbols. For example, take a binary predicate encoded as a box* $p : 1 \to x^2$ *(interpreted as the formula* $\exists\, a \cdot \exists\, b \cdot p(a, b)$*) then the diagram* $p \fatsemi \texttt{merge}(x)$ *is interpreted as the formula* $\exists\, a \cdot \exists\, b \cdot p(a, b) \wedge a = b$ *or equivalently* $\exists\, a \cdot p(a, a)$*. Thus, every first-order logic formula can be written as a diagram with boxes for predicates, spiders for identity and bubbles for negation. The equivalence of formulae can be defined as a quotient of a free hypergraph category with bubbles, i.e. all the rules of first-order logic can be given in terms of diagrams.*

Example 0.1.29. *The category of complex tensors* $\mathbf{Tensor}_{\mathbb{C}}$ *is †-hypergraph with the spiders given in example 0.1.26. Any unitary matrix* $U : n \to n$ *defines another family of spiders* $U^{\otimes a} \fatsemi \texttt{spider}_{\phi,a,b}(n) \fatsemi (U^\dagger)^{\otimes b}$. *In fact, every unitary arises in this way, see Heunen and Vicary [HV19, Corollary 5.32]. Thus, the axioms for spiders allow us to define any orthonormal basis without ever mentioning basis vectors: they are merely the states* $v : 1 \to n$ *for which the comonoid is natural, i.e.* $v \fatsemi \texttt{split}(x) = v \otimes v$ *and* $v \fatsemi \texttt{counit}(x) = \texttt{id}(1)$.

Example 0.1.30. *With the spiders defined in example* **??**, $\mathbf{Circ}$ *is a †-hypergraph category and the evaluation functor* $\mathbf{Circ} \to \mathbf{Tensor}_{\mathbb{C}}$ *is a †-hypergraph functor.*

DisCoPy implements spiders for types of length one (i.e. generating objects) as a subclass of `Box` and spiders for arbitrary types as a method `Diagram.spiders`. That is, we make every free category and every functor †-hypergraph by default.

Listing 0.1.31. Implementation of †-hypergraph categories and functors.

```
class Spider(Box):
    def __init__(self, a: int, b: int, x: Ty, phase=None):
        assert len(x) == 1
        self.object, self.phase = x, phase or 0
        name = "Spider({})".format(', '.join(map(str, (a, b, x, phase))))
        super().__init__(name, dom=x ** a, cod=x ** b)

    def dagger(self):
        a, b, x = len(self.cod), len(self.dom), self.object
        phase = None if self.phase is None else -self.phase
        return Spider(a, b, x, phase)

def coherence(factory):
    @classmethod
```

```
    def method(cls, a: int, b: int, x: Ty, phase=None) -> Diagram:
        if len(x) == 0:
            assert phase is None
            return cls.id(x)
        if phase is not None:  # Coherence for phase shifters.
            shift = factory(1, 1, x, phase)
            return method(cls, a, 1, x) >> shift >> method(cls, 1, b, x)
        if (a, b) in [(1, 0), (0, 1)]: # Coherence for (co)units.
            return cls.tensor([factory(1, 1, obj) for obj in x])
        if (co, a, b) == [(True, 1, 2), (False, 2, 1)]:  # Coherence for (co)products.
            if len(x) == 1: return factory(a, b, x)
            co_product = factory(a, b, x[0]) @ method(cls, a, b, x[1:])
            braid = x[0] @ cls.braid(x[0], x[1:]) @ x[1:]
            return co_product >> braid if co else braid >> co_product
        if a == 1:  # We can now assume b > 2.
            return method(cls, 1, b - 1, x) >> method(cls, 1, 2, x) @ (x ** (b - 2))
        if b == 1:  # We can now assume a > 2.
            return method(cls, 2, 1, x) @ (x ** (a - 2)) >> method(cls, a - 1, 1, x)
        return method(cls, a, 1, x) >> method(cls, 1, b, x)

Diagram.spiders = coherence(Spider)

class Functor(braided.Functor):
    def __call__(self, other):
        if isinstance(other, Spider):
            a, b, x, phase = len(other.dom), len(other.cod), other.object, other.phase
            return self.cod.ar.spiders(a, b, self(x), phase)
        return super().__call__(other)
```

