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

February 24, 2022

Abstract

This thesis introduces a framework for quantum natural language processing (QNLP) based on a simple yet powerful analogy between computational linguistics and quantum mechanics: grammar as entanglement. The grammar of a sentence connects the meaning of words in the same way that entanglement connects the states of quantum systems, they are both structures of information flow. We turn this language-to-qubit analogy into an algorithm that maps the grammatical structure of sentences onto the architecture of parameterised quantum circuits. We then use a hybrid classical-quantum algorithm to train the model so that evaluating the circuits computes the meaning of sentences in some data-driven task. The implementation of these QNLP models led to the development of DisCoPy, a Python library for computing with string diagrams based on a premonoidal approach to computational category theory. We formalise our QNLP models as monoidal functors from grammar to quantum circuits and we introduce the idea of functorial learning, i.e. learning structure-preserving functors from diagram-like data. In order to learn optimal functor parameters via gradient descent, we introduce the notion of diagrammatic differentiation, a graphical calculus for computing the gradient of quantum circuits and string diagrams in general.

Contents

Introduction

What are quantum computers good for?

> Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy.
>
> *Simulating Physics with Computers*, Feynman (1981)

Quantum computers harness the principles of quantum theory such as superposition and entanglement to solve information-processing tasks. In the last 42 years, quantum computing has gone from theoretical speculations to the implementation of machines that can solve problems beyond what is possible with classical means. This section will sketch a brief and biased history of the field and of its future challenges.

In 1980, Benioff [Ben80] takes the abstract definition of a computer and makes it physical: he designs a quantum mechanical system whose time evolution encodes the computation steps of a given Turing machine. In retrospect, this may be taken as the first proof that quantum mechanics can simulate classical computers. The same year, Manin [Man80] looks at the opposite direction: he argues that it would take exponential time for a classical computer to simulate a generic quantum system. Feynman [Fey82; Fey85] comes to the same conclusion and suggests a way to simulate quantum mechanics much more efficiently: building a quantum computer!

So what are quantum computers good for? Feynman's intuition gives us a first, trivial answer: at least quantum computers could simulate quantum mechanics efficiently. Deutsch [Deu85] makes the question formal by defining quantum Turing machines and the circuit model. Deutsch and Jozsa [DJ92] design the first quantum algorithm and prove that it solves *some* problem exponentially faster

than any classical *deterministic* algorithm.[1] Simon [Sim94] improves on their result by designing a problem that a quantum computer can solve exponentially faster than any classical algorithm. Deutsch-Jozsa and Simon relied on oracles[2] and promises[3] and their problems have little practical use. However, they inspired Shor's algorithm [Sho94] for prime factorisation and discrete logarithm. These two problems are believed to require exponential time for a classical computer and their hardness is at the basis of the public-key cryptography schemes currently used on the internet.

In 1997, Grover provides another application for quantum computers: "searching for a needle in a haystack" [Gro97]. Formally, given a function $f : X \to \{0, 1\}$ and the promise that there is a unique $x \in X$ with $f(x) = 1$, Grover's algorithm finds x in $O(\sqrt{|X|})$ steps, quadratically faster than the optimal $O(|X|)$ classical algorithm. Grover's algorithm may be used to brute-force symmetric cryptographic keys twice bigger than what is possible classically [BBD09]. It can also be used to obtain quadratic speedups for the exhaustive search involved in the solution of NP-hard problems such as constraint satisfaction [Amb04]. Independently, Benett et al. [Ben+97] prove that Grover's algorithm is in fact optimal, adding evidence to the conjecture that quantum computers cannot solve these NP-hard problems in polynomial time. Chuang et al. [CGK98] give the first experimental demonstration of a quantum algorithm, running Grover's algorithm on two qubits.

Shor's and Grover's discovery of the first real-world applications sparked a considerable interest in quantum computing. The core of these two algorithms has then been abstracted away in terms of two subroutines: phase estimation [Kit95] and amplitude amplification [Bra+02], respectively. Making use of both these subroutines, the HHL[4] algorithm [HHL09] tackles one of the most ubiquitous problems in scientific computing: solving systems of linear equations. Given a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, we want to find a vector x such that $Ax = b$. Under some assumptions on the sparsity and the condition number of A, HHL finds (an approximation of) x in time logarithmic in n when a classical algorithm would take quadratic time simply to read the entries of A. This initiated a new wave of enthusiasm for quantum computing with the promise of exponential speedups for machine learning tasks such as regression [WBL12], clustering [LMR13], classification [RML14], dimensionality reduction [LMR14] and recommendation [KP16].

[1] A classical *randomised* algorithm solves the problem in constant time with high probability.
[2] An oracle is a black box that allows a Turing machine to solve a certain problem in one step.
[3] The input is promised to satisfy a certain property, which may be hard to check.
[4] Named after its discoverers Harrow, Hassidim and Lloyd.

The narrative is appealing: machine learning is about finding patterns in large amounts of data represented as high-dimensional vectors and tensors, which is precisely what quantum computers are good at. The argument can be formalised in terms of complexity theory: HHL is BQP-complete[1] hence if there is an exponential advantage for quantum algorithms at all there must be one for HHL.

However, the exponential speedup of HHL comes with some caveats, thoroughly analysed by Aaronson [Aar15]. Two of these challenges are common to many quantum algorithms: 1) the efficient encoding of classical data into quantum states and 2) the efficient extraction of classical data via quantum measurements. Indeed, what HHL really takes as input is not a vector b but a quantum state $|b\rangle = \sum_{i=1}^{n} b_i|i\rangle$ called its amplitude encoding. Either the input vector b has enough structure that we can describe it with a simple, explicit formula. This is the case for example in the calculation of electromagnetic scattering cross-sections [CJS13]. Or we assume that our classical data has been loaded onto a quantum random-access memory (qRAM) that can prepare the state in logarithmic time [GLM08]. Not only is qRAM a daunting challenge from an engineering point of view, in some cases it also requires too much error correction for the state preparation to be efficient [Aru+15]. Symmetrically, the output of HHL is not the solution vector x itself but a quantum state $|x\rangle$ from which we can measure some observable $\langle x|M|x\rangle$. If preparing the state $|b\rangle$ requires a number of gates exponential in the number of qubits, or if we need exponentially many measurements of $|x\rangle$ to compute our classical output, then the quantum speedup disappears.

