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 Rigidity: 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 $\texttt{cap}(x) : 1 \to x \otimes x^l$ called *cup* and *cap* (equivalently, counit and unit) such that:

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

These are called the *snake equations* because they are drawn as follows.

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, i.e.

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

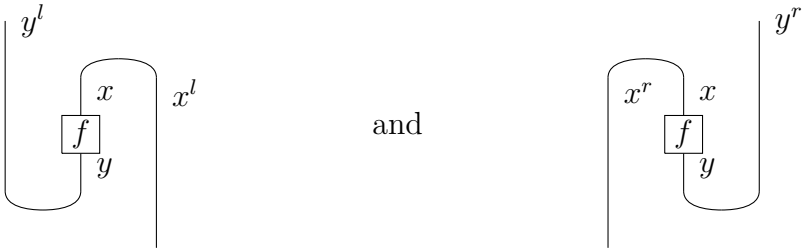
- and $\texttt{cup}(1) = \texttt{cap}(1) = \texttt{id}(1)$.

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



When these two functors coincide, C is called *pivotal*: we can rotate diagrams by 360 degrees. This is the case for any rigid category C with a dagger structure: the dagger of cups and caps for an object x are caps and cups for its left-adjoint x^l. If the transpose functors are the identity-on-objects, i.e. every object is its own left and right adjoint, C is called *self-dual*. In a self-dual pivotal category, the snake equations implie that transpose is in fact an identity-on-objects involution, i.e. a dagger. Note that if a category is self-dual and strictly rigid, then its objects must be a commutative monoid: $x \otimes y = (y \otimes x)^l = y \otimes x$. $(\mathbb{N}, +, 0)$ is the only free monoid that is commutative, hence self-dual, strictly rigid and foo (free-on-objects) together imply that the category is a PRO, i.e. natural numbers as objects with addition as tensor.

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.*

Example 0.1.2. *The foo-monoidal category* $\mathbf{Tensor}_{\mathbb{S}}$ *(with lists of natural numbers as objects and tensors as arrows) is pivotal. Left and right adjoints are 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. By monoidal equivalence, the non-foo category of matrices with natural numbers as objects* $\mathbf{Mat}_{\mathbb{S}} \simeq \mathbf{Tensor}_{\mathbb{S}}$ *is also pivotal. It is furthermore self-dual: multiplication is commutative so the product of a list is equal to that of its reversal.*

Example 0.1.3. *The foo-monoidal category* **Circ** *is self-dual 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: this is the first result of categorical quantum mechanics [AC08].*

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* **??***. 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 them up to equality. Thus, we have defined a subcategory $\mathbf{RigidCat} \hookrightarrow \mathbf{MonCat}$. 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.

[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 $x = x \otimes x^l \otimes x$.

There is a forgetful functor $U : \mathbf{RigidCat} \to \mathbf{RigidSig}$ which sends any strictly-rigid foo-monoidal category to its underlying rigid signature. We now describe its left-adjoint $F^r : \mathbf{RigidSig} \to \mathbf{RigidCat}$. Given a rigid signature Σ, we define a monoidal signature $\Sigma^r = \Sigma \cup \{\mathsf{cup}(x)\}_{x \in \Sigma_0} \cup \{\mathsf{cap}(x)\}_{x \in \Sigma_0}$. 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))

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

class Ty(monoidal.Ty, Ob):
    def __init__(self, objects: list[Ob | str] = None):
        monoidal.Ty.__init__(self, objects=map(Ob.upgrade, objects or []))

    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 that* `Ty` *satisfies the axioms for objects in strictly rigid categories.*

```
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`. We need to be careful with the order of inheritance however, diagram equality is defined in terms of box equality, so if we had `Box.__eq__ = Diagram.__eq__` then checking equality would enter an infinite loop. `Cup` (`Cap`) is a subclass of `Box` initialised by a pair of types of length one `x` and `y` such that `x == y.l` (`x.l == y`, respectively). They are attached to the diagram class with `Diagram.cup, Diagram.cap = Cup, Cap`. The class methods `cups` and `caps` construct cups and caps for arbitrary types by repeated calls to `cup` and `cap`. Thus, when we define a subclass of `Diagram`, we only need to define `cup` and `cap` on generating objects and `cups` and `caps` will be defined for all types by induction.

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

```
class Diagram(monoidal.Diagram):
    @classmethod
    def cups(cls, x, y):
        if len(x) == 0: return cls.id(Ty())
        if len(x) == 1: return cls.cup(x, y)
        return self.id(x[0]) @ cls.cups(x[1:], y[:-1]) @ self.id(y[-1])\
            >> cls.cup(x[0], y[-1])

    @classmethod
    def caps(cls, x, y): ...  # Symmetric to cups.

