Category Theory for Quantum Natural Language Processing

Alexis TOUMI

Wolfson College

University of Oxford

A (draft of a) thesis (to be) submitted for the degree of

Doctor of Philosophy

January 13, 2022

Contents

1

DisCoPy: monoidal categories in Python

Python has become the programming language of choice for most applications in both natural language processing (e.g. Stanford NLP [Man+14], NLTK [LB02] and SpaCy [HM17]) and quantum computing (with development kits like Qiskit [Cro18] and PennyLane [Ber+20] and interfaces to compilers like pytket [Siv+20]). Thus, it was the obvious choice of language for an implementation of QNLP. However, unlike functional programming languages like Haskell, Python has little support for category theory. Indeed, before the release of DisCoPy, the only existing Python framework for category theory was a module of SymPy [Meu+17] that can draw commutative diagrams in finite categories. Hence, the first step in implementing QNLP was to develop our own framework for applied category theory in Python: DisCoPy. The main feature was the drawing of string diagrams (e.g. the grammatical structure of sentences) and the application of functors (e.g. to quantum circuits, either executed on quantum hardware or classically simulated).

String diagrams have become the lingua franca of applied category theory. However, the definitions one can find in the literature usually fall into one of two extremes: either definitions by general abstract nonsense or definitions by example and appeal to intuition. On one side of the spectrum, the standard technical reference has become the *Geometry of tensor calculus* [JS91] where Joyal and Street define string diagrams as equivalence classes of labeled topological graphs embedded in the plane and then characterise them as the arrows of free monoidal categories. On the other, *Picturing quantum processes* [CK17] contains over a thousand string diagrams but their formal definition as well as any mention of category theory are relegated to mere appendices.

This chapter contains a description of the DisCoPy package alongside an elementary list-based definition of string diagrams. The first section introduces categories and functors for the Python programmer, i.e. with no mathematical prerequisites. The second section introduces monoidal categories for the Python programmer, defining string diagrams from first principles. The third section gives

the category theoretic foundations for our definition, which we call the premonoidal approach. The fourth section defines the drawing and reading algorithms for string diagrams, which arise as the two sides of the equivalence between the premonoidal and the topological definitions. The fifth section introduces monoidal categories with extra structure (rigid, biclosed, symmetric, cartesian, hypergraph) and the inheritence mechanism which implements this hierarchy of structure. The last section discusses the relationship between our list-based premonoidal approach and the exisiting graph-based definitions of diagrams in symmetric monoidal categories.

1.1 Categories in Python

What are categories and how can they be useful to the Python programmer? This section will answer this question by taking the standard mathematical definitions and breaking them into *data*, which can be translated into Python code, and *axioms*, which cannot be formally verified in Python, but can be translated into test cases. The data for a category is given by a tuple $C = (C_0, C_1, \texttt{dom}, \texttt{cod}, \texttt{id}, \texttt{then})$ where:

- C_0 and C_1 are classes of *objects* and *arrows* respectively,
- $\texttt{dom}, \texttt{cod} : C_1 \to C_0$ are functions called *domain* and *codomain*, we write $f : x \to y$ whenever $\texttt{dom}(f) = x$ and $\texttt{cod}(f) = y$ for $f \in C_1$ and $x, y \in C_0$,
- $\texttt{id} : C_0 \to C_1$ is a function called *identity*,
- $\texttt{then} : C_1 \times C_1 \to C_1$ is a partial function called *composition*, denoted by $(\fatsemi)$.

such that the following axioms hold:

- $\texttt{id}(x) : x \to x$ for all objects $x \in C_0$,
- for all arrows $f, g \in C_1$, the composition $f \fatsemi g$ is defined iff $\texttt{cod}(f) = \texttt{dom}(g)$, moreover we have $f \fatsemi g : \texttt{dom}(f) \to \texttt{cod}(g)$,
- $\texttt{id}(\texttt{dom}(f)) \fatsemi f = f = f \fatsemi \texttt{id}(\texttt{cod}(f))$ for all arrows $f \in C_1$,
- $f \fatsemi (g \fatsemi h) = (f \fatsemi g) \fatsemi h$ whenever either side is defined for $f, g, h \in C_1$.

Note that we play with the overloaded meaning of the word *class*: we use it to mean both a mathematical collection that need not be a set, and a Python class with its methods and attributes. Reading it in the latter sense, `dom` and `cod` are *attributes* of the arrow class, `then` is a *method*, `id` is a *static method*. Thus, implementing a category in Python means nothing more than subclassing the abstract classes `Ob` and `Arrow` of figure 1.1, and then checking that the axioms hold via some (necessarily non-exhaustive) software tests. The data for a functor $F : C \to D$ is given by a pair of overloaded functions $F_0 : C_0 \to D_0$ and $F_1 : C_1 \to D_1$ such that:

- $F(\texttt{dom}(f)) = \texttt{dom}(F(f))$ and $F(\texttt{cod}(f)) = \texttt{cod}(F(f))$ for all $f \in C_1$,
- $F(\texttt{id}(x)) = \texttt{id}(F(x))$ and $F(f \fatsemi g) = F(f) \fatsemi F(g)$ for all $x \in C_0$ and $f, g \in C_1$.