Shor, Grover and HHL all assume *fault-tolerant* quantum computers [Sho96]. Indeed, any machine we can build will be subject to noise when performing quantum operations, errors are inevitable: we need an error correcting code that can correct these errors faster than they appear. This is the content of the *quantum threshold theorem* [AB08] which proves the possibility of fault-tolerant quantum computing given physical error rates below a certain threshold. One noteworthy example of such a quantum error correction scheme is Kitaev's toric code [Kit03] and the general idea of topological quantum computation [Fre+03] which offers the long-term hope for a quantum computer that is fault-tolerant "by its physical nature". However this hope relies on the existence of quasi-particles called Majorana zero-modes, which as of 2021 has yet to be experimentally demonstrated [Bal].

The road to large-scale fault-tolerant quantum computing will most likely be a long one. So in the meantime, what can we do with the noisy intermediate-scale

[1] A BQP-complete problem is one that is polynomial-time equivalent to the circuit model, the hardest problem that a quantum computer can solve with bounded error in polynomial time.

quantum machines we have available today, in the so-called NISQ era [Pre18]? Most answers involve a hybrid classical-quantum approach where a classical algorithm is used to optimise the preparation of quantum states [McC+16]. Prominent examples include the quantum approximate optimisation algorithm (QAOA [FGG14]) for combinatorial problems such as maximum cut and the variational quantum eigensolver (VQE [Per+14]) for approximating the ground state of chemical systems. These variational algorithms depend on the choice of a parameterised quantum circuit called the *ansatz*, based on the structure of the problem and the resources available. Some families of ansätze such as instantaneous quantum polynomial-time (IQP) circuits are believed to be hard to simulate classically even at constant depth [SB09], opening the door to potentially near-term NISQ speedups.

Although the hybrid approach first appeared in the context of machine learning [Ban+08], the idea of using parameterised quantum circuits as machine learning models went mostly unnoticed for a decade [BLS19]. It was rediscovered under the name of quantum neural networks [FN18] then implemented on two-qubits [Hav+19], generating a new wave of attention for quantum machine learning. The idea is straightforward: 1) encode the input vector $x \in \mathbb{R}^n$ as a quantum state $|\phi_x\rangle$ via the ansatz of our choice, 2) initialise a random vector of parameters $\theta \in \mathbb{R}^d$ and encode it as a measurement M_θ, again via some choice of ansatz 3) take the probability $y = \langle\phi(x)|M_\theta|\phi(x)\rangle$ as the prediction of the model. A classical algorithm then uses this quantum prediction as a subroutine to find the optimal parameters θ in some data-driven task such as classification.

One of the many challenges on the way to solving real-world problems with parameterised quantum circuits is the existence of *barren plateaus* [McC+18]: with random circuits as ansatz, the probability of non-zero gradients is exponentially small in the number of qubits and our classical optimisation gets lost in a flat landscape. One can help but notice the striking similarity with the vanishing gradient problem for classical neural networks, formulated twenty years earlier [Hoc98]. Barren plateaus do not appear in circuits with enough structure such as quantum convolutional networks [Pes+21], they can also be mitigated by structured initialisation strategies [Gra+19]. Another direction is to avoid gradients altogether and use *kernel methods* [SK19]: instead of learning a measurement M_θ, we use our NISQ device to estimate the distance $|\langle\phi_{x'}|\phi_x\rangle|^2$ between pairs of input vectors $x, x' \in \mathbb{R}^n$ embedded in the high-dimensional Hilbert space of our ansatz. We then use a classical support vector machine to find the optimal hyperplane that separates our data, with theoretical guarantees to learn quantum models at least

as good as the variational approach [Sch21].

Random quantum circuits may be unsuitable for machine learning, but they play a crucial role in the quest for *quantum advantage*, the experimental demonstration of a quantum computer solving a task that cannot be solved by classical means in any reasonable time. We are back to Feynman's original intuition: sampling from a random quantum circuit is the perfect candidate for such a task. The end of 2019 saw the first claim of such an advantage with a 53-qubit computer [Aru+19]. The claim was almost immediately contested by a classical simulation of 54 qubits in two and a half days [Ped+19] then in five minutes [Yon+21]. Zhong et al. [Zho+20] made a new claim with a 76-photon linear optical quantum computer followed by another with a 66-qubit computer [Wu+21; Zhu+21]. They estimate that a classical simulation of the sampling task they completed in a couple of hours would take at least ten thousand years.

Now that quantum computers are being demonstrated to compute something beyond classical, the question remains: can they compute something *useful*?

Why should we make NLP quantum?

> A girl operator typed out on a keyboard the following Russian text in English characters: "Mi pyeryedayem mislyi posryedstvom ryechi". The machine printed a translation almost simultaneously: "We transmit thoughts by means of speech." The operator did not know Russian.
>
> *New York Times* (8th January 1954)

The previous section hinted at the fact that quantum computing cannot simply solve any problem faster. There needs to be some structure that a quantum computer can exploit: its own structure in the case of physics simulation or the group-theoretic structure of cryptographic protocols in Shor's algorithm. So why should we expect quantum computers to be any good at natural language processing (NLP)? This section will argue that natural language shares a common structure with quantum theory, in the form of two linguistic principles: *compositionality* and *distributionality*.

The history of artificial intelligence (AI) starts in 1950 with a philosophical question from Turing [Tur50]: "Can machines think?" reformulated in terms of a game, now known as the Turing test, in which a machine tries to convince a

human interrogator that it is human too. In order to put human and machine on an equal footing, Turing suggests to let them communicate only via written language: his thought experiment actually defined an NLP task. Only four years later, NLP goes from philosophical speculation to experimental demonstration: the IBM 701 computer successfully translated sentences from Russian to English such as "They produce alcohol out of potatoes." [Hut04]. With only six grammatical rules and a 250-word vocabulary taken from organic chemistry and other general topics, this first experiment generated a great deal of public attention and the overly-optimistic prediction that machine translation would be an accomplished task in "five, perhaps three" years.

Two years later, Chomsky [Cho56; Cho57] proposes a hierarchy of models for natural language syntax which hints at why NLP would not be solved so fast. In the most expressive model, which he argues is the most appropriate for studying natural language, the parsing problem is in fact Turing-complete. Let alone machine translation, merely deciding whether a given sequence of words is grammatical can go beyond the power of any physical computer. Chomsky's parsing problem is a linguistic reinterpretation of an older problem from Thue [Thu14], now known as the *word problem for monoids*[1] and proved undecidable by Post [Pos47] and Markov [Mar47] independently. This reveals a three-way connection between theoretical linguistics, computer science and abstract algebra which will pervade much of this thesis. But if we are interested in solving practical NLP problems, why should we care about such abstract constructions as formal grammars?

