Cambridge Quantum

DisCoPy — Monoidal Categories in Python

by Giovanni de Felice, Alexis Toumi, Bob Coecke

Cambridge Quantum
Department of Computer Science
University of Oxford, United Kingdom
GIOVANNI DE FELICE
giovanni.defelice@cs.ox.ac.uk
ALEXIS TOUMI
alexis.toumi@cs.ox.ac.uk
BOB COECKE
bob.coecke@cs.ox.ac.uk

DisCoPy: Monoidal Categories in Python

Giovanni de Felice, Alexis Toumi, Bob Coecke

Department of Computer Science, University of Oxford.   Cambridge Quantum Computing Ltd.

`{firstname.lastname}@cs.ox.ac.uk`

We introduce DisCoPy, an open source toolbox for computing with monoidal categories. The library provides an intuitive syntax for defining string diagrams and monoidal functors. Its modularity allows the efficient implementation of computational experiments in the various applications of category theory where diagrams have become a lingua franca. As an example, we used DisCoPy to perform natural language processing on quantum hardware for the first time.

Introduction

String diagrams are a graphical calculus for monoidal categories, introduced independently by Hotz [34] in computer science and Penrose [56] in physics, then formalised by Joyal and Street [35, 36]. Graphical languages are surveyed in Selinger [62], they have become a standard tool in applied category theory. We cite a few of the growing list of applications: computer science [12, 1], quantum theory and the ZX-calculus [16, 2, 19], networks and control theory [5, 6, 7], concurrency [11], databases and knowledge representation [55, 10], Bayesian reasoning and causality [21, 13, 39], linguistics and cognition [14, 9], functional programming [60], machine learning and game theory [29, 30]. In all these applications, string diagrams are the syntax and structure-preserving functors are used to compute their semantics in concrete categories.

There are several existing proof assistants with graphical user interfaces for rewriting string diagrams in a more or less automated fashion: quantomatic [41] and PyZX [40] for the ZX-calculus, globular [8] and its successor homotopy.io [59] for higher categories, cartographer [65] for symmetric monoidal categories. However, these are all stand-alone tools which use different task-specific encodings for diagrams, preventing interoperability between them and with the software ecosystems of application domains.

DisCoPy (Distributional Compositional Python) is not yet-another rewriting tool, rather it is meant as a toolbox for compiling diagrams into code, be it for neural networks, tensor computation or quantum circuits. It provides an intuitive Python syntax for diagrams, allowing to visualise and reason about computation at a high level of abstraction. Monoidal functors allow to translate these diagrams into concrete computation, interfacing with optimised task-specific libraries. DisCoPy is an open source package, it is available with an extensive documentation and demonstration notebooks hosted at:

`https://github.com/oxford-quantum-group/discopy`

This paper describes the architecture of DisCoPy, focusing on the translation from abstract categorical definitions into their concrete implementation in Python. We assume some working knowledge of category theory and refer the reader to [49] and to [3] for an introduction. Implementing a category in an object-oriented programming language amounts to defining a pair of classes for its objects and arrows, together with a pair of methods for identity and composition. When the category is free, composition is implemented by list concatenation and identity by the empty list. Concrete categories may then be defined by subclassing this free category and overriding identity and composition. These are expected to respect the usual associativity and unit axioms, however they cannot be formally checked in Python.

Starting from free categories (section 1) as a base class, more structure can be added by subclassing and adding new methods. Quotient categories can be implemented by a method for computing normal forms. For instance, monoidal categories (section 2) subclass categories with an extra method for tensor product and one for interchanger normal form. For now, we implemented Cartesian and rigid monoidal categories (section 3), as these provide a syntax for the concrete categories implemented in DisCoPy: Python functions (appendix A) and numpy [69] tensors (section 4). The development of DisCoPy was first motivated by the implementation of natural language processing on quantum hardware. Hence, we implemented quantum circuits (section 5) as a subclass of rigid monoidal categories with an extra method for evaluation as numpy tensors and interface with the t|ket⟩ compiler [64].

We hope that this toolbox will prove to be of use to the applied category theory community, and plan on adding more categorical tools to it in the future.

1   cat.py

We give a brief introduction to the concepts of signatures, free categories and functors, fixing some notation for the rest of the paper. We then describe how these concepts, once implemented in Python, give the architecture of `cat`, the first module of the DisCoPy package.

A *simple signature* Σ is given by a pair of sets Σ_0, Σ_1 of generating objects and arrows, and a pair of functions $\mathsf{dom}, \mathsf{cod} : \Sigma_1 \to \Sigma_0$ called the domain and the codomain respectively. A morphism of signatures $F : \Sigma \to \Sigma'$ is a pair of functions $F_0 : \Sigma_0 \to \Sigma'_0$, $F_1 : \Sigma_1 \to \Sigma'_1$ which commute with dom and cod. The *free category* $\mathbf{C}(\Sigma)$ generated by a signature Σ, is defined as follows: the objects are given by Σ_0 and an arrow $f : s \to t$ is given by a list $f = f_1 \dots f_n \in \Sigma_1^n$ such that $\mathsf{dom}(f_1) = s$, $\mathsf{cod}(f_n) = t$ and $\mathsf{cod}(f_i) = \mathsf{dom}(f_{i+1})$ for all $i \leq n$. Identity arrows are given by the empty list and composition is given by list contatenation. The universal property of $\mathbf{C}(\Sigma)$ may be stated as follows: functors $F : \mathbf{C}(\Sigma) \to \mathbf{D}$ from a free category are uniquely defined by their image on the signature Σ, i.e. by a morphism of signatures $F : \Sigma \to U(\mathbf{D})$ for U the forgetful functor. These abstract definitions are implemented in an object-oriented fashion with the following Python classes, where the tuples denote the arguments to the corresponding class constructors.

class `cat.Ob(name)` is given by any Python object `name`. Equality of objects is given by the equality of names, i.e. `x == y` if and only if `x.name == y.name`.

class `cat.Arrow(dom, cod, boxes)` is given by a pair of `Ob`-instances `dom, cod` and a sequence of `Box`-instances `boxes`. Axioms are checked at initialisation, `AxiomError` is raised otherwise. An `Arrow`-instance `f` has the following methods:

- `f.dagger()` returns the dagger of an arrow, it has syntactic sugar `f[::-1]`.
- `f.then(g)` returns the composition of two arrows, it is written `f » g` or `g « f`.
- `Arrow.id(x)` returns the identity arrow on a given object, it is shortened to `Id(x)`.

Axioms are implemented as part of the testing suite for the DisCoPy package: associativity `f » (g » h) == (f » g) » h` and unitality `f » Id(f.cod) == f == Id(f.dom) » f` follow directly from the monoid structure of Python sequences. `Arrow`-instances can be manipulated using the standard Python syntax for sequences: they are iterable, indexable and sliceable, i.e. `f[i] == f.boxes[i]` and `f[i:j]` returns the arrow `f[i] » ... » f[j]`. Printing an arrow yields a DisCoPy expression which generates it, which proves useful for debugging and interactive programming.

class `cat.Box(name, dom, cod, data=None, _dagger=False)` is a subclass of `Arrow`, i.e. a box `f` is defined as an arrow with `f.boxes == [f]`. `data` is an optional argument, it can be used to attach arbitrary Python data to a box. `_dagger` is an optional Boolean argument, set to `False` by default. It is used to construct an involutive identity-on-objects contravariant endo-functor, i.e. a dagger.

class `cat.Functor(ob, ar, ob_factory=Ob, ar_factory=Arrow)` is given by a pair of mappings `ob, ar` from `Ob` to `ob_factory` and from `Box` to `ar_factory` respectively. The domain of a `Functor`-instance `F` is defined implicitly as the free category generated by the domain of `ob` and `ar`. The two factory methods allow to build functors with arbitrary codomains, `ar_factory` is required to provide methods `id` and `then`.

