DisCoPy: the Hierarchy of Graphical Languages in Python

Alexis Toumi Richie Yeung Boldizsár Poór Giovanni de Felice
Quantinuum – Quantum Compositional Intelligence
17 Beaumont street, OX1 2NA Oxford, UK
`firstname@discopy.org`

DisCoPy is free software with extensive documentation and demonstration notebooks available at:

`https://docs.discopy.org`

The library is already the topic of a tool paper [5] aimed at applied category theorists and one [16] aimed at quantum computer scientists. It is also documented by the PhD theses of the last and first authors of this report. The former [4] develops a framework for natural language processing based on categories for syntax and functors for semantics. The latter [15] applies this to *quantum natural language processing* (QNLP) defined in terms of functors from grammar to parameterised quantum circuits which can then be trained using *diagrammatic differentiation* [17]. The application of DisCoPy to QNLP has now grown into its own library, lambeq [12].

DisCoPy aims at becoming the fundamental package for all the applications of string diagrams. The use of Python for applied category theory (ACT) is motivated by two main reasons. First, it has become the programming language of reference for machine learning and quantum computing, two killer applications of ACT. Second, Python is a programming language of choice for students and beginners. We believe that DisCoPy can help both applied category theorists pick up programming skills and Python programmers pick up category theory concepts.

Talk Outline: We will start with the core features of DisCoPy: how to build monoidal diagrams and transform them using monoidal functors. This functorial transformation from a syntactic to a semantic category can be used to perform computation, for example for Quantum Natural Language Processing. In particular, we will highlight the integration with the PyTorch, JAX and tensornetwork libraries for machine learning. Then we will present the recent developments in DisCoPy v1.0. First we will give a high level overview of how the package is organised in this latest version, which effectively implements the hierarchy of graphical languages for monoidal categories as described in Selinger's survey [13] (See Figure 2a for summary). Then we introduce the new data structure for hypergraph diagrams and contrast it with the existing data structure for planar diagrams. The word problem complexity of different monoidal categories will also be discussed.

1 Planar diagrams

DisCoPy implements free categories and functors in Python — that is, there are no additional rules beyond the axioms that relate the morphisms of the category. Each type of category is defined in its own module. The generating objects and morphisms in the base category `cat` are represented by the classes `Ob(name: str)` and `Box(name: str, dom: Ob, cod: Ob)` respectively. Composite morphisms can be built using the `>>` binary operator, which returns an `Arrow(inside: tuple[Arrow,...], dom: Ob, cod: Ob)` object. Identity morphisms are represented by the class `Id(dom: Ob)`.

The `monoidal` module is where planar diagrams are implemented. `Ty(inside: tuple[Ob,...])` is a subclass of `Ob` with a method `Ty.tensor(self, *others: Ty) -> Ty` for concatenation shortened to the binary operator `@` with the empty type `Ty()` as unit, i.e. `Ty` is the free monoid over `Ob`.

Submitted to:
ACT 2023

`Diagram(inside: tuple[Layer,...], dom: Ty, cod: Ty)` is a subclass of `Arrow` with layers as boxes. The method `Diagram.normal_form` solves the word problem for (connected) planar diagrams [7] by repeatedly applying *interchanger* rewrites from `Id(f.dom) @ g >> f @ Id(g.cod)` to `f @ Id(g.dom) >> Id(f.cod) @ g`.

2 Functors

A `Functor` in DisCoPy transforms a diagram into another diagram in a way that respects functoriality, the monoidal structure and the additional axioms of the source category. When the target category is chosen to be `python`, `matrix` or `tensor`, the functor can be used to compute the 'semantics' of the diagram. In practice, one may wish to first convert to an intermediate category, such as `tk` or `zx`, to optimise the diagram before actually performing the computation, and there is a tensor network simulator in DisCoPy that computes the same functorial semantics but composes the tensor boxes in a different, more efficient order. Concretely, `Functor(ob: Map, ar: Map, cod: Category)` is given by an optional codomain and a pair of mappings `Map = dict | Callable`: one from `Ob` to `cod.ob` and one from `Box` to `cod.ar`. By default, the codomain is the free `Category(Ob, Arrow)`. The domain is defined implicitly by the domain of `ob` and `ar`. Functors also have their own methods `id` and `then` so we can define `CAT = Category(Category, Functor)`.

3 Hypergraphs diagrams

The encoding of string diagrams as lists of layers makes it possible to draw them and evaluate them simply with a `for` loop. However this comes at the cost of representing swaps explicitly as generating morphisms subject to naturality conditions. Alternatively, string diagrams for symmetric categories can be encoded as *discrete cospans of hypergraphs* [3] where the equations for symmetric, traced, compact and hypergraph categories all come for free. This is implemented in DisCoPy's `hypergraph` module.

The class `Hypergraph(dom, cod, boxes, wires, spider_types)` is defined by:

- a pair of types `dom: Ty`, `cod: Ty` and a list of `boxes: tuple[Box, ...]`,
- a mapping `spider_types: dict[Any, Ty]` where the set of keys are called spiders,
- a list of `wires: tuple[Any, ...]` that encodes a mapping from input and output ports to spiders, i.e. with `set(wires) <= set(spider_types) and len(wires) == len(ports)` where:

  ```
  ports = dom.tensor(*[box.dom @ box.cod for box in boxes]) @ cod
  ```

  and such that `all(spider_types[wires[i]] == x for i, x in enumerate(ports))`.