Most NLP tasks of interest involve natural language *semantics*: we want machines to compute the *meaning* of sentences. Given the grammatical structure of a sentence, we can compute its meaning as a function of the meanings of its words. This is known as the *principle of compositionality*, usually attributed to Frege.[2] It was already implicit in Boole's *laws of thought* [Boo54] and then made explicit by Carnap [Car47]. Montague [Mon70a; Mon70b; Mon73] applied this principle in linguistics for the first time, arguing that there is "no important theoretical difference between natural languages and the artificial languages of logicians". From a theoretical principle, one may argue that compositionality became the basis of the symbolic approach to NLP, also known as *good old-fashioned AI* (GOFAI) [Hau89]. Word meanings are first encoded in a machine-readable format, then the machine

[1] Historically, Thue, Markov and Post were working with *semigroups*, i.e. unitless monoids.

[2] Compositionality does not appear in any of Frege's published work [Pel01]. Frege [Fre84] did state what is known as the *context principle*: "it is enough if the sentence as whole has meaning; thereby also its parts obtain their meanings". This can be taken as a kind of dual to compositionality: the meanings of the words are functions of the meaning of the sentence.

can compose them to answer complex questions. This approach culminated in 2011 with IBM Watson defeating a human champion at *Jeopardy!* [LF11].

The same year, Apple deploy their virtual assistant in the pocket of millions of users, soon followed by internet giants Amazon and Google. While Siri, Alexa and their competitors have made NLP mainstream, none of them make any explicit use of formal grammars. Instead of the complex grammatical analysis and knowledge representation of expert systems like Watson, the AI of these next-generation NLP machines is powered by deep neural networks and machine learning of big data. Although their architecture got increasingly complex, these neural networks implement a simple statistical concept: *language models*, i.e. probability distributions over sequences of words. Instead of the compositionality of symbolic AI, these statistical methods rely on another linguistic principle, *distributionality*: words with similar distributions have similar meanings. Intuitively,

This principle may be traced back to Wittgenstein's *Philosophical Investigations*: "the meaning of a word is its use in the language" [Wit53], usually shortened into the slogan *meaning is use*. It was then formulated in the context of computational linguistics by Harris [Har54], Weaver [Wea55] and Firth [Fir57], who coined the famous quotation: "You shall know a word by the company it keeps!" Before deep neural networks took over, the standard way to formalise distributionality had been *vector space models* [SWY75]. We have a set of N words appearing in a set of M documents and we simply count how many times each word appears in each document to get a $M \times N$ matrix. We normalise it with a weighting scheme like tf-idf (term frequency by inverse document frequency), factorise it (via e.g. singular value decomposition or non-negative matrix factorisation) and we're done! The columns of the matrix encode the meanings of words, taking their inner product yields a measure of word similarity which can then be used in tasks such as classification or clustering. This method has the advantage of simplicity and it works surprisingly well in a wide range of applications from spam detection to movie recommendation [TP10]. Its main limitation is that a sentence is represented not as a sequence but as a *bag of words*, the word vectors will be the same whether the corpus contained "dog bites man" or "man bites dog". A standard way to fix this is to compute vectors not for words in isolation but for n-grams, windows of n consecutive words for some fixed size n. However the fix has its own limits: if n is too small we cannot detect any long-range correlations, if it is too big then the matrix is so sparse that we cannot detect anything at all.

In contrast, the recurrent neural networks (RNNs) of Rumelhart, Hinton and

Williams [RHW86] are inherently sequential and their internal state can encode arbitrarily long-range correlations. At each step, the network processes the next word in a sequence and updates its internal state. This internal memory can then be used to predict the rest of the sequence, or fed as input to another network e.g. for translation into another language. Once the obstacles to training were overcome (such as the vanishing gradients mentioned above), RNN architectures such as long short-term memory (LSTM) [HS97] set records in a variety of NLP tasks such as language modeling [SMH11], speech recognition [GMH13] and machine translation [SVL14]. The purely sequential approach of RNNs turned out to be limited: when the network is done reading, the information from the first word has to propagate through the entire text before it can be translated. Bidirectional RNNs [SP97] fix this issue by reading both left-to-right and right-to-left. Nonetheless, it is somewhat unsatisfactory from a cognitive perspective (humans manage to understand text without reading backward, why should a machine do that?) and also harder to use in online settings where words need to be processed one at a time.

Attention mechanisms provide a much more elegant solution: instead of assuming that the "company" of a word is its immediate left and right neighbourhood, we let the neural network itself learn which words are relevant to which. First introduced as a way to boost the performance of RNNs on translation tasks [BCB16], attention has then become the basis of the *transformer model* [Vas+17]: a stack of attention mechanisms which process sequences without recurrence altogether. Starting with BERT [Dev+19], transformers have replaced RNNs as the state-of-the-art NLP model, culminating with the GPT-3 language generator authoring its own article in *The Guardian* [GPT20]: "A robot wrote this entire article. Are you scared yet, human?"

Indeed *why* should we be scared? Because we are ignorant of *how* the robot wrote the article and we cannot explain what in its billions of parameters made it write the way it did. Transformers and neural networks in general are *black boxes*: we can probe the way they map inputs to outputs, but if we look at the terabytes of weights in between, we find no interpreation of the mapping. Moreover without explanability there can be no fairness: if we cannot explain how its decisions are made, we can hardly prevent the network from reproducing the discriminations present both in the datasets and in the assumptions of the data scientist. We argue that explainable AI requires to make the distributional black boxes transparent by endowing them with a compositional structure: we need *compositional distributional* (DisCo) models that reconcile symbolic GOFAI

with deep learning.

DisCo models have their roots in neuropsychology rather than AI. Indeed, they first appeared as models of the brain rather than architectures of learning machines. In their seminal work [MP43], McCullogh and Pitts give the first formal definition of neural networks and show how their "all-or-nothing" behaviour[1] allow them to encode a fragment of propositional logic. Hebb [Heb49] then introduced the first biological mechanism to explain learning and structured perception: "neurons that fire together, wire together". These computational models of the brain became the basis of *connectionism* [Smo87; Smo88] and the *neurosymbolic* [Hil97] approach to AI: high-level symbolic reasoning emerges from low-level neural networks. An influential example is Smolensky's *tensor product representation* [Smo90], where discrete structures such as lists and trees are embedded into the tensor product of two vector spaces, one for variables and one for values. Concretely, a list $x_1, \dots, x_n$ of n vectors of dimension d is represented as a tensor $\sum_{i \leq n} |i\rangle \otimes x_i \in \mathbb{R}^n \otimes \mathbb{R}^d$. Smolensky [Smo90] is also the first to make the analogy between the distributional representations of compositional structures in AI and the group representations of quantum physics. He argues that symbolic structures embed in neural networks in the same way that the symmetries of particles embed in their state space: via *representation theory*, a precursor of *category theory* which we discuss in the next section.