Example 1.1 *In order to illustrate the syntax, we give a basic example of a free functor.*

```
x, y, z = Ob('x'), Ob('y'), Ob('z')
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', z, x)
F = cat.Functor(ob={x: y, y: z, z: x}, ar={f: g, g: h})
assert F(f >> g) == F(f) >> F(g) == g >> h
```

Example 1.2 `PythonFunctor` *implements functors into Python functions, see appendix A.*

```
F = PythonFunctor(
    ob={x: 0, y: 1, z: 2}, ar={f: lambda: 42, g: lambda x: (x, x + 1)})
assert F(f >> g)() == (42, 43)
```

Example 1.3 *The category of matrices is implemented by the* `Tensor` *class, see section 4.*

```
F = TensorFunctor(ob={x: 1, y: 2, z: 2}, ar={f: [0, 1], g: [0, 1, 1, 0]})
assert F(f >> g) == F(f) >> F(g) == [1, 0]
```

All the objects, arrows, boxes and functors constructed with DisCoPy are representable in the following sense: for any instance `x`, the string `repr(x)` is a valid Python expression which evaluates to `x`, assuming that the names defining `x` are representable themselves. Hence, DisCoPy data structures can be serialized and exported in a standard format [66] for interoperability with other category theory tools. Note that while the data structures and methods of DisCoPy are purely functional, boxes may hold mutable a `data` attribute. It is used to build boxes that are not finitely generated such as the weights of neural networks (appendix A) or the phases of quantum circuits (section 5).

2   monoidal.py

In this section, we describe the core data structure behind DisCoPy: the `Diagram` class, an implementation of the arrows of a free monoidal category. We begin with a definition of free monoidal categories via free *premonoidal categories*.

Definition 2.1 *A (strict) monoidal category is a category* **C** *equipped with an associative and unital functor* $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ *for* $\times$ *the Cartesian product. A (strict) premonoidal category is a category* **C** *equipped with an associative and unital functor* $\boxtimes : \mathbf{C} \,\square\, \mathbf{C} \to \mathbf{C}$ *for* $\square$ *given by the following pushout in* **Cat***:*

$$
\begin{array}{ccc}
\mathbf{C}_0 \times \mathbf{D}_0 & \longrightarrow & \mathbf{C} \times \mathbf{D}_0 \\
\downarrow & & \downarrow \\
\mathbf{C}_0 \times \mathbf{D} & \longrightarrow & \mathbf{C} \,\square\, \mathbf{D}
\end{array}
$$

where $\mathbf{C}_0, \mathbf{D}_0$ *are the discrete category of objects and the maps are given by the inclusions.*

Example 2.2 *The Keisli category for a strong monad over a monoidal category is a premonoidal category. It is monoidal precisely when the monad is commutative. The state monad over the category of sets yields a denotational semantics for side-effects [57].*

Example 2.3 *For a semiring* $\mathbb{S}$*, the distribution monad* $X \mapsto \mathbb{S}^X$ *over the category of finite sets yields a premonoidal category with matrices over* $\mathbb{S}$ *as arrows and Kronecker product as tensor. It is monoidal when* $\mathbb{S}$ *is commutative.*

The pushout $\Box$ is known as the *funny tensor product*, it may also be defined by the following universal property. Let $\mathbf{C} \Rightarrow \mathbf{D}$ be the category where objects are functors $\mathbf{C} \to \mathbf{D}$ and arrows are the transformations with no naturality requirement. Then $- \Box \mathbf{C}$ is characterised as the left adjoint of $\mathbf{C} \Rightarrow -$ in **Cat**. This makes $\Box$ a closed symmetric monoidal structure over **Cat**, the unique such structure apart from the usual Cartesian product, see [27]. A functor from the funny tensor product $\mathbf{C} \Box \mathbf{D}$ can be understood as a functor which is "separately functorial" in its two arguments $\mathbf{C}$ and $\mathbf{D}$, in analogy to separate continuity. Premonoidal categories are also known as one-object sesquicategories (one-and-a-half categories), i.e. 2-categories without the interchange law, see [67]. Our motivation for working in a premonoidal setting is two-fold: 1) free premonoidal categories have a simple presentation as free categories and 2) free monoidal categories may then be described as quotient categories.

We define a *monoidal signature* Σ as a pair of sets Σ_0, Σ_1 with a pair of functions into the free monoid $\mathsf{dom}, \mathsf{cod} : \Sigma_1 \to \Sigma_0^\star$. Generating arrows $f \in \Sigma_1$ are depicted as boxes with input $\mathsf{dom}(f)$ and output $\mathsf{cod}(f)$. Given a monoidal signature Σ, we define a simple signature $L(\Sigma)$ with objects $\Sigma_0^\star$, arrows $\Sigma_0^\star \times \Sigma_1 \times \Sigma_0^\star$ with $\mathsf{dom}(u, f, v) = usv$ and $\mathsf{cod}(u, f, v) = utv$ for $s = \mathsf{dom}(f)$ and $t = \mathsf{cod}(f)$. A layer $(u, f, v) \in \Sigma_0^\star \times \Sigma_1 \times \Sigma_0^\star$ is depicted as a box with wires to its left and right:
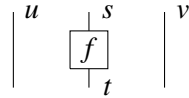
Proposition 2.4 *Given a monoidal signature* Σ*, the free premonoidal category is the free category* $\mathbf{PMC}(\Sigma) = \mathbf{C}(L(\Sigma))$ *generated by the simple signature of layers* $L(\Sigma)$*.*

We define a *diagram* as an arrow of $\mathbf{C}(L(\Sigma))$. Diagrams are uniquely defined by a domain, a list of generators and an *offset* for each box: the number of wires passing to its left. Two diagrams are equal in the free monoidal category if they are related by a series of *interchangers*, where $u, v, w \in \Sigma_0^\star$ and $s \xrightarrow{f} t, s' \xrightarrow{f'} t' \in \Sigma_1$:
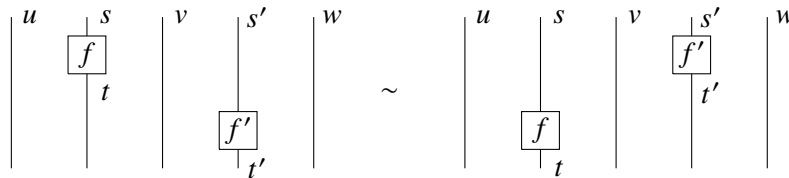
The *right interchangers*, going from the right- to the left-hand side of the previous equivalence are terminating on boundary-connected diagrams. Given a boundary-connected diagram with n boxes, a normal form can be reached in at most $O(n^3)$ steps [25, Theorem 36]. This makes the word problem for monoidal categories — i.e. given two arbitrary diagrams, are they equal up to interchanger? — decidable in polynomial time, see [25, Theorem 48].