Example 0.1.32. *FOL formula as a diagram model as a functor*

The equality of hypergraph diagrams reduces to hypergraph isomorphism, it will be discussed in section 0.3. The equality of non-commutative spiders is not implemented yet, spider fusion would be a natural extension of the snake removal algorithm for rigid diagrams: we find pairs fusable spiders then apply interchangers to make them adjacent. The possible obstructions are more serious for spiders than for cups and caps however, for example consider the diagram $\texttt{spider}_{0,3}(x) \fatsemi x \otimes f \otimes g \fatsemi \texttt{spider}_{3,0}(x)$. The two three-legged spiders want to fuse but the boxes f and g stand on the way, the best we can do is to bend their output wires with two cups and get a four-legged spider $\texttt{spider}_{0,4}(x) \fatsemi x \otimes f \otimes g \otimes x \fatsemi \texttt{cup}(x) \otimes \texttt{cup}(x)$.

0.1.4 Cartesian closed categories

With hypergraph diagrams, we have enough syntax to discuss quantum protocols and first-order logic. However, the spiders of hypergraph categories are of no use if we want to interpret our diagrams as (pure) Python functions. Indeed, **Pyth** has the property that every function $f : x \to y \otimes z$ into a composite system $y \otimes z$ is in fact a tensor product $f = f_0 \otimes f_1$ of two separate functions $f_0 : x \to y$ and $f_1 : x \to z$. If a Python type x had caps (let alone spiders) then we could break them in two with the consequence that the identity function on x is constant, i.e. x is trivial [CK17, ?]. Moreover, there is only one (pure) effect of every type, discarding it. Thus if a Python type x had cups then we could break them apart as well with the same consequence: only the trivial Python type can have spiders. A similar argument destroys our hopes for time reversal in Python: if we had a dagger on **Pyth**, every function would be equal to every other.

Now if we go back to the intuition of diagrams as pipelines and their wires as carrying data, not all might be lost about spiders. Indeed, it makes sense to split a data-carrying wire: it means we are copying information. Closing a data-carrying wire is the counit of the copying comonoid, it means we are deleting information. In this context, the special condition would translate as follows: if we copy some data then merge the two copies back together, then we haven't done anything. In order for the spider fusion equations to hold, we would need the monoid to take any two inputs and assert that they are equal or abort the computation otherwise, i.e. we would need side effects. Even more weirdly, we would need the unit of the monoid to be equal to anything else.

Rather than complaining that classical computing is weird because we cannot coherently merge data back together, we should embrace this as a feature, not a bug: in Python we can copy and discard data (at least assuming that we have enough RAM and that the garbage collector is doing its job). This means we can still keep the comonoid half of our spiders, forget that they are spiders and come to realise that they are in fact *natural comonoids*. Indeed, the functions `copy = lambda *xs: xs + xs` and `delete = lambda *xs: ()` define a pair of natural transformations $x \to x \otimes x$ and $x \to 1$ in **Pyth**:

- `copy(f(xs)) == f(copy(xs)[:n]), f(copy(xs)[n:])`
- `delete(f(xs)) == delete(xs)`

for all pure functions `f` and inputs `xs` with `n = len(xs)`.

A *cartesian category* is a symmetric category with coherent, natural commutative comonoids, a cartesian category that is also a PROP is called a *Lawvere theory* [Law63]. Lawvere theories were introduced as a high-level language for *universal algebra*: take the boxes $x^n \to x$ to be the primitive n-ary operations of your language (e.g. for rigs we have unary boxes for 0 and 1, binary boxes for $+$ and $\times$) then the diagrams in the free Lawvere theory are all the terms of your language. We can write down any universally quantified axiom as a relation between diagrams and the cartesian functors (i.e. the symmetric functors that respects comonoids) from the resulting quotient to the category of sets are *algebras*, i.e. sets equipped with operations that satisfy the axioms. The natural transformations between models are precisely the homomorphisms between the algebras, i.e. the functions that commute with the operations. If we add colours back in and allow many generating objects, we can define many-sorted theories such as that of modules, with a ring acting on a group. If we take every pair of objects $(x, y) \in C_0 \times C_0$ as colour and a box $(x, y) \otimes (y, z) \to (x, z)$ for every possible composition, then we can even define the Lawvere theory of categories with C_0 as objects. Thus, a category can also be seen as a functor from this Lawvere theory to sets, a functor can also be seen as a natural transformation between such functors.