Clark and Pulman [CP07b] propose to apply this tensor product representation to NLP, but they note its main weakness: lists of different lengths do not live in the same space, which makes it impossible to compare sentences with different grammatical structures. The categorical compositional distributional (DisCoCat) models of Clark, Coecke and Sadrzadeh [CCS08; CCS10] overcome this issue by taking the analogy with quantum one step further. Word meanings and grammatical structure are to linguistics what quantum states and entanglement structure are to physics. DisCoCat word meanings live in vector spaces and they compose with tensor products: the states of quantum theory do too. Grammar tells you how words are connected and how information flows in a sentence and in the same way, entanglement connects quantum states and tells you how information flows in a complex quantum system. This analogy allows to borrow well-established mathematical tools from quantum theory, and it was implemented on classical hardware with some empirical success on small-scale tasks such as sentence comparison [Gre+10] and word sense disambiguation [GS11; KSP13]. However representing the meaning of sentences as quantum processes comes with a price: they can be

[1] A neuron's response is either maximal or zero, regardless of the stimulus strength.

exponentially hard to simulate classically.

If DisCoCat models are intractable for classical computers, why not use a quantum computer instead? Zeng and Coecke [ZC16] answered this question with the first quantum natural language processing (QNLP) algorithm[1] and the proof of a quadratic speedup on a sentence classification task. Wieber et al. [Wie+19] later defined a QNLP algorithm based on a generalisation of the tensor product representation and proved it is BQP-complete: if any quantum algorithm has an exponential advantage, then in principle there must be one for QNLP. However promising they may be, both algorithms assume fault-tolerance and they are at least as far away from solving real-world problems as Grover and HHL.

This is where the work presented in this thesis comes in: we show it is possible to implement DisCoCat models on the machines available today. The author and collaborators [Mei+20a; Coe+20a] introduced the first NISQ-friendly framework for QNLP by translating DisCoCat models into variational quantum algorithms. We then implemented this framework and demonstrated the first QNLP experiment on a toy question-answering task [Mei+20b] and more recent experiments showed empirical success on a larger-scale classification task [Lor+21]. Our framework was later applied to machine translation [Abb+21; Vic21], word-sense disambiguation [Hof21] and even to generative music [Mir+21]. Future experiments will have to demonstrate that QNLP is more than a mere analogy and that it can achieve *quantum advantage on a useful task*. But before we can discuss our implementation in detail, we have to make the DisCoCat analogy formal.

How can category theory help?

> I should still hope to create a kind of *universal symbolistic* (*spécieuse générale*) in which all truths of reason would be reduced to a kind of calculus.
>
> *Letter to Nicolas Remond*, Leibniz (1714)

"Every sufficiently good analogy is yearning to become a functor" [Bae06] and we will see that the analogy behind DisCoCat models is indeed a functor. Coecke et al. [CGS13] make a meta-analogy between their models of natural language and *topological quantum field theories* (TQFTs). Intuitively, there is an analogy

[1] We do not consider previous algorithms that are inspired by quantum theory but run on classical computers such as the frameworks of Chen [Che02] and Blacoe et al. [BKL13].

between regions of spacetime and quantum processes: both can be composed either in sequence or in parallel. TQFTs formalise this analogy: they assign a quantum system to each region of space and a quantum process to each region of spacetime, in a way that respects sequential and parallel composition. In the same structure-preserving way, DisCoCat models assign a vector space to each grammatical type and a linear map to each grammatical derivation. Both TQFTs and DisCoCat can be given a one-sentence definition in terms of category theory: they are examples of *functors into the category of vector spaces*.

How can the same piece of general abstract nonsense (category theory's nickname) apply to both quantum gravity and natural language processing? And how can this nonsense be of any help in the implementation of QNLP algorithms? This section will answer with a brief and biased history of category theory and its applications to quantum physics and computational linguistics, from an abstract framework for meta-mathematics to a concrete toolbox for NLP on quantum hardware. First, a short philosophical digression on the etymology of the words "functor" and "category" shall bring some light to their divergent meanings in mathematics and linguistics.

The word "functor" first appears in Carnap's *Logical syntax of language* [Car37] to describe what would be called a *function symbol* in a modern textbook on first-order logic. He introduces them as a way to reduce the laws of empirical sciences like physics to the pure syntax of his formal logic, taking the example of a *temperature functor* T such that $T(3) = 5$ means "the temperature at position 3 is 5"[1]. In the linguistics community, this meaning has then drifted to become synonymous with *function words* such as "such", "as", "with", etc. These words do not refer to anything in the world but serve as the grammatical glue between the *lexical words* that describe things and actions. They represent less than one thousandth of our vocabulary but nearly half of the words we speak [CP07a].

Categories (from the ancient Greek , "that which can be said") have a much older philosophical tradition. In his *Categories* [Ari66], Aristotle first makes the distinction between the simple forms of speech (the things that are "said without any combination" such as "man" or "arguing") and the composite ones such as "a man argued". He then classifies the simple, atomic things into ten categories: "each signifies either substance or quantity or qualification or a relative or where or when or being-in-a-position or having or doing or being-affected". A common explanation [Ryl37] for how Aristotle arrived at such a list is that it comes from

[1] MacLane [Mac38] would later remark that Carnap's formal language cannot express the coordinate system for positions, nor the scale in which temperature is measured.

the possible *types of questions*: the answer to "What is it?" has to be a substance, the answer to "How much?" a quantity, etc. Although he was using language as a tool, his system of categories aims at classifying things in the world, not forms of speech: it was meant as an *ontology*, not a grammar. In his *Critique of Pure Reason* [Kan81], Kant revisits Aristotle's system to classify not the world, but the mind: he defines categories of understanding rather than categories of being. The idea that every object (whether in the world or in the mind) is an object of a certain type has then become foundational in mathematical logic and Russell's *theory of types* [Rus03]. The same idea has also had a great influence in linguistics and especially in the *categorial grammar* tradition initiated by Ajdukiewicz [Ajd35] and Bar-Hillel [Bar53; Bar54], where categories have now become synonymous with *grammatical types* such as nouns, verbs, etc.