Proposition 2.5 *Given a monoidal signature Σ, the free monoidal category is the quotient* **MC**(Σ) = **PMC**(Σ)/$\mathcal{I}$ *for* $\mathcal{I}$ *the interchanger relation, see [25, Theorem 16].*

Example 2.6 *Context-free grammars (CFGs) are a special case of free monoidal categories, where non-terminal symbols are generating objects and production rules are generators with an atomic type as domain. Syntax trees are diagrams, their normal form correspond to the leftmost derivation in a CFG. Weighted CFGs can be defined as free monoidal categories equipped with a monoidal functor into a monoid delooping, see [63].*

Note that a monoidal category generated by one object, i.e. with $\Sigma_0 = \{1\}$, is called a *PRO*. An example is the category of circuits described in section 5. This combinatorial definition yields an implementation of the free monoidal category where the generating objects Σ_0 are given by `Ob`-instances and the generating arrows Σ_1 are given by `Box`-instances.

class `monoidal.Ty(x_1, ..., x_n)` is a subclass of `cat.Ob`. A type `x` is given by a (possibly empty) list of objects, It has a method `x.tensor(y)`, shortened to `x @ y`, which inherits the monoid structure of lists with the empty type `Ty()` as unit.

class `monoidal.PRO(n)` is a subclass of `Ty` generated by `Ob(1)`. A `PRO` type is given a natural number `n` with addition as tensor, i.e. `PRO(n) == Ty(n * (1, ))`.

class `monoidal.Layer(left, box, right)` is a subclass of `cat.Box`. A layer is given by a `Box`-instance `box` and a pair of `Ty`-instances `left` and `right`. While essential to the internal structure of DisCoPy, layers remain invisible to the end user of the package.

class `monoidal.Diagram(dom, cod, boxes, offsets, layers=None)` subclasses `Arrow`. A diagram `f` is given by a pair of types `dom, cod`, a pair of equal-length sequences `boxes, offsets` of `Box`-instances and natural numbers respectively. `layers` is an optional argument, if omitted it will be computed from `dom`, `boxes` and `offsets`. `f` has all the methods of `Arrow` plus the following:

- `f.tensor(g)`, shortened to `f @ g`, returns the diagram with `(f @ g).boxes == f.boxes + g.boxes` and `(f @ g).offsets == f.offsets + [n + len(f.cod) for n in g.offsets]`.

- `f.interchange(i, j, left=False)` returns a diagram with the boxes at indices `i` and `j` interchanged or raises an `InterchangerError` if they are connected. This method gets called recursively whenever `i < j + 1` or `j < i - 1`. If there is a choice in how to interchange, then we apply the right interchanger by default.

- `f.normalize(left=False)` yields a reduction sequence applying right interchangers repeatedly. Setting `left=True` will apply left interchangers instead.

- `f.normal_form(left=False)` returns the last output of `f.normalize` if `f` is boundary-connected, otherwise it raises a `NotImplementedError`.

- `f.draw(to_tikz=False, **params)` draws a diagram using `networkx` [53], see the DisCoPy documentation [23] for a complete list of parameters. Diagrams are displayed with `matplotlib` [51] by default, setting `to_tikz=True` outputs a list of Ti*kz* [68] commands instead. This is how all the diagrams in this article were produced.

- `f.to_gif(g, h, ..., **params)` takes a list of diagrams and returns an animated GIF which can be used to visualise a rewriting process in a jupyter notebook [42].

As for `Arrow`, `Diagram`-instances can be manipulated as lists. Indexing a diagram `f` returns a layer `Id(left) @ box @ Id(right)` such that `f == f[0] » ...  » f[-1]`.

class `monoidal.Box(name, dom, cod, data=None, _dagger=False)` is a subclass of `cat.Box` and `Diagram`. A box `f` is defined as a diagram with `f.boxes, f.offsets == [f], [0]`.

class `monoidal.Functor(ob, ar, ob_factory=Ty, ar_factory=Diagram)` is a subclass of `cat.` `Functor`. A monoidal functor `F` is given by a pair of mappings `ob, ar` from `Ty` to `ob_factory` and from `Box` to `ar_factory` respectively. Factory methods are optional arguments that allow arbitrary codomains. `ob_factory` is required to provide a `tensor` method, `ar_factory` should provide `id`, `then` and `tensor`.

The implementation of the `Diagram` data structure and its normal form follows directly from their formal definitions, they require no further explanation. The implementation of the `draw` method however requires some non-trivial choices: given the combinatorial encoding of a diagram, which embedding on the plane should we return? Moreover, the drawing algorithm may be treated as a proof of the equivalence from the combinatorial to the geometric definition of diagrams introduced by Joyal and Street [35, 36]. The other direction, i.e. from a planar embedding (encoded as a grid of pixels) to its combinatorial encoding as a DisCoPy `Diagram`, is part of an in-development application for automated string diagram recognition. We leave the details to appendix B.

3   rigid.py

In this section, we describe the module `rigid`, an implementation of rigid monoidal categories. We present *snake removal*, the algorithm for normalising rigid diagrams, and its application to the semantics of *pregroup grammars*. Note that rigid categories are also called *autonomous* [35, 24], they are equivalent to the compact 2-categories of Preller, Lambek [58] with one object.

A (strict) monoidal category **C** is *rigid* when every object x has left and right adjoints x^l and x^r and four morphisms $x \otimes x^l \xrightarrow{\epsilon} 1 \xrightarrow{\eta} x^l \otimes x$ and $x^r \otimes x \xrightarrow{\epsilon'} 1 \xrightarrow{\eta'} x \otimes x^r$ depicted as cups and caps, subject to $(\epsilon' \otimes 1_x) \circ (1_x \otimes \eta') = 1_x = (1_x \otimes \epsilon) \circ (\eta \otimes 1_x)$. Note that any monoidal functor between rigid categories is isomorphic to a rigid functor, i.e. which sends cups to cups and caps to caps.

In a rigid category, left and right adjoints are unique up to a unique isomorphism. They cancel each other — i.e. $(x^l)^r = x = (x^r)^l$ — and they are anti-homomorphisms of the tensor product — i.e. $1^l = 1^r = 1$, $(x \otimes y)^l = y^l \otimes x^l$ and $(x \otimes y)^r = y^r \otimes x^r$. Thus, the adjoint of a product can always be written as a product of adjoints. Given a set of atomic types Σ_0, the objects of the free rigid category are given by lists of pairs $(x, n) \in \Sigma_0 \times \mathbb{Z}$ with the inclusion $\Sigma_0 \hookrightarrow \Sigma_0 \times \mathbb{Z}$ defined by $x \mapsto (x, 0)$ and the adjoints $(x, n)^l = (x, n - 1)$ and $(x, n)^r = (x, n + 1)$. We define a *rigid signature* Σ as a pair of sets Σ_0, Σ_1 with a pair of functions $\mathsf{dom}, \mathsf{cod} : \Sigma_1 \to (\Sigma_0 \times \mathbb{Z})^\star$, then the free rigid category is given by the quotient $\mathbf{RC}(\Sigma) = \mathbf{MC}(\Sigma') / \mathcal{R}$ where Σ' is the monoidal signature with $\Sigma'_0 = \Sigma_0 \times \mathbb{Z}$ and $\Sigma'_1 = \Sigma_1 + \{\, \mathsf{cup}_x : x \otimes x^r \to 1, \mathsf{cap}_x : 1 \to x \otimes x^l \,\}_{x \in \Sigma'_0}$. The cups and caps for the unit are the identity, those for $x \otimes y$ are given by nesting the cups and caps of x and y. The relations $\mathcal{R}$ are given by the snake equation for every $x \in \Sigma'_0$:

The `rigid` module implements classes `Ty`, `Diagram`, `Box` and `Functor` which subclass those from `monoidal`. Pregroup types, i.e. the objects of free rigid categories, are implemented as `Ty`-instances `x` with two attributes `x.l` and `x.r` for the adjoints. `Box` has two subclasses `Cup` and `Cap` implementing the adjunction for simple types. The `Diagram` class has two static methods `cups` and `caps` which implement the adjunction for product types. The `normalize` and `normal_form` methods are overriden to implement snake removal: for each pair of cup and cap forming a snake, we first apply interchangers to make them adjacent, then replace the snake with an identity, see [26, Definition 2.12].