Figure 1.1: Abstract Python classes for categories and functors.

```
class Ob:
    ...

class Arrow:
    dom : Ob
    cod : Ob

    @staticmethod
    def id(x : Ob) -> Arrow:
        ...

    def then(self, other : Arrow) -> Arrow:
        ...

class Functor:
    @overload
    def __call__(self, x : Ob) -> Ob:
        ...

    @overload
    def __call__(self, f : Arrow) -> Arrow:
        ...
```

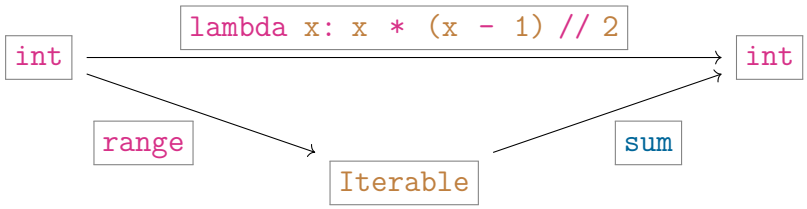
Thus, implementing a functor in Python amounts to subclassing the `Functor` class of figure 1.1 (and then checking that the axioms hold).

Example 1.1.1 *We can define the category* **Python** *with objects the class of all Python types and arrows the class of all Python functions. Domain and codomain of may be extracted from type annotations. Identity may given by* `lambda *xs: xs` *and the composition by* `lambda f, g: lambda *xs: f(*g(*xs))`*. (The star takes care of functions with multiple arguments.) However, equality of functions in Python is undecidable so there will be no way to check the axioms hold in general. We can define endofunctors* **Python** $\to$ **Python** *the same way they are defined in Haskell. For example, we can define a list functor which sends a type* `t` *to* `List[t]` *and a function* `f` *to* `lambda *xs: map(f, xs)`*.

Example 1.1.2 *When the class of objects and arrows are in fact sets, C is called a* small category*. For example, the category* **FinSet** *has the set of all finite sets as objects and the set of all functions between them as arrows. This time equality of functions between finite sets is decidable, so we can write unit tests that check that the axioms hold on specific examples.*

Example 1.1.3 *When the class of objects and arrow are finite sets, we can draw the category as a finite graph with objects as vertices and arrows as edges, together with a list of equations between the paths. A functor $F : C \to D$ from such a finite category C is called a* commutative diagram *in D. They play the same*

role in category theory as equations in set theory. For example, take the following commutative diagram in **Python***:*



Example 1.1.4 *The category* $\mathbf{Mat}_{\mathbb{S}}$ *has natural numbers as objects and* $n \times m$ *matrices with values in* $\mathbb{S}$ *as arrows* $n \to m$*. The identity and composition are given by the identity matrix and matrix multiplication respectively. In order for matrix multiplication to be well-defined and for* $\mathbf{Mat}_{\mathbb{S}}$ *to be a category, the scalars* $\mathbb{S}$ *should have the structure of a* rig *(a riNg without Negatives). When the scalars are complex numbers* $\mathbb{S} = \mathbb{C}$*, the category* $\mathbf{Mat}_{\mathbb{C}}$ *is equivalent to the category of finite dimensional vector spaces and linear maps. When the scalars are Booleans with disjunction and conjunction as addition and multiplication* $\mathbb{S} = \mathbb{B}$*, the category* $\mathbf{Mat}_{\mathbb{B}}$ *is equivalent to the category of finite sets and relations. There is a faithful functor (i.e. injective on arrows with the same domain and codomain)* $\mathbf{FinSet} \to \mathbf{Mat}_{\mathbb{B}}$ *which sends finite sets to their cardinality and functions to their graph.*

Example 1.1.5 *The category* **Circ** *has natural numbers as objects and n-qubit quantum circuits as arrows* $n \to n$*. There is a functor* `eval` $: \mathbf{Circ} \to \mathbf{Mat}_{\mathbb{C}}$ *which sends* n *qubits to* 2^n *dimensions and evaluates each circuit to its unitary matrix.*

Example 1.1.6 *Just about any class of mathematical structures as objects and their homomorphisms as arrows will yield a category. For example, the category* **Set** *of sets and functions, the category* **Mon** *of monoids and homomorphisms, the category* **Cat** *of small categories and functors, etc.*

References

[Ber+20] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. "PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations". In: *arXiv:1811.04968 [physics, physics:quant-ph]* (Feb. 2020). arXiv: 1811.04968 [physics, physics:quant-ph].

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

[Cro18] Andrew Cross. "The IBM Q Experience and QISKit Open-Source Quantum Computing Software". In: 2018 (Jan. 2018), p. L58.003.

[HM17] Matthew Honnibal and Ines Montani. "spaCy 2: Natural Language Understanding with Bloom Embeddings, Convolutional Neural Networks and Incremental Parsing". In: *To appear* 7.1 (2017), pp. 411–420.

[JS91] André Joyal and Ross Street. "The Geometry of Tensor Calculus, I". In: *Advances in Mathematics* 88.1 (July 1991), pp. 55–112. DOI: 10.1016/0001-8708(91)90003-P.

[LB02] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *arXiv:cs/0205028* (May 2002). arXiv: cs/0205028.

[Man+14] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 2014, pp. 55–60.

[Meu+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. "SymPy: Symbolic Computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. DOI: 10.7717/peerj-cs.103.

[Siv+20] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. "Tket : A Retargetable Compiler for NISQ Devices". In: *arXiv:2003.10611 [quant-ph]* (Mar. 2020). arXiv: 2003.10611 [quant-ph].