Independently of their use in linguistics, Eilenberg and MacLane [EM42a; EM42b; EM45] gave categories and functors their current mathematical definition. Inspired by Aristotle's categories of things and Kant's categories of thoughts, they defined categories as types of *mathematical structures*: sets, groups, spaces, etc. Their great insight was to focus not on the content of the objects (elements, points, etc.) but on the composition of the *arrows* between them: functions, homomorphisms, continuous maps, etc. Applying the same insight to categories themselves, what really matters are the arrows between them: *functors*, maps from one category to another that preserve the form of arrows.[1] A prototypical example is Poincaré's construction of the fundamental group of a topological space [Poi95], which can be defined as a functor from the category of (pointed) topological spaces to that of groups: every continuous map between spaces induces a homomorphism between their fundamental groups, in a way that respects composition and identity. Thus, the abstraction of category theory allowed to formalise the analogies between topology and algebra, proving results about one using methods from the other. It was then used as a tool for the foundation of algebraic geometry by the school of Grothendieck [GD60], which brought the analogy between geometric shapes and algebraic equations to a new level of abstraction and led to the development of *topos theory*.

The establishment of category theory as an independent discipline and as a foundation for mathematics owes much to the work of Lawvere. His influential Ph.D. thesis [Law63] on *functorial semantics* set up a framework for model the-

[1] We can play the same game again: what matters are not so much the functors themselves but the *natural transformations* between them, which is what category theory was originally meant to define. To keep playing that game is to fall in the rabbit hole of infinity category theory [RV16].

ory where logical theories are categories and their models are functors. He then undertook the axiomatisation of the category of sets [Law64] and the category of categories [Law66]. The resulting notion of elementary topos [Law70a] subsumed Grothendieck's definition and emphasised the foundational concept of *adjunction* [Law69; Law70b]. "Adjoint functors arise everywhere" became the slogan of MacLane's classic textbook *Categories for the working mathematician* [Mac71]. Lambek [Lam68; Lam69; Lam72] used the related notion of *cartesian closed categories* to extend the Curry-Howard correspondance between logic and computation into a trinity with category theory: proofs and programs are arrows, logical formulae and data types are objects. The discovery of this three-fold connection resulted in a wide range of applications of category theory to theoretical computer science, surveyed in Scott [Sco00].

This unification of mathematics, logic and computer science has been followed by an ongoing program of categorical foundations for physics, initiated by Lawvere's topos-theoretic treatment of classical dynamics [Law79] and continuum physics [LS86] with Schanuel. As we mentioned at the start of this section, the work of Atiyah [Ati88], Baez and Dolan [BD95] on TQFTs showed categories and functors to be essential tools in the grand unification project of quantum gravity [Bae06]. This now quaternary analogy between physics, mathematics, logic and computation was popularised by Baez and Stay in their *Rosetta Stone* [BS09]. On more concrete grounds, this connection between category theory and quantum physics appeared in Selinger's proposal of a quantum programming language [Sel04] and the development of a quantum lambda calculus [Van04; SV06; SV+09]. The same insight blossomed in the school of *categorical quantum mechanics* (CQM) led by Abramsky and Coecke [AC04; AC08], where quantum processes are arrows in *compact closed categories*. This approach culminated in the *ZX calculus* of Coecke and Duncan [CD08; CD11], a categorical axiomatisation which was proved complete for qubit quantum computing [JPV18; HNW18] with applications including error correction [Cha+18; GF19], circuit optimisation [KvdW20; Dun+20; dBBW20], compilation [CSD20; dGD20] and extraction [Bac+20].

In quantum computing as well, adjunction is fundamental: it underlies the definition of entanglement and the proof of correctness for the *teleportation protocol*. Back in 2004 when Coecke first presented this result at the McGill category theory seminar, Lambek immediately pointed out the analogy with his *pregroup grammars* [Lam99b; Lam01] where adjunction is the only grammatical rule[1]. Half

[1]See [Coe19] for a first-hand account of this story and a praise of Jim Lambek.

a century beforehand, Lambek [Lam58; Lam59; Lam61] had started to unravel the analogy between the derivations in categorial grammars and proof trees in mathematical logic. He then extended this analogy in *Categorial and categorical grammar* [Lam88] where he showed that these grammatical derivations are in fact arrows in *closed monoidal categories* and proposed to cast Montague semantics as a topos-valued functor. Later, he argued not "that categories should play a role in linguistics, but rather that they already do" [Lam99a]. Indeed, Hotz [Hot66] had already proved that Chomsky's generative grammars were *free monoidal categories*, although his original German article was never translated to English and remains confidential. The idea of using functors as semantics had appeared implicitly in Knuth [Knu68] in the context-free case and was made explicit by Benson [Ben70] for unrestricted grammars. From this categorical formulation of linguistics, Lambek [Lam10] first suggested the analogy between linguistics and physics which is the basis of this thesis: *pregroup reductions as quantum processes*.

It is remarkable that Lambek could foresee QNLP without *string diagrams*[1], probably the most powerful tool in the hands of the applied category theorist. They first appeared in another confidential article from Hotz [Hot65] as a formalisation of the diagrams commonly used in electronics. Penrose [Pen71] then used the same notation as an informal shortcut for tedious tensor calculations, and later applied it to relativity theory with Rindler [PR84]. Joyal and Street [JS88; JS91; JS95] gave the first topological definition of string diagrams and characterised them as the arrows of free monoidal categories. At first a piece of mathematical folklore that was hand-drawn on blackboards and rarely included in publications, string diagrams were published at a much bigger scale with the advent of typesetting tools like LaTeXand Ti*k*Z. Selinger's survey [Sel10], makes the hierarchy of categorical structures (symmetric, compact closed, etc.) correspond to a hierarchy of graphical gadgets (swaps, wire bending, etc.). In *Picturing Quantum Processes* [CK17], Coecke and Kissinger introduce quantum theory with over a thousand diagrams. And the list of applications keeps growing: electronics [BF15] and chemistry [BP17], control theory [BE14] and concurrency [BSZ14], databases [BSS18] and knowledge representation [Pat17], Bayesian inference [CS12; CJ19] and causality [KU19], cognition [Bol+17] and game theory [Gha+18], functional programming [Ril18] and machine learning [FST17].