Example 3.1 *A rigid category is compact-closed if it is also symmetric monoidal. In that case, the left and right adjoints coincide. The category of matrices over a commutative semiring with Kronecker product as tensor is compact-closed, thus it is rigid.*

Example 3.2 *Lambek's pregroup grammars [46, 47, 48] can be defined in terms of free rigid categories. Indeed, a pregroup is a thin rigid category, i.e. with at most one arrow between any two objects. A pregroup grammar G is given by a vocabulary V, a finite set of basic types B with $s \in B$ the sentence type and a finite dictionnary $D \subseteq V \times (B \times \mathbb{Z})^\star$ assigning pregroup types to words. Let Σ_G be the rigid signature with generating objects $V + B$ and arrows $w \to t$ for each dictionnary entry $(w,t) \in D$ and $\mathbf{G} = \mathbf{RC}(\Sigma_G)$. The language of G is given by $L(G) = \{u \in V^\star \mid \exists\, f : u \to s \in \mathbf{G}\}$. That is, a list of words $u \in V^\star$ is a grammatical sentence whenever there is a diagram $f = g \circ d$ with $d : u \to t$ a product of dictionnary entries and $g : t \to s$ a pregroup derivation generated by cups and caps. We do not draw the wires for words and depict the dictionnary entries as triangles. For example, let $B = \{s,n\}$ and $D = \{\,one,\ two,\ three \to n,\ \ plus \to n^r \otimes n \otimes n^l,\ \ equals \to n^r \otimes s \otimes n^l\,\}$, then "one plus two equals three" is a grammatical sentence:*
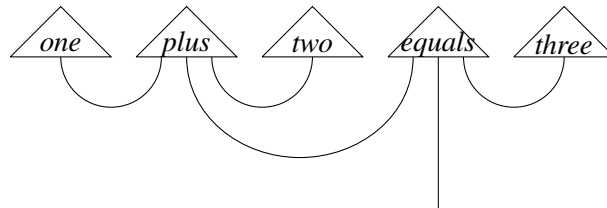
Example 3.3 *For any monoidal category $\mathbf{C}$, there is a free rigid category $A(\mathbf{C})$ with a fully-faithful monoidal functor $\mathbf{C} \hookrightarrow A(\mathbf{C})$, see [24]. Concretely, this means that in a rigid diagram with boxes coming from a monoidal category, if the domain and codomain have no adjoint types then all snakes can be removed. This allows to give a semantics to a pregroup grammar G as a rigid functor $F : \mathbf{G} \to A(\mathbf{C})$. For example, let one, two $: 1 \to n$ and plus $: n \times n \to n$ be functions, then we can compute the meaning of "one plus two":*
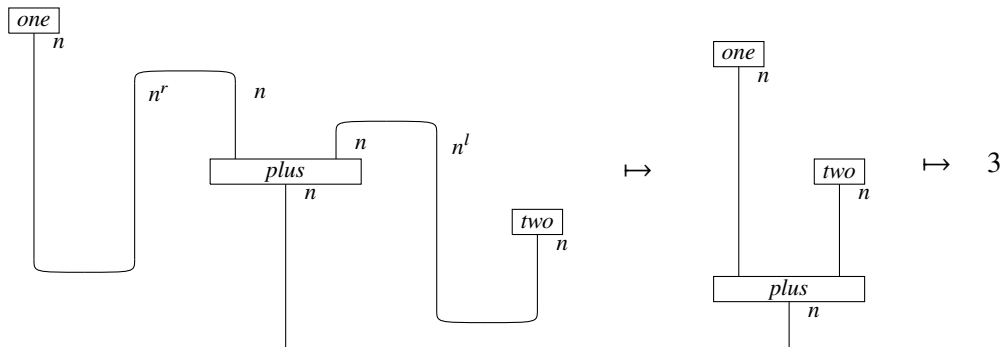
where the first step is snake removal and the second is function evaluation as in appendix A.

4   tensor.py

Let $\mathbf{Mat}_{\mathbb{S}}$ be the category with objects the natural numbers and arrows $m \to n$ the matrices $[n] \times [m] \to \mathbb{S}$ for a commutative semiring $\mathbb{S}$, with Kronecker product as tensor. $\mathbf{Mat}_{\mathbb{S}}$ is compact closed, i.e. both symmetric monoidal and rigid. It is furthermore self-dual, i.e. objects are isomorphic to their adjoints. For $\mathbb{S} = \mathbb{B}$, we get a category equivalent to finite sets and relations with Cartesian product. For $\mathbb{S} = \mathbb{C}$, it is equivalent to finite-dimensional complex vector spaces and linear maps with the usual tensor product. In practice, it is more convenient to consider an equivalent category $\mathbf{Tensor}_{\mathbb{S}}$ where objects are lists of natural numbers and adjoints are given by list reversal. Arrows $(m_1, \ldots, m_k) \to (n_1, \ldots, n_{k'})$ are tensors of order $k + k'$, i.e. matrices $m_1 \times \cdots \times m_k \to n_1 \times \cdots \times n_{k'}$.

class `Dim(n_0, ..., n_k)` is a subclass of `rigid.Ty` generated by natural numbers.

class `Tensor(dom, cod, array)` is a subclass of `rigid.Box` given by `Dim`-instances `dom` and `cod` and a numpy [69] `array` of the appropriate shape. `then` and `tensor` are both implemented using `numpy.tensordot`, `cups` and `caps` return a reshaped identity.

class `TensorFunctor(ob, ar)` is a subclass of `rigid.Functor` where `ob` and `ar` are mappings from `Ty` to `Dim` and from `Box` to `Tensor` respectively.

Remark 4.1 *All the methods of the* `Tensor` *class are writen in* `jax.numpy`*, the subset of Python+*`numpy` *that supports automatic differentiation with* `jax` *[31].*

Example 4.2 *Tensor networks can be defined as diagrams with a functor into tensors, contraction is given by functor application. They have been applied to both condensed matter physics and machine learning, see [54] for an introduction. Interfacing* DisCoPy *with tensor network tools such as [43, 33, 61] is left for future work.*

Example 4.3 *Relational databases can be defined as Boolean tensors: a table with k columns is a state $1 \to (n_1, \ldots, n_k)$ in* $\mathbf{Tensor}_{\mathbb{B}}$*. Conjunctive queries are diagrams, where query containment gives the structure of a free Cartesian bicategory, see [10]. Query evaluation over a relational database is the application a functor into Boolean tensors.*

Example 4.4 *The distributional compositional (DisCo) models of Coecke et al. [14, 15] can be defined as functors $F : \mathbf{G} \to \mathbf{Tensor}_{\mathbb{S}}$ from the rigid category $\mathbf{G}$ generated by a pregroup grammar (see example 3.2) into tensors, i.e. they map pregroup types $t \in \mathbf{G}$ to dimensions $F(t) \in \mathbb{N}^{\star}$ and dictionnary entries $w \to t$ to tensors of shape $F(t)$. When $F(s) = 1$, the meaning for a grammatical sentence $g : w_1 \ldots w_n \to s$ is a scalar $F(g) \in \mathbb{S}$ which can be computed as the contraction of a tensor network. DisCo models into real vector spaces, i.e. with $\mathbb{S} = \mathbb{R}$, received experimental support, see [32, 38, 37]. Relational DisCo models, i.e. with $\mathbb{S} = \mathbb{B}$, have been applied to question answering, see [18, 22].*

5   circuit.py

Quantum circuits are a standard model for quantum computation. They form the arrows of a PROP, i.e. a symmetric monoidal category generated by one object, called a qubit. We define $\mathbf{Circ}$ as the free PROP generated by n-qubit gates $g : n \to n$, scalars $\{s : 0 \to 0\}_{s \in \mathbb{C}}$, post-selection $\{\mathtt{bra}_i : 1 \to 0\}_{i \in \{0,1\}}$ and preparation $\{\mathtt{ket}_i : 0 \to 1\}_{i \in \{0,1\}}$ of ancilla qubits in the computational basis. Circuit evaluation is defined as a monoidal functor $\mathtt{eval} : \mathbf{Circ} \to \mathbf{Tensor}_{\mathbb{C}}$ which sends each gate $g : n \to n$ to its unitary matrix $\mathtt{eval}(g) : 2^n \to 2^n$.