The method `Hypergraph.tensor` concatenates the attributes of two hypergraphs then reorders the wires. The composition `Hypergraph.then` computes the push-out of cospans via reflexive transitive closure. The three methods `Hypergraph.id`, `swap` and `spiders` generate all the hypergraphs with no boxes. `Hypergraph.from_box` wraps up a box as a hypergraph while `to_diagram` represents the hypergraph as a planar `Diagram`, with explicit `Swap` and `Spider` boxes from the `frobenius` module. Indeed, spiders are also known as special commutative Frobenius algebras. The inverse translation `Diagram.to_hypergraph` is a functor with `cod=Category(Ty, Hypergraph)`. Two hypergraphs are equal when their attributes are equal up to a permutation of the boxes and spiders, this is computed by reduction to the graph isomorphism algorithm of NetworkX [10]. Hypergraphs are the arrows of *free hypergraph categories*, i.e. symmetric categories with a supply [8] of spiders.

A powerful new feature built on top of the `Hypergraph` class allows to construct string diagrams using the standard syntax for Python functions, a form of *substructural type system* which is illustrated in Figure 1a.

Future work

Plans for future developments include the implementation of free bicategories in terms of diagrams with colours, as well as double categories where wires can go horizontally. Another promising direction is the implementation of *double-pushout rewriting* via interfaces to existing libraries [1, 14, 11]. Diagram rewriting can then itself be represented in terms of *higher-dimensional diagrams*, with a generalisation of our list-based `Diagram` data structure [2] or with *combinatorial directed cell complexes* [9] which generalise our graph-based `Hypergraph` data structure.

References

[1] Jakob L. Andersen, Christoph Flamm, Daniel Merkle & Peter F. Stadler (2016): *A Software Package for Chemically Inspired Graph Transformation*. In Rachid Echahed & Mark Minas, editors: *Graph Transformation*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 73–88, doi:10.1007/978-3-319-40530-8_5.

[2] Krzysztof Bar & Jamie Vicary (2017): *Data Structures for Quasistrict Higher Categories*. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pp. 1–12, doi:10.1109/LICS.2017.8005147.

[3] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2022): *String Diagram Rewrite Theory I: Rewriting with Frobenius Structure*. *Journal of the ACM* 69(2), pp. 14:1–14:58, doi:10.1145/3502719. arXiv:2012.01847.

[4] Giovanni de Felice (2022): *Categorical Tools for Natural Language Processing*. Ph.D. thesis, University of Oxford. arXiv:2212.06636.

[5] Giovanni de Felice, Alexis Toumi & Bob Coecke (2020): *DisCoPy: Monoidal Categories in Python*. In: *Proceedings of the 3rd Annual International Applied Category Theory Conference, ACT*, 333, EPTCS, doi:10.4204/EPTCS.333.13.

[6] Antonin Delpeuch & Jamie Vicary (2021): *The Word Problem for Braided Monoidal Categories Is Unknot-Hard*. *ArXiv e-prints*. arXiv:2105.04237.

[7] Antonin Delpeuch & Jamie Vicary (2022): *Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm*. *Logical Methods in Computer Science* 18(1), doi:10.46298/lmcs-18(1:10)2022.

[8] Brendan Fong & David I. Spivak (2020): *Supplying Bells and Whistles in Symmetric Monoidal Categories*, doi:10.48550/arXiv.1908.02633. arXiv:1908.02633.

[9] Amar Hadzihasanovic & Diana Kessler (2022): *Data Structures for Topologically Sound Higher-Dimensional Diagram Rewriting*, doi:10.48550/arXiv.2209.09509. arXiv:2209.09509.

[10] Aric Hagberg, Pieter Swart & Daniel S Chult (2008): *Exploring Network Structure, Dynamics, and Function Using Networkx*. Technical Report LA-UR-08-05495; LA-UR-08-5495, Los Alamos National Lab. (LANL), Los Alamos, NM (United States).

[11] Russ Harmer & Eugenia Oshurko (2020): *Knowledge Representation and Update in Hierarchies of Graphs*. *Journal of Logical and Algebraic Methods in Programming* 114, p. 100559, doi:10.1016/j.jlamp.2020.100559.

[12] Dimitri Kartsaklis, Ian Fan, Richie Yeung, Anna Pearson, Robin Lorenz, Alexis Toumi, Giovanni de Felice, Konstantinos Meichanetzidis, Stephen Clark & Bob Coecke (2021): *Lambeq: An Efficient High-Level Python Library for Quantum NLP*. *ArXiv e-prints*. arXiv:2110.04236.

[13] Peter 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.

[14] 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.

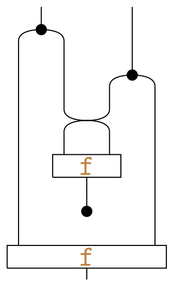
[15] Alexis Toumi (2022): *Category Theory for Quantum Natural Language Processing*. Ph.D. thesis, University of Oxford. arXiv:2212.06615.