[1]String diagrams do not appear in any of Lambek's published work. Instead, he either uses lines of equations, proof trees or "underlinks" for pregroup adjunctions [Lam08]. He admits "not having had the patience to absorb" the topological definition of Joyal-Street string diagrams [Lam10].

If they are a great tool for writing scientific papers, string diagrams can also be a powerful data structure for developing software applications: quantomatic [KZ15] and its successor PyZX [KvdW19] perform automatic rewriting of diagrams in the ZX calculus, globular [BKV18] and its successor homotopy.io [RV19] are proof assistants for higher category theory, cartographer [SWZ19] and catlab [PSV21] implement diagrams in symmetric monoidal categories, which are also implicit in the circuit data structure of the t|ket⟩ compiler [Siv+20]. String diagrams are the main data structure of our QNLP algorithms: we translate the diagrams of sentences into diagrams of quantum circuits. As none of the existing category theory software was flexible enough, we had to implement our own: DisCoPy [Fel+20], a Python library for computing with functors and diagrams in monoidal categories. DisCoPy then became the engine underlying lambeq [Kar+21], a high-level library for experimental QNLP. Although its development was driven by the implementation of DisCoCat models on quantum computers, DisCoPy was designed as a general-purpose toolkit for applied category theory. It is freely available[1] (as in free beer and in free speech), reliable (with 100% code coverage) and extensively documented[2].

In conclusion, category theory can really be a *theory of anything*: from algebraic geometry and quantum gravity to natural language processing. There is a striking analogy between category theory and string diagrams as a universal graphical language and the *characteristica universalis* and *calculus ratiocinator* dreamt by Leibniz three hundred years ago, a formal language and computational framework that would be able to express all of mathematics, science and philosophy. Indeed, not only can categories be tools for the working mathematicians and scientists, they can also be of help to the philosophers. In the footsteps of Grassmann's *Ausdehnungslehre* [Gra44] and his project of an algebraic formalisation of Hegel, Lawvere [Law89; Law91; Law92; Law96] set out to formulate Hegelian dialectics in terms of adjunctions. This led to the ongoing effort of Schreiber, Corfield and their collaborators on the nLab [SCn21] to translate *Wissenschaft Der Logik* [Heg12] in terms of category theory. Not only can it accommodate the absolute idealism of Hegel, category theory can also deal with the pragmatism of Peirce [Pei06], who developed first-order logic independently of Frege using what was later recognised as the first string diagrams [BT98; BT00; MZ16; HS20]. String diagrams have also been used to model Wittgenstein's language games as functors from a grammar to a category of games [HL18]. In recent work [FTC20], we applied these

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

[2] `https://discopy.readthedocs.io/`

functorial language games to question answering, going from philosophy to NLP via category theory.

Contributions

The first chapter is an extended version of the DisCoPy paper [FTC20]. It emerged from a dialectic teacher-student collaboration with Giovani de Felice: implementing our own category theory library was a way to teach him Python programming. Bob Coecke then added the capital letters to the name of DisCoPy. We list the contributions of each section.

1. We[1] give an introduction to elementary category theory for the Python programmer which is at the same time an introduction to object-oriented programming for the applied category theorist. This includes an implementation of:
    - the category **Pyth** with Python types as objects and functions as arrows (listing 1.1.7),
    - the category $\mathbf{Mat}_{\mathbb{S}}$ with natural numbers as objects and matrices with entries in a rig $\mathbb{S}$ as arrows (listing 1.1.10),
    - free categories (listing 1.1.12) with quantum circuits as example (1.1.13),
    - the category **Cat** with categories as objects and functors as arrows (listing 1.1.14),
    - quotient categories (section 1.1.4),
    - categories with a dagger structure, i.e. an identity-on-objects contravariant involutive endofunctor (section 1.1.5),
    - categories with sums, i.e. enriched in commutative monoids, and bubbles, i.e. arbitrary unary operators on homsets, with the example of neural networks (1.1.28) and propositional logic (1.1.29).
2. We give an elementary definition of string diagrams for monoidal categories. Our construction decomposes the free monoidal category construction into

[1]The "we" of this section refers to the author of this thesis. Although we believe that science is collaboration and that the notion of personal contribution is obsolete, it is in fact required by university regulations: "Where some part of the thesis is not solely the work of the candidate or has been carried out in collaboration with one or more persons, the candidate shall submit a clear statement of the extent of his or her own contribution."

three basic steps: 1) a layer endofunctor on the category of monoidal signatures, 2) the free premonoidal category as a free category of layers and 3) the free monoidal category as a quotient by interchangers. To the best of our knowledge, this *premonoidal approach* had been relegated to mathematical folklore: it was known by those who knew it, yet it never appeared in print. The monoidal categories we implement are all strict and furthermore they are *free on objects* (foo), lemma 1.2.14 shows that every monoidal category is monoidally equivalent to a foo one. This includes:

- **Pyth** with lists of types as objects and tupling as tensor (listing 1.2.16),
- $\mathbf{Tensor}_{\mathbb{S}} \simeq \mathbf{Mat}_{\mathbb{S}}$ with lists of natural numbers as objects and Kronecker product as tensor (listing 1.2.17),
- free monoidal categories (listing 1.2.24) with quantum circuits as example (1.2.26),
- quotient monoidal categories (listing 1.2.4) with quantum circuit optimisation as example (1.2.31),
- monoidal categories with daggers, sums and bubbles (section 1.2.5) with the example of post-processed quantum circuits (1.2.36) and first-order logic à la Peirce (1.2.37).

DisCoPy uses a *point-free* or *tacit programming* style where diagrams are described only by composition and tensor. We discuss how to go from tacit to explicit programming, defining diagrams using the standard syntax for Python functions (section 1.2.6).

3. We prove the equivalence between our elementary definition of diagrams in terms of list of layers and the topological definition in terms of *labeled generic progressive plane graphs*. One side of this equivalence underlies the drawing algorithm of DisCoPy, the other side is the basis of a prototype for an automatic diagram recognition algorithm. We then discuss how to extend this to non-generic, non-progressive, non-planar, non-graph-like diagrams, which opens the door to the next section.