0.2 A premonoidal approach

0.2.1 Abstract premonoidal categories

0.2.2 Concrete premonoidal categories

0.2.3 Free premonoidal categories

0.2.4 The state construction

0.3 Related & future work

0.3.1 Graph-based data structures

0.3.2 Higher-dimensional diagrams

1

Quantum natural language processing

2

Diagrammatic differentiation

References

[Abr96] L. Abrams. "Two-Dimensional Topological Quantum Field Theories and Frobenius Algebras". In: *Journal of Knot Theory and Its Ramifications* 05.05 (1996), pp. 569–587. DOI: 10.1142/S0218216596000333. eprint: http://www.worldscientific.com/doi/pdf/10.1142/S0218216596000333.

[CK17] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge: Cambridge University Press, 2017. DOI: 10.1017/9781316219317.

[DV21] Antonin Delpeuch and Jamie Vicary. "The Word Problem for Braided Monoidal Categories Is Unknot-Hard". In: *arXiv:2105.04237 [math]* (May 2021). arXiv: 2105.04237 [math].

[DV19] Lawrence Dunn and Jamie Vicary. "Coherence for Frobenius Pseudomonoids and the Geometry of Linear Proofs". In: *arXiv:1601.05372 [cs]* (2019). DOI: 10.23638/LMCS-15(3:5)2019. arXiv: 1601.05372 [cs].

[Hak61] Wolfgang Haken. "Theorie Der Normalflächen". In: *Acta Mathematica* 105.3 (1961), pp. 245–375.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Oxford Graduate Texts in Mathematics. Oxford: Oxford University Press, 2019. DOI: 10.1093/oso/9780198739623.001.0001.

[JS88] André Joyal and Ross Street. "Planar Diagrams and Tensor Algebra". In: *Unpublished manuscript, available from Ross Street's website* (1988).

[Lac15] Marc Lackenby. "A Polynomial Upper Bound on Reidemeister Moves". In: *Annals of Mathematics* (2015), pp. 491–564.

[Lam99] Joachim Lambek. "Type Grammar Revisited". In: *Logical Aspects of Computational Linguistics*. Ed. by Alain Lecomte, François Lamarche, and Guy Perrier. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–27.

[Lau05] Aaron D. Lauda. "Frobenius Algebras and Planar Open String Topological Field Theories". In: *arXiv:math/0508349* (Aug. 2005). arXiv: math/0508349.

[Law63] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. In: *Proceedings of the National Academy of Sciences of the United States of America* 50.5 (1963), pp. 869–872.

[PSV21] Evan Patterson, David I. Spivak, and Dmitry Vagner. “Wiring Diagrams as Normal Forms for Computing in Symmetric Monoidal Categories”. In: *arXiv:2101.12046 [cs]* (Jan. 2021). DOI: 10.4204/EPTCS.333.4. arXiv: 2101.12046 [cs].

[Pei06] Charles Santiago Sanders Peirce. “Prolegomena to an Apology of Pragmaticism”. In: *The Monist* 16.4 (1906), pp. 492–546.

[Pen71] Roger Penrose. “Applications of Negative Dimensional Tensors”. In: *Scribd* (1971).

[PL07] Anne Preller and Joachim Lambek. “Free Compact 2-Categories”. In: *Mathematical Structures in Computer Science* 17.2 (2007), pp. 309–340. DOI: 10.1017/S0960129506005901.

[Rei13] Kurt Reidemeister. *Knotentheorie*. Vol. 1. Springer-Verlag, 2013.

[Sel07] Peter Selinger. “Dagger Compact Closed Categories and Completely Positive Maps: (Extended Abstract)”. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005) 170 (Mar. 2007), pp. 139–163. DOI: 10.1016/j.entcs.2006.12.018.