Given the circuit for an *n*-qubit state $c : 0 \rightarrow n$, measurement results are a tensor $\mathtt{measure}(c) : 1 \rightarrow 2^n$ of non-negative reals in $\mathbf{Tensor}_{\mathbb{R}^+}$, computed using the Born rule. Note that if a circuit contains scalars or post selection, the measurement results need not be a normalised probability distribution.

The quotient $\mathbf{Circ}_\backsim$, where $c = c'$ iff $\mathtt{eval}(c) = \mathtt{eval}(c')$, is a compact-closed category. Cups and caps are given by the (unnormalised) Bell effect and state, the snake equation implies the correctness of the teleportation protocol. See [20] for an introduction to diagrammatic reasoning and quantum processes.

class `Circuit(dom, cod, boxes, offsets)` is a subclass of `rigid.Diagram` with `PRO`-instances as `dom` and `cod`. It has methods `eval`, implemented as a `TensorFunctor`, and `measure` which computes the Born rule.

class `Bra(b_0, ..., b_n)` and `Ket(b_0, ..., b_n)` are subclasses of `Circuit` and `rigid.Box` given by a bitstring `b_0, ..., b_n`.

class `Gate(name, n_qubits, array)` is a subclass of `Circuit` and `rigid.Box` with instances `H`, `CX`, `SWAP`, etc. Phases are implemented as subclasses `Rx` and `Rz`.

class `CircuitFunctor(ob, ar)` is a subclass of `rigid.Functor` where `ob` and `ar` are mappings from `Ty` to `PRO` and from `Box` to `Circuit` respectively.

The methods `to_tk` and `from_tk` translate back and forth between DisCoPy's `Circuit` class and that of t|ket⟩ [64], which can then be compiled and executed on quantum hardware or simplified using `pyzx` [40]. Note that in the translation from DisCoPy diagrams to the directed acyclic graphs of t|ket⟩, we treat the `SWAP` gate as a logical gate, i.e. it simply renames the two qubits. In the other direction, we introduce `SWAP` gates whenever a t|ket⟩ gate is applied to non-adjacent qubits. Thus, `from_tk(c.to_tk())` is equal to the original circuit `c` up to the axioms of symmetric monoidal categories.

Example 5.1 *The quantum algorithms for natural language processing (NLP) of [71] can be defined as rigid functors* $\mathbf{G} \rightarrow \mathbf{Circ}$ *from a pregroup grammar (see examples 3.2 and 4.4) to the category of circuits. See [17] for a discussion of distributional compositional models for NLP on quantum hardware. A proof-of-concept was implemented using* DisCoPy, *see the notebook of the first experiments here and there [52] for more details.*

References

[1] S. Abramsky (1996): *Retracing Some Paths in Process Algebra*. In: *CONCUR'96: Concurrency Theory*, *Lecture Notes in Computer Science* 1119, Springer, pp. 1–17, doi:10.1007/3-540-61604-7_44.

[2] Samson Abramsky & Bob Coecke (2008): *Categorical Quantum Mechanics*. arXiv:0808.1023 [quant-ph].

[3] Steve Awodey (2006): *Category Theory*. Ebsco Publishing, doi:10.1093/acprof:oso/9780198568612.001.0001.

[4] John C. Baez, Brandon Coya & Franciscus Rebro (2018): *Props in Network Theory*. arXiv:1707.08321 [math-ph].

[5] John C. Baez & Jason Erbele (2014): *Categories in Control*. arXiv:1405.6881 [quant-ph].

[6] John C. Baez & Brendan Fong (2015): *A Compositional Framework for Passive Linear Networks*.

[7] John C. Baez & Blake S. Pollard (2017): *A Compositional Framework for Reaction Networks*. *Reviews in Mathematical Physics* 29(09), p. 1750028, doi:10.1142/S0129055X17500283.

[8] Krzysztof Bar, Aleks Kissinger & Jamie Vicary: *Globular: An Online Proof Assistant for Higher-Dimensional Rewriting*. arXiv:1612.01093 [cs, math], doi:10.23638/LMCS-14(1:8)2018.

[9] Joe Bolt, Bob Coecke, Fabrizio Genovese, Martha Lewis, Dan Marsden & Robin Piedeleu (2017): *Interacting Conceptual Spaces I : Grammatical Composition of Concepts*. *CoRR* abs/1703.08314.

[10] Filippo Bonchi, Jens Seeber & Pawel Sobocinski (2018): *Graphical Conjunctive Queries*. In Dan Ghica & Achim Jung, editors: 27*th EACSL Annual Conference on Computer Science Logic* (*CSL* 2018), *Leibniz International Proceedings in Informatics* (*LIPIcs*) 119, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 13:1–13:23, doi:10.4230/LIPIcs.CSL.2018.13. Available at http://drops.dagstuhl.de/opus/volltexte/2018/9680.

[11] Filippo Bonchi, Paweł Sobociński & Fabio Zanasi (2014): *A Categorical Semantics of Signal Flow Graphs*. In Paolo Baldan & Daniele Gorla, editors: *CONCUR* 2014 – *Concurrency Theory*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 435–450, doi:10.1007/978-3-662-44584-6_30.

[12] C. Brown & G. Hutton (1994): *Categories, Allegories and Circuit Design*. In: *Proceedings of the* 9*th Annual IEEE Symposium on Logic in Computer Science* (*LICS*), IEEE Computer Society, pp. 372–381, doi:10.1109/LICS.1994.316052.

[13] Kenta Cho & Bart Jacobs (2019): *Disintegration and Bayesian Inversion via String Diagrams*. *Mathematical Structures in Computer Science* 29(7), pp. 938–971, doi:10.1017/S0960129518000488.

[14] Stephen Clark, Bob Coecke & Mehrnoosh Sadrzadeh (2008): *A Compositional Distributional Model of Meaning*. In: *Proceedings of the Second Symposium on Quantum Interaction* (*QI*-2008), pp. 133–140.

[15] Stephen Clark, Bob Coecke & Mehrnoosh Sadrzadeh (2010): *Mathematical Foundations for a Compositional Distributional Model of Meaning*. In J. van Benthem, M. Moortgat & W. Buszkowski, editors: *A Festschrift for Jim Lambek*, *Linguistic Analysis* 36, pp. 345–384.

[16] Bob Coecke (2005): *Kindergarten Quantum Mechanics*. arXiv:quant-ph/0510032, doi:10.1063/1.2158713.

[17] Bob Coecke (2019): *The Mathematics of Text Structure*.

[18] Bob Coecke, Giovanni de Felice, Dan Marsden & Alexis Toumi (2018): *Towards Compositional Distributional Discourse Analysis*. *Electronic Proceedings in Theoretical Computer Science* 283, pp. 1–12, doi:10.4204/EPTCS.283.1.

[19] Bob Coecke & Ross Duncan (2008): *Interacting Quantum Observables*. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir & Igor Walukiewicz, editors: *Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 298–310, doi:10.1007/978-3-540-70583-3_25.