4. We describe our object-oriented implementation of monoidal categories with extra structure. The hierarchy of categorical structures (monoidal, closed, rigid, etc.) is encoded in a hierarchy of Python classes and an inheritance mechanism implements the free-forgetful adjunctions between them. This includes an implementation of:

- free rigid categories, for which we introduce the *snake removal* algorithm to compute normal forms (section 1.4.1),
- the syntax for diagrams in free braided, symmetric, tortile and compact-closed categories (section 1.4.2),
- the syntax for diagrams in free hypergraph categories, i.e. with coherent special commutative Frobenius algebras on each object (section 1.4.3),
- the syntax for diagrams in free cartesian and cocartesian diagrams (section 1.4.4) with **Pyth** as an example of a *rig category* with two monoidal structures (listing 1.4.37),
- the free biproduct completion as the category of matrices with arrows as entries (section 1.4.5), taking quantum measurements as example (1.4.42),
- the syntax for diagrams in closed monoidal categories (section 1.4.6) with currying of functions in **Pyth** as example (1.4.44).

5. We discuss the relationship between our premonoidal approach and the existing graph-based data structures for diagrams in symmetric monoidal categories. This includes:

   - a comparison between our definition of *premonoidal diagrams* as lists of layers and the free premonoidal category as a state construction over a monoidal category (section 1.5.1),
   - a definition of *hypergraph diagrams*, i.e. the arrows of free hypergraph categories, and the subcategories of compact, traced and symmetric diagrams (section 1.5.2),
   - an implementation of **Pyth** as a traced cartesian and cocartesian category (listing 1.5.9) and $\mathbf{Mat}_{\mathbb{B}} \simeq \mathbf{FinRel}$ as a traced biproduct category (listing 1.5.11),
   - an implementation of hypergraph diagrams and its comparison with premonoidal diagrams (section 1.5.3),
   - an implementation of free sesquicategories (i.e. 2-categories without interchangers) with *coloured diagrams* as 2-cells (listing 1.5.13),
   - an implementation of **Cat** as a sesquicategory with (not-necessarily-natural) transformations as 2-cells (listing 1.5.14),

- an implementation of free monoidal 2-categories with diagrams as 1-cells and rewrites as 2-cells (listing 1.5.16).

The second chapter deals with QNLP, building on [Mei+20b; Coe+20a; Mei+20a]. It was joint work with Bob Coecke, Giovanni de Felice and Konstantinos Meichanetzidis. Although we shared a common office, Stefano Gogioso arrived at the same ideas independently with his collaborator Nicolò Chiappori.

- We define QNLP models as functors from grammar to quantum circuits and show that any DisCoCat model can be implemented in this way.
- We develop a rewriting strategy for the resulting circuits which reduces both the required number of qubits and the amount of post-selection. The underlying algorithm, called *snake removal*, computes the normal form of diagrams in rigid monoidal categories.
- We introduce a hybrid classical-quantum algorithm to train QNLP models on a question-answering task. The underlying idea of *functorial learning*, i.e. learning structure-preserving functors from diagram-like data, provides a theoretical framework for machine learning on structured data.

The last section introduces *diagrammatic differentiation*, a graphical calculus for computing the gradients of parameterised diagrams which applies to the training of QNLP models but also to functorial learning in general. Most of the material has been published in joint work with Richie Yeung and Giovanni de Felice [TYF21].

- We generalise the dual number construction from rings to monoidal categories. Dual diagrams are formal sums of a string diagram (the real part) and its derivative with respect to some parameter (the epsilon part).
- We introduce graphical gadgets called bubbles, which can encode arbitrary unary operators on monoidal categories. In particular, they encode differentiation of diagrams and allow to express the standard rules of calculus (linearity, product, chain) entirely in terms of diagrams.
- We study diagrammatic differentiation for the ZX calculus. In the pure case, this allows to compute the gradients of linear maps with respect to phase parameters. In the mixed classical-quantum case, this yields a definition of the parameter-shift rules used in quantum machine learning.
- We define the gradient of QNLP models and any parameterised functor.

Publications

The material presented in this thesis builds on the following publications.

[Mei+20a] Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni de Felice, Nicolò Chiappori, Alexis Toumi, and Bob Coecke. "Quantum Natural Language Processing on Near-Term Quantum Computers". In: *Proceedings 17th International Conference on Quantum Physics and Logic, QPL 2020, Paris, France, June 2 - 6, 2020*. Ed. by Benoît Valiron, Shane Mansfield, Pablo Arrighi, and Prakash Panangaden. Vol. 340. EPTCS. 2020, pp. 213–229. DOI: 10.4204/EPTCS.340.11.

[FTC20] Giovanni de Felice, Alexis Toumi, and Bob Coecke. "DisCoPy: Monoidal Categories in Python". In: *Proceedings of the 3rd Annual International Applied Category Theory Conference, ACT*. Vol. 333. EPTCS, 2020. DOI: 10.4204/EPTCS.333.13.

[Coe+20a] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "Foundations for Near-Term Quantum Natural Language Processing". In: *CoRR* abs/2012.03755 (2020). arXiv: 2012.03755.

[Mei+20b] Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. "Grammar-Aware Question-Answering on Quantum Computers". In: *ArXiv e-prints* (2020). arXiv: 2012.03756.

[Kar+21] Dimitri Kartsaklis, Ian Fan, Richie Yeung, Anna Pearson, Robin Lorenz, Alexis Toumi, Giovanni de Felice, Konstantinos Meichanetzidis, Stephen Clark, and Bob Coecke. "Lambeq: An Efficient High-Level Python Library for Quantum NLP". In: *CoRR* abs/2110.04236 (2021). arXiv: 2110.04236.

[TYF21] Alexis Toumi, Richie Yeung, and Giovanni de Felice. "Diagrammatic Differentiation for Quantum Machine Learning". In: *Proceedings 18th International Conference on Quantum Physics and Logic, QPL 2021, Gdansk, Poland, and Online, 7-11 June 2021*. Ed. by Chris Heunen and Miriam Backens. Vol. 343. EPTCS. 2021, pp. 132–144. DOI: 10.4204/EPTCS.343.7.

During his DPhil, the author has also published the following articles.

[Bor+19] Emanuela Boros, Alexis Toumi, Erwan Rouchet, Bastien Abadie, Dominique Stutzmann, and Christopher Kermorvant. "Automatic Page Classification in a Large Collection of Manuscripts Based on the International Image Interoperability Framework". In: *International Conference on Document Analysis and Recognition*. 2019. DOI: 10.1109/ICDAR.2019.00126.

[FMT19] Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "Functorial Question Answering". In: *Proceedings Applied Category Theory 2019, ACT 2019, University of Oxford, UK*. Vol. 323. EPTCS. 2019. DOI: 10.4204/EPTCS.323.6.