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

class Cap(Box):
    def __init__(self, x, y):
        assert len(x) == 1 and x.l == y
```

```
        super().__init__("Cap({}, {})".format(x, y), Ty(), x @ y)

Diagram.cup, Diagram.cap = Cup, 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.

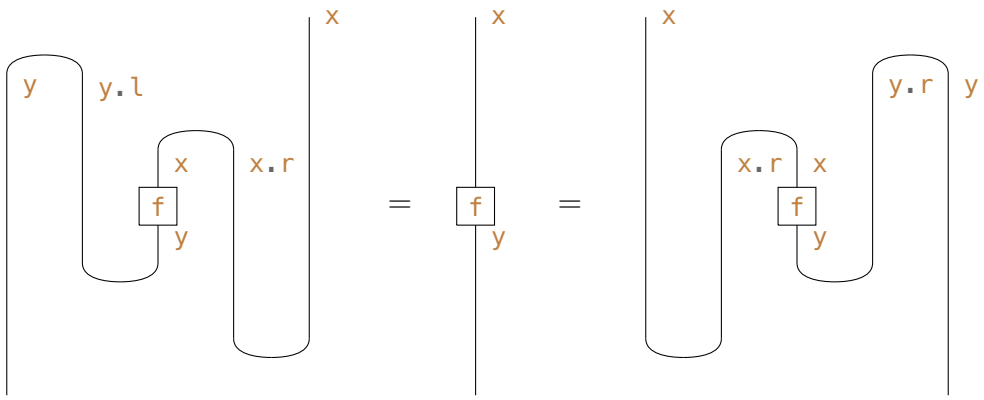
```
Obstruction = tuple[list[int], list[int]]
Yankable = tuple[int, int, Obstruction, bool]

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.*

```
left_snake = Cap(x, x.l) @ x >> x @ Cup(x.l, x)
right_snake = x.l @ Cap(x, x.l) >> Cup(x.l, x) @ x.l