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

[21] Bob Coecke & Robert W. Spekkens (2012): *Picturing Classical and Quantum Bayesian Inference*. *Synthese* 186(3), pp. 651–696, doi:10.1007/s11229-011-9917-5.

[22] Giovanni de Felice, Konstantinos Meichanetzidis & Alexis Toumi (2019): *Functorial Question Answering*. arXiv:1905.07408 [cs, math], doi:10.4204/EPTCS.323.6.

[23] Giovanni de Felice & Alexis Toumi: *Discopy 0.2.3 Documentation*. https://discopy.readthedocs.io/en/master/.

[24] Antonin Delpeuch (2014): *Autonomization of Monoidal Categories*. arXiv:1411.3827 [cs, math], doi:10.4204/EPTCS.323.3.

[25] Antonin Delpeuch & Jamie Vicary (2018): *Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm*. arXiv:1804.07832 [cs].

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

[27] François Foltz, Christian Lair & GM Kelly (1980): *Algebraic Categories with Few Monoidal Biclosed Structures or None*. *Journal of Pure and Applied Algebra* 17(2), pp. 171–177, doi:10.1016/0022-4049(80)90082-1.

[28] Brendan Fong & Michael Johnson (2019): *Lenses and Learners*. arXiv:1903.03671 [cs, math].

[29] Brendan Fong, David I. Spivak & Rémy Tuyéras (2017): *Backprop as Functor: A Compositional Perspective on Supervised Learning*. doi:10.1109/LICS.2019.8785665.

[30] Neil Ghani, Jules Hedges, Viktor Winschel & Philipp Zahn (2018): *Compositional Game Theory*. arXiv:1603.04641 [cs], doi:10.1145/3209108.3209165.

[31] google/jax (2020): *Composable Transformations of Python+NumPy Programs: Differentiate, Vectorize, JIT to GPU/TPU, and More*. https://github.com/google/jax.

[32] Edward Grefenstette & Mehrnoosh Sadrzadeh (2011): *Experimental Support for a Categorical Compositional Distributional Model of Meaning*. In: *The 2014 Conference on Empirical Methods on Natural Language Processing.*, pp. 1394–1404.

[33] Johannes Hauschild & Frank Pollmann (2018): *Efficient Numerical Simulations with Tensor Networks: Tensor Network Python (TeNPy)*. *SciPost Physics Lecture Notes*, p. 5, doi:10.21468/SciPostPhysLectNotes.5.

[34] Günter Hotz (1965): *Eine Algebraisierung Des Syntheseproblems von Schaltkreisen I*. *Elektronische Informationsverarbeitung und Kybernetik* 1, pp. 185–205.

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

[36] André Joyal & Ross Street (1991): *The Geometry of Tensor Calculus, I*. *Advances in Mathematics* 88(1), pp. 55–112, doi:10.1016/0001-8708(91)90003-P.

[37] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh & Stephen Pulman (2013): *Separating Disambiguation from Composition in Distributional Semantics*, p. 10.

[38] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh & Stephen G. Pulman (2012): *A Unified Sentence Space for Categorical Distributional-Compositional Semantics: Theory and Experiments*. In: *COLING*.

[39] Aleks Kissinger & Sander Uijlen (2019): *A Categorical Semantics for Causal Structure*. arXiv:1701.04732 [math-ph, physics:quant-ph], doi:10.23638/LMCS-15(3:15)2019.

[40] Aleks Kissinger & John van de Wetering (2019): *PyZX: Large Scale Automated Diagrammatic Reasoning*. arXiv:1904.04735 [quant-ph], doi:10.4204/EPTCS.318.14.

[41] Aleks Kissinger & Vladimir Zamdzhiev (2015): *Quantomatic: A Proof Assistant for Diagrammatic Reasoning*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, Springer International Publishing, pp. 326–336, doi:10.1007/978-3-319-21401-6_22.

[42] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay et al. (2016): *Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows*. In: *ELPUB*, pp. 87–90, doi:10.3233/978-1-61499-649-1-87.

[43] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar & Maja Pantic (2018): *TensorLy: Tensor Learning in Python*. arXiv:1610.09555 [cs].

[44] Stephen Lack (2004): *Composing PROPs*. *Theory and Applications of Categories* [*electronic only*] 13, pp. 147–163.

[45] Yves Lafont (2003): *Towards an Algebraic Theory of Boolean Circuits*. *Journal of Pure and Applied Algebra* 184(2-3), pp. 257–310, doi:10.1016/S0022-4049(03)00069-0.

[46] Joachim Lambek (1999): *Type Grammar Revisited*. In Alain Lecomte, François Lamarche & Guy Perrier, editors: *Logical Aspects of Computational Linguistics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–27, doi:10.1007/3-540-48975-4_1.

[47] Joachim Lambek (2001): *Type Grammars as Pregroups*. *Grammars* 4, pp. 21–39, doi:10.1023/A:1011444711686.

[48] Joachim Lambek (2008): *From Word to Sentence: A Computational Algebraic Approach to Grammar*. Open Access Publications, Polimetrica.

[49] S.M. Lane (1998): *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Springer New York, doi:10.1007/978-1-4612-9839-7.

[50] F. William Lawvere (1963): *Functorial Semantics of Algebraic Theories*. *Proceedings of the National Academy of Sciences of the United States of America* 50(5), pp. 869–872, doi:10.1073/pnas.50.5.869.

[51] matplotlib (2020): *Plotting with Python*. https://github.com/matplotlib/matplotlib.

[52] Konstantinos Meichanetzidis (2020): *Quantum Natural Language Processing*. https://medium.com/cambridge-quantum-computing/quantum-natural-language-processing-748d6f27b31d.

[53] networkx (2020): *Python Software for Complex Networks*. https://github.com/networkx/networkx.

[54] Roman Orus (2014): *A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States*. Annals of Physics 349, pp. 117–158, doi:10.1016/j.aop.2014.06.013.

[55] Evan Patterson (2017): *Knowledge Representation in Bicategories of Relations*. arXiv:1706.00526 [cs, math].

[56] Roger Penrose (1971): *Applications of Negative Dimensional Tensors*. Scribd.

[57] John Power & Edmund Robinson (1997): *Premonoidal Categories and Notions of Computation*. Mathematical Structures in Computer Science 7(5), pp. 453–468, doi:10.1017/S0960129597002375.

[58] Anne Preller & Joachim Lambek (2007): *Free Compact 2-Categories*. Mathematical Structures in Computer Science 17(2), pp. 309–340, doi:10.1017/S0960129506005901.

[59] David Reutter & Jamie Vicary (2019): *High-Level Methods for Homotopy Construction in Associative n-Categories*. arXiv:1902.03831 [math], doi:10.1109/LICS.2019.8785895.

[60] Mitchell Riley (2018): *Categories of Optics*. arXiv:1809.00738 [math].

[61] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal & Stefan Leichenauer (2019): *TensorNetwork: A Library for Physics and Machine Learning*. arXiv:1905.01330 [cond-mat, physics:hep-th, physics:physics, stat].

[62] P. Selinger (2010): *A Survey of Graphical Languages for Monoidal Categories*. New Structures for Physics, pp. 289–355, doi:10.1007/978-3-642-12821-9_4.

[63] Dan Shiebler, Alexis Toumi & Mehrnoosh Sadrzadeh (2020): *Incremental Monoidal Grammars*. arXiv:2001.02296 [cs].

[64] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington & Ross Duncan (2020): *Tket : A Retargetable Compiler for NISQ Devices*. arXiv:2003.10611 [quant-ph].

[65] Paweł Sobociński, Paul W. Wilson & Fabio Zanasi (2019): *CARTOGRAPHER: A Tool for String Diagrammatic Reasoning*. In: CALCO 2019, 139, pp. 20:1–20:7, doi:10.4230/LIPIcs.CALCO.2019.20.