[16] Alexis Toumi, Giovanni de Felice & Richie Yeung (2022): *DisCoPy for the Quantum Computer Scientist*. *QPL*. arXiv:2205.05190.

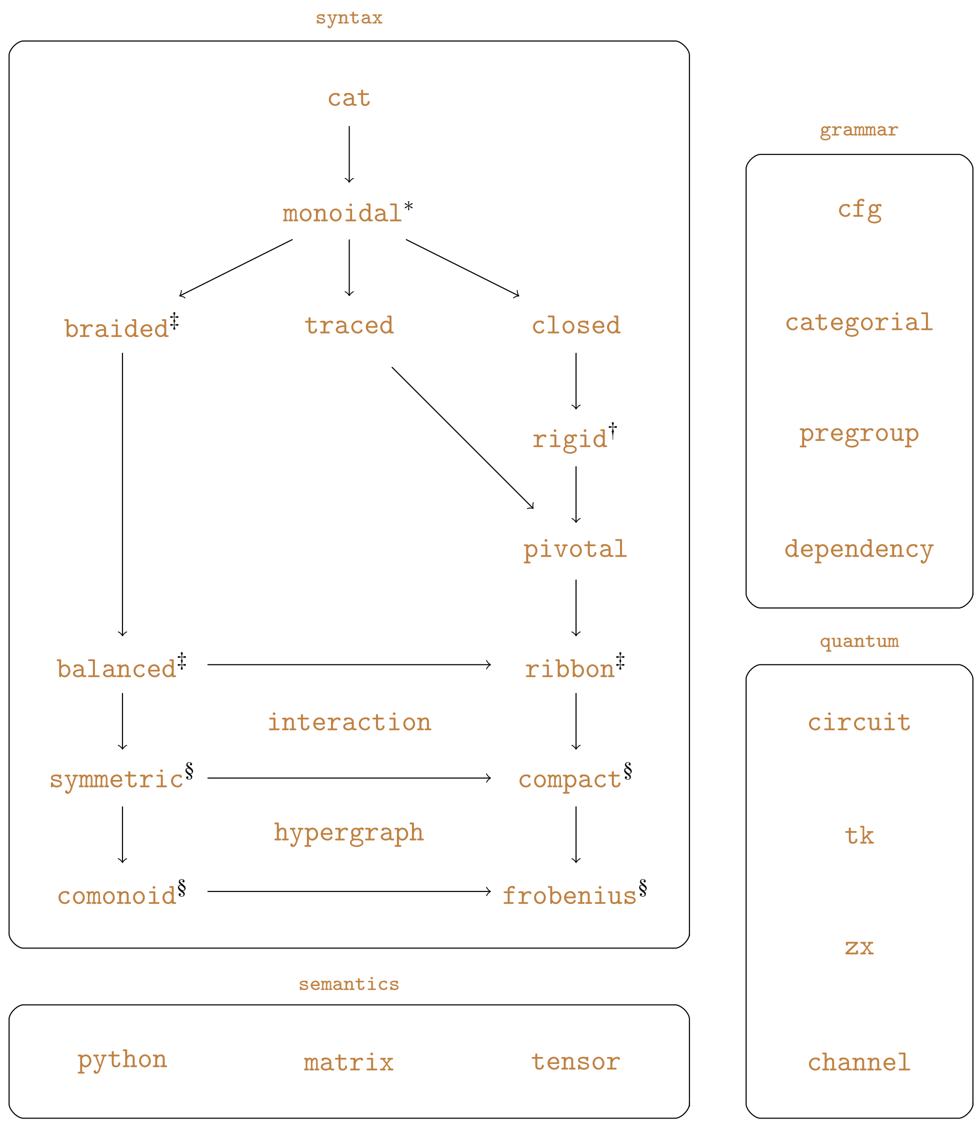
[17] Alexis Toumi, Richie Yeung & Giovanni de Felice (2021): *Diagrammatic Differentiation for Quantum Machine Learning*. *QPL*. arXiv:2103.07960.

(a) Defining a string diagram and applying a functor with DisCoPy.

```
x, y = map(Ty, "xy"); f = Box('f', x @ x, y)

@Diagram.from_callable(x @ x, y)
def diagram(a, b): c = f(b, a); return f(a, b)

assert diagram == Copy(x) @ Copy(x)\
    >> x @ (Swap(x, x) >> f >> Copy(y, n=0)) @ x >> f

F = python.Functor(ob={x: int, y: bool}, ar={f: lambda a, b: a < b})
assert F(diagram)(42, 43); diagram.draw(to_tikz=True)
```



(a) Architecture of the DisCoPy library.



* Planar diagram equivalence takes quadratic time and normalisation is cubic time in the worst case [7].
† Rigid diagram normal form is computed in polynomial time with *snake removal*, see [15, §1.4.1].
‡ Braided diagram equivalence is unknot hard [6], it is currently not implemented in DisCoPy.
§ Hypergraph diagram equivalence reduces to graph isomorphism which is implemented with NetworkX [10].