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

Example 0.1.10. *We can check that left and right transpose cancel.*

```
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 self-dual pivotal category.

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

class Circuit(monoidal.Circuit, Diagram):
    cup = lambda _, _: sqrt2 @ Ket(0, 0) >> H @ Id
    cap = lambda x, y: Circuit.cap(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 `ob_factory` does 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):
    ob_factory, ar_factory = 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.ob_factory, '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.ar_factory.cups(self(other.dom[:1]), self(other.dom[1:]))
        if isinstance(other, Cap):
            return self.ar_factory.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 = lambda x, y: Tensor(Tensor.id(x).inside, x + y, [])
Tensor.caps = lambda x, y: Tensor(Tensor.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]]}
    ob_factory=list[int], ar_factory=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. 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 each box. When defining the normal form of pivotal diagrams, we would need to make a choice between the diagrams for left or right transpose. Another solution is to add a new box $f^T : y^r \to x^r$ 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 also has a dagger, we get a four-fold symmetry on each box: itself, its dagger, its transpose and its dagger-transpose (also called its conjugate). This is still being developed by the DisCoPy community.

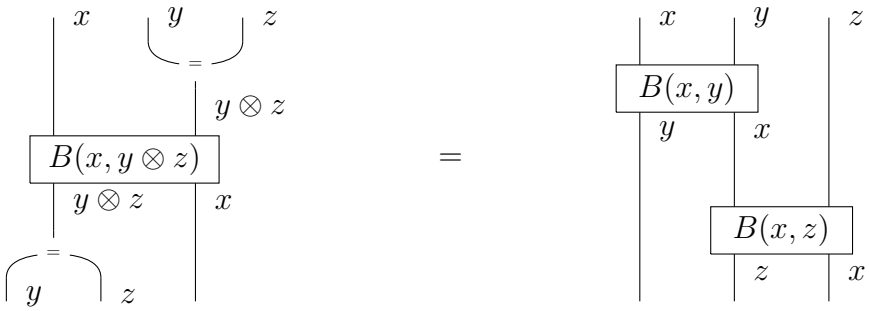
0.1.2 Symmetry: wire swapping

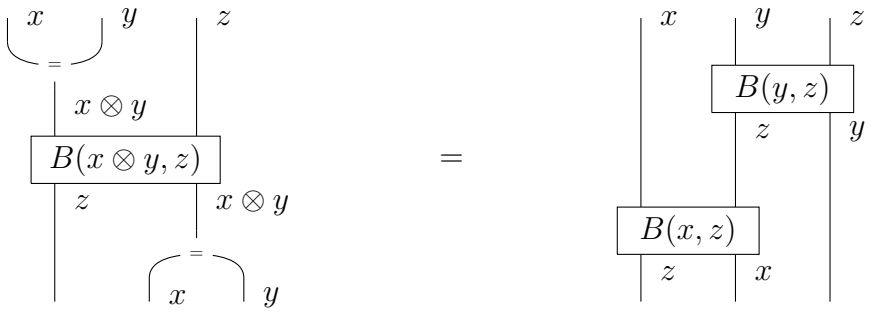
With rigid and pivotal categories, we have removed the assumption that diagrams are progressive: we can bend wires. With braided and symmetric monoidal categories, we now remove the planarity assumption: we can swap wires.

A monoidal category C is *braided* when it comes with a natural isomorphism $B(x, y) : x \otimes y \to y \otimes x$ subject to the following *hexagon equations*:

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

which owe their name to the shape of the corresponding commutative diagrams when C is non-strict monoidal. 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**. We can draw the hexagon equations as non-free-on-objects diagrams, i.e. with explicit equality boxes:



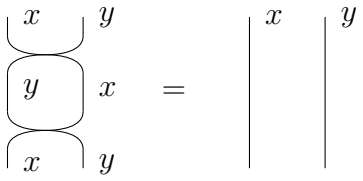
and 

This can be taken as an inductive definition: the braiding $B(x, 1)$ of an object with the unit 1 is the identity, and 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 braided category C is *symmetric* if the natural transformation B is its own inverse $B = B^{-1}$, in this case it is called a *swap*. A symmetric functor is a braided functor between symmetric categories, thus we get a category **SymCat**, with a forgetful functor U : **SymCat** $\to$ **MonSig**. We now describe its left adjoint. Given a monoidal signature Σ, the free symmetric category is a quotient $F^s(\Sigma) = F(\Sigma \cup \{B(x, y)\}_{x,y \in \Sigma_0})/R$ of the free monoidal category generated by Σ and the swaps B for each pair of generating objects $x, y \in \Sigma_0$. The relation R is given by the following axioms for a self-inverse natural transformation:

- $B(x, y) \fatsemi B(y, x) = \mathtt{id}(x \otimes y)$,
- $f \otimes x \fatsemi B(b, x) = B(a, x) \fatsemi x \otimes f$,
- $x \otimes f \fatsemi B(x, b) = B(x, a) \fatsemi f \otimes x$,

for all generating objects $x, y \in \Sigma_0$ and boxes (including swaps) $f : a \to b$ in $\Sigma_1 \cup \{B(x, y)\}_{x,y \in \Sigma_0}$. From B being self-inverse 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. Drawing B as a swap, we get the following diagrammatic equations:



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:

DisCoPy implements free symmetric categories with a class `Swap` for types of length one and a class method `Diagram.swaps` for types of arbitrary length.

Listing 0.1.16. Implementation of free symmetric categories and functors.

```
class Swap(monoidal.Box):
    def __init__(self, x: Ty, y: Ty):
        assert len(x) == len(y) == 1
        super().__init__("Swap({}, {})".format(x, y), x @ y, y @ x)

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

Diagram.swap, Diagram.swaps = Swap, classmethod(swaps)

class Functor(monoidal.Functor):
    def __call__(self, other):
        return self.ar_factory.swaps(self(other[0]), self(other[1]))\
            if isinstance(other, Swap) else super().__call__(other)
```

Listing 0.1.17. Implementation of **Pyth** and $\mathbf{Tensor}_{\mathbb{S}}$ as symmetric categories.

```
Function.swap = lambda t0, t1:\
    Function(lambda x, y: (y, x), [t0, t1], [t1, t0])
Function.swaps = classmethod(swaps)

def tensor_swap(cls, m: int, n: int):
    inside = [[(i0, j0) == (i1, j1) for j0 in range(n) for i0 in range(m)]
                                    for i1 in range(m) for j1 in range(n)]
    return cls(inside, [m], [n])
Tensor.swap, Tensor.swaps = classmethod(tensor_swap), classmethod(swaps)
```

Free braided categories are defined in a similar way, with boxes for both the braid B and its inverse B^{-1}.

[DV21]

0.1.3 Hypergraph: wire splitting

0.1.4 Cartesian closed categories

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

References

[AC08] Samson Abramsky and Bob Coecke. "Categorical Quantum Mechanics". In: *arXiv:0808.1023 [quant-ph]* (Aug. 2008). arXiv: 0808.1023 [quant-ph].

[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].

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

[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.

[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.