[66] Statebox (2020): *Exchange Format for Morphisms in Monoidal Categories*. https://github.com/statebox/monmor-spec.

[67] Ross Street (1996): *Categorical Structures*. Handbook of algebra 1, pp. 529–577.

[68] Till Tantau (2013): *Graph Drawing in TikZ*. In Walter Didimo & Maurizio Patrignani, editors: Graph Drawing, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 517–528, doi:10.1007/978-3-642-36763-2_46.

[69] Stefan van der Walt, S. Chris Colbert & Gael Varoquaux (2011): *The NumPy Array: A Structure for Efficient Numerical Computation*. Computing in Science Engineering 13(2), pp. 22–30, doi:10.1109/MCSE.2011.37.

[70] Simon Wadsley & Nick Woods (2015): *PROPs for Linear Systems*. arXiv:1505.00048 [math].

[71] William Zeng & Bob Coecke (2016): *Quantum Algorithms for Compositional Natural Language Processing*. Electronic Proceedings in Theoretical Computer Science 221, pp. 67–75, doi:10.4204/EPTCS.221.8.
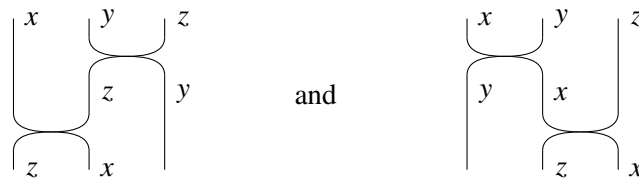
A   cartesian.py

This appendix describes `cartesian.Diagram` and `PythonFunctor`, an implementation of *Lawvere theories* and their models. We first give a short introduction to symmetric monoidal categories (SMC), PROPs and functorial semantics.

A (strict) monoidal category **C** is *symmetric* when it comes equipped with a natural transformation $\sigma_{x,y} : x \otimes y \to y \otimes x$ such that $\sigma_{y,x} \circ \sigma_{x,y} = 1_{x \otimes y}$ (involution) and $\sigma_{x,y \otimes z} = (1_z \otimes \sigma_{x,z}) \circ (\sigma_{x,y} \otimes 1_z)$

(hexagon). A component $\sigma_{x,y}$ is depicted as a swap of the wires for x and y. The free symmetric monoidal category $\mathbf{SMC}(\Sigma)$ generated by a monoidal signature Σ can be defined as a quotient $\mathbf{MC}(\Sigma')/\mathcal{S}$ of the free monoidal category generated by the disjoint union $\Sigma' = \Sigma + \{\sigma_{x,y}\}_{x,y \in \Sigma_0}$. The relation $\mathcal{S}$ is generated by the rules for involution and naturality:
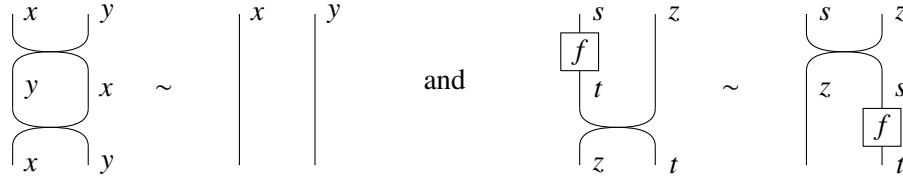
for all $x, y, z \in \Sigma_0$ and $f : s \to t$ in Σ'. Note that $s, t \in \Sigma_0^\star$ may be of arbitrary length, in which case $\sigma_{s,z}$ and $\sigma_{t,z}$ are defined as ladders of swaps, i.e. the symmetry for compound types $\sigma_{x \otimes y, z}$ and $\sigma_{x, y \otimes z}$ is defined inductively by:

The symmetry for the empty type is defined to be the identity $\sigma_{x,1} = \sigma_{1,x} = 1_x$. An SMC where the tensor is the Cartesian product and the unit is terminal (i.e. a category with finite products) is called a *Cartesian category*. Equivalently, an SMC is Cartesian when objects carry a natural commutative comonoid structure [62, 6.1]. Given a monoidal signature Σ, the free Cartesian category $\mathbf{CC}(\Sigma)$ is the quotient $\mathbf{MC}(\Sigma'')/(\mathcal{S} + \mathcal{P})$ for $\Sigma'' = \Sigma' + \{\mu_x : x \to x \otimes x\}_{x \in \Sigma_0} + \{\epsilon_x : x \to 1\}_{x \in \Sigma_0}$. The components μ_x and ϵ_x are depicted as wire splitting and ending respectively. The comonoids of non-atomic types inductively. That for the unit is the identity and the comonoid of $x \otimes y$ is given by:
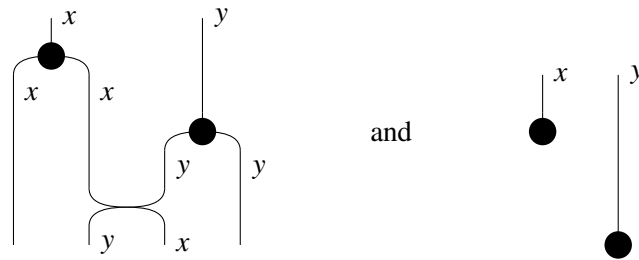
The relation $(\mathcal{S} + \mathcal{P})$ is given by the axioms for commutative comonoids plus naturality of symmetry, coproduct and counit for each generating arrow. A *PROP* (PROduct and Permutation) [44] is an SMC generated by one object, a *Lawvere theory* [50] is a Cartesian category generated one object.

Lawvere theories are implemented by the class `cartesian.Diagram`, a subclass of `monoidal.Diagram` with static methods `swap`, `copy` and `delete` implementing the structural morphisms. `cartesian.Box` is a subclass of `monoidal.Box` where each instance holds a Python function with natural numbers as domain and codomain. A Cartesian box `f` with `f.dom, f.cod == (m, n)` sends m-tuples to n-tuples, it can be defined in the standard syntax for Python functions using the decorator `@disco(m, n)`. `Swap`, `Copy` and `Del` are subclasses of `cartesian.Box` which implement the symmetry and comonoid on the generating object. `cartesian.Functor` is a subclass of `monoidal.Functor` which preserves symmetry and product. `Function` is a subclass of

`cartesian.Box` where `then` and `tensor` are overriden by function composition and tuple concate-nation. The `PythonFunctor` class implements Cartesian functors into `Function`, i.e. it maps the formal composition of diagrams to the concrete composition of functions. Note that when `f` and `g` have side-effects, the tensor `f @ g` is in general different from `Id(f.dom) @ g » f @ Id(g.cod)`. Thus, `Function` is closer to the implementation of a premonoidal than a monoidal category, see example 2.2.