[Fel+20] Giovanni de Felice, Elena Di Lavore, Mario Román, and Alexis Toumi. "Functorial Language Games for Question Answering". In: *Proceedings of the 3rd Annual International Applied Category Theory Conference 2020, ACT 2020, Cambridge, USA, 6-10th July 2020*. Ed. by David I. Spivak and Jamie Vicary. Vol. 333. EPTCS. 2020, pp. 311–321. DOI: 10.4204/EPTCS.333.21.

[STS20] Dan Shiebler, Alexis Toumi, and Mehrnoosh Sadrzadeh. "Incremental Monoidal Grammars". In: *CoRR* abs/2001.02296 (2020). arXiv: 2001.02296.

[TK21] Alexis Toumi and Alex Koziell-Pipe. "Functorial Language Models". In: *CoRR* abs/2103.14411 (2021). arXiv: 2103.14411.

[Coe+21] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "How to Make Qubits Speak". In: *CoRR* abs/2107.06776 (2021). arXiv: 2107.06776.

[McP+21] Lachlan McPheat, Gijs Wijnholds, Mehrnoosh Sadrzadeh, Adriana Correia, and Alexis Toumi. "Anaphora and Ellipsis in Lambek Calculus with a Relevant Modality: Syntax and Semantics". In: *CoRR* abs/2110.10641 (2021). arXiv: 2110.10641.

Outreach

The content of this thesis has also been the subject of science popularisation aimed at a wide audience.

- A blog post summarising our first experiment and two podcasts with long discussions on the topic.

  [Coe+20b] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. *Quantum Natural Language Processing*. Apr. 7, 2020. URL: https://medium.com/cambridge-quantum-computing/quantum-natural-language-processing-748d6f27b31d (visited on 02/24/2022).

  [Fut21] Futurati Podcast. *Ep. 52: Bob Coecke and Konstantinos Meichanetzidis on Quantum Natural Language Processing*. Sept. 21, 2021. URL: https://www.youtube.com/watch?v=5YZG96t8SLQ (visited on 02/24/2022).

  [Mac21] Machine Learning Street Talk. *#53 Quantum Natural Language Processing - Prof Bob Coecke*. 2021. URL: https://www.youtube.com/watch?v=X9uSV1YcOy4 (visited on 02/24/2022).

- A hackathon where students implemented QNLP experiments with DisCoPy.

  [Mol21] Paula Garcia Molina. *QNLP Qiskit Hackathon*. Oct. 22, 2021. URL: https://github.com/PaulaGarciaMolina/QNLP_Qiskit_Hackathon (visited on 02/24/2022).

- A presentation at an educational event for programmers and data scientists.

  [20b] *PyData Berlin*. Sept. 21, 2020. URL: https://www.youtube.com/watch?v=5jK8qEQvR-o (visited on 02/24/2022).

- A press release explaining QNLP in plain English.

[20a] "CQC Researchers Make Major Quantum NLP Advance in Steps Toward 'Meaning Aware' Computers". In: *The Quantum Insider* (Dec. 10, 2020). URL: https://thequantuminsider.com/2020/12/10/meaning-aware-computers-cqc-researchers-make-major-nlp-advance-in-using-quantum-computers-to-understand-language-and-towards-achieving-meaningful-quantum-advantage/ (visited on 02/24/2022).

- Press releases introducing lambeq [Kar+21] to a business audience.

[21] "Cambridge Quantum Releases World's First Quantum Natural Language Processing Toolkit and Library". In: *HPC wire* (Oct. 13, 2021). URL: https://www.hpcwire.com/off-the-wire/cambridge-quantum-releases-worlds-first-quantum-natural-language-processing-toolkit/ (visited on 02/24/2022).

[Smi21] Paul Smith-Goodson. "Cambridge Quantum Makes Quantum Natural Language Processing A Reality". In: *Forbes* (Oct. 13, 2021). URL: https://www.forbes.com/sites/moorinsights/2021/10/13/cambridge-quantum-makes-quantum-natural-language-processing-a-reality/ (visited on 02/24/2022).

1

DisCoPy: Python for the applied category theorist

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. Its main feature are 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.

The aims of this chapter are three-fold: 1) it gives an overview of the DisCoPy package and its design principles, 2) it introduces elementary category theory to the Python programmer and 3) it introduces object-oriented programming to the applied category theorist. The first section introduces categories and functors with no mathematical prerequisites apart from sets and monoids. The second section introduces monoidal categories, defining string diagrams from first principles. The third 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 fourth section introduces monoidal categories with extra structure and the inheritence mechanism which implements this hierarchy of structure. The fifth section gives the category theoretic foundations for our definition of diagrams, which we call the premonoidal approach, it discusses the relationship between this approach and the exisiting graph-based data structures for diagrams in symmetric monoidal categories.

Note that the code presented in this thesis represents a significant refactoring of the original implementation of DisCoPy as available online at the time this thesis is submitted[1].

1.1 Categories in Python

1.1.1 Abstract categories

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*,

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

- $\texttt{id} : C_0 \to C_1$ is a function called *identity*,
- $\texttt{then} : C_1 \times C_1 \to C_1$ is a partial function called *composition*, denoted by $(\fatsemi)$.

Given two objects $x, y \in C_0$, the set[1] $C(x, y) = \{f \in C_1 \mid \texttt{dom}(f), \texttt{cod}(f) = x, y\}$ is called a *homset* and we write $f : x \to y$ whenever $f \in C(x, y)$. We denote the composition $\texttt{then}(f, g)$ by $f \fatsemi g$, translated to `f >> g` or `g << f` in Python. The axioms for the category C are the following:

- $\texttt{id}(x) : x \to x$ 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 listing 1.1.1, and then checking that the axioms hold via some (necessarily non-exhaustive) software tests. The `Category` class is nothing more than a named tuple with two attributes `ob` and `ar` for its object and arrow class respectively.

Listing 1.1.1. Abstract classes for categories, functors and transformations.

```
class Ob: ...

class Arrow:
    dom: Object
    cod: Object

    @staticmethod
    def id(x: Ob) -> Arrow[x, x]: ...
    def then(self, other: Arrow[self.cod, y]) -> Arrow[self.dom, y]: ...

@dataclass
```

[1] We will assume that this forms a set rather than a proper class, i.e. we will only work with *locally small* categories.

```python
class Category:
    ob: type = Ob
    ar: type = Arrow

class Functor:
    dom: Category
    cod: Category

    @overload
    def __call__(self, x: self.dom.ob) -> self.cod.ob: ...