Example A.1 *The Lawvere theory $\mathbf{F}$ with no generating arrows is the opposite of the category of finite sets with disjoint union as monoidal structure: diagrams $f : m \to n$ in $\mathbf{F}$ correspond precisely to the graphs of the functions $f : [n] \to [m]$. The subcategories of diagrams in $\mathbf{F}$ with no coproduct, no counit and no symmetry correspond to injective, surjective and monotone functions respectively. The subcategory of $\mathbf{F}$ generated by symmetry alone, i.e. the free SMC generated by one object, corresponds to bijections. The normal form for diagrams in $\mathbf{F}$ is given by decomposing any function into a surjection and a monotone injection, see [45, Theorem 3].*

Example A.2 *For a semiring $\mathbb{S}$, the category of matrices over $\mathbb{S}$ with direct sum as monoidal product can be defined as a Lawvere theory generated by a commutative monoid $+ : 2 \to 1$ with unit $e : 0 \to 1$ and scalars $\{s : 1 \to 1\}_{s \in \mathbb{S}}$. The relations are given by the axioms for semirings, see [70]. The naturality relation for the comonoid with respect to $+$ and e are called the bialgebra laws, they have applications in control and network theory [5, 4].*

Example A.3 *Let $\mathbf{NN}$ be the Lawvere theory generated by sum $+ : 2 \to 1$, activation $a : 1 \to 1$, finite sets of weights $\{w_i : 1 \to 1\}_{i \in W}$ and biases $\{b_i : 0 \to 1\}_{i \in B}$. The arrows of $\mathbf{NN}$ are the diagrams of neural network architectures. Given a set of parameters $\theta : W + B \to \mathbb{R}$, an implementation is given by the model $F_\theta : \mathbf{NN} \to \mathbf{Set}$ such that $F_\theta(1) = \mathbb{R}$, $F_\theta(b_i) = \{() \mapsto \theta(b_i)\}$, $F_\theta(w_i) = \{x \mapsto \theta(w_i) \cdot x\}$, $F_\theta(+) = \{(x, y) \mapsto x + y\}$ and $F_\theta(a) : \mathbb{R} \to \mathbb{R}$ is a non-linearity such as sigmoid. A loss $l : (\mathbb{R}^m \to \mathbb{R}^n) \to \mathbb{R}$ for an architecture $f : m \to n$ in $\mathbf{NN}$ takes an implementation and returns a real number encoding its success at some data-driven task. The gradient of $\theta \mapsto l(F_\theta(f))$ can be computed by back-propagation over the architecture f, using automatic differentiation tools such as `jax` [31]. The back-propagation algorithm is itself part of a functor $\mathbf{NN} \to \mathbf{Learn}$, where $\mathbf{Learn}$ is a Cartesian category of supervised learning algorithms, see [29, 28].*

B drawing.py

We reformulate some of the definitions and results from Joyal, Street [35].

Definition B.1 *A* topological graph*, also called 1d cell complex, is a tuple $(\Gamma, \Gamma_0, \Gamma_1)$ of a Hausdorff space Γ and a closed, discrete subset $\Gamma_0 \subseteq \Gamma$ of nodes such that $\Gamma - \Gamma_0 = \bigsqcup \Gamma_1$ is the sum of its connected components Γ_1, called edges, each homeomorphic to an open intervals with boundary in Γ_0.*

Definition B.2 *A planar graph between two real numbers $a < b$ is a finite topological graph Γ embedded in $\mathbb{R} \times [a, b]$ such that every $x \in \Gamma \cap (\mathbb{R} \times \{a, b\})$ is a node in Γ_0 and belongs to the closure of exactly one edge in Γ_1.*

The points in $\Gamma_0 \cap (\mathbb{R} \times \{a, b\})$ are called outer nodes, they are the boundary of the domain and codomain $\mathsf{dom}(\Gamma), \mathsf{cod}(\Gamma) \in \Gamma_1^\star$ of the planar graph Γ. The points $f \in \Gamma_0 - (\mathbb{R} \times \{a, b\})$ are called inner nodes, they have a domain and codomain $\mathsf{dom}(f), \mathsf{cod}(f) \in \Gamma_1^\star$ given by the edges that have f as boundary. The composition $\Gamma \circ \Gamma'$ and tensor $\Gamma \otimes \Gamma'$ are defined by rescaling and pasting the two planar graphs Γ, Γ' vertically and horizontally respectively, see [35, §4].

A planar graph Γ is *progressive*, or recumbent, when the second projection $e \to [a,b]$ is injective for every edge $e \in \Gamma_1$. Progressive planar graphs have no cups or caps. A progressive planar graph is *generic* when the projection $\Gamma_0 - (\mathbb{R} \times \{a,b\}) \to (a,b)$ is injective, i.e. there are no two inner nodes at the same height. A deformation of planar graphs is a continuous map $h : \Gamma \times [0,1] \to [a,b] \times \mathbb{R}$ such that 1) for all $t \in [0,1]$, $h(-,t)$ is an embedding whose image is a planar graph, and 2) for all $x \in \Gamma_0$, if $h(x,t)$ is inner for some t then it is inner for all values of $t \in [0,1]$. A deformation of planar graphs is progressive (generic) when for all $t \in [0,1]$, $h(-,t)$ is progressive (generic).

The induced equivalence relation — with $\Gamma_0 \sim \Gamma_1$ if and only if there is a deformation h with $h(-,0) = \Gamma_0$ and $h(-,1) = \Gamma_1$ — is a congruence with respect to composition and tensor of planar graphs, i.e. if $\Gamma_0 \sim \Gamma_1$ and $\Omega_0 \sim \Omega_1$ then $\Gamma_0 \otimes \Omega_0 \sim \Gamma_1 \otimes \Omega_1$ and $\Gamma_0 \circ \Omega_0 \sim \Gamma_1 \circ \Omega_1$. Furthermore, tensor and composition are associative and they respect the interchange law up to progessive deformation. Thus, progessive planar graphs up to progessive deformation form a strict monoidal category [35, Proposition 4].

Given a monoidal signature Σ, a *valuation* v of a planar graph Γ is a pair of functions $v_0 : \Gamma_1 \to \Sigma_0$ and $v_1 : \Gamma_0 - (\mathbb{R} \times \{a,b\}) \to \Sigma_1$ that send edges to objects and inner nodes to arrows, which commute with domain and codomain. Progressive planar graphs valued in Σ (up to progressive deformation) are the free monoidal category [35, Theorem 5]. We conjecture that generic planar graphs up to generic deformation are the free premonoidal category.

From the universal property, we know that the category of progressive planar graph is equivalent to the combinatorial definition of free monoidal categories given in section 2. Concretely, this equivalence is witnessed by the following pair of algorithms translating between planar graphs and diagrams.

Input : A progressive planar graph Γ
Output `Diagram(dom, cod, boxes, offsets)`
:
1 Compute the connected components Γ_1 and their boundaries Γ_0.
2 Order the nodes by height and partition them into $\Gamma_0 = $ `dom` $+$ `boxes` $+$ `cod`.
3 Find the start and end points $\Gamma_1 \to ($ `dom` $+$ `boxes` $) \times ($ `boxes` $+$ `cod` $)$ for each edge, the
     preimage of this map gives the domain and codomain for each box.
4 Compute `offsets` as the number of edges to the left of each box.

Algorithm 1: read

Input : `Diagram(dom, cod, boxes, offsets)`
Output A progressive planar graph Γ
:
1 Set $\Gamma_0 := \{ (i,0) \mid i < $ `len(dom)` $\}$ and $\Gamma_1 := \emptyset$.
2 **for** `height, (box, offset) in enumerate(zip(boxes, offsets))` **do**
3 $\quad$ Deform Γ so that there is at least `len(box.cod) + 1` horizontal space between the edges
     $\quad$ of $\mathsf{cod}(\Gamma)$ at index `offset` and `offset + len(box.dom)`.
4 $\quad$ Set $\Gamma_0 := \Gamma_0 + \{$ `box` $\}$ for `box` $= (m, $ `height` $+ \frac{1}{2})$ with m computed at step 3 and
     $\quad$ $\Gamma_1 := \Gamma_1 + \{x \to $ `box` $\mid x \in$ `box.dom` $\} + \{$ `box` $\to x \mid x \in$ `box.cod` $\}$.
5 Return $\Gamma := \Gamma + \{(i,j) \to (i, $ `max(len(boxes), 1)` $) \mid (i,j) \in \mathsf{cod}(\Gamma)\}$.

Algorithm 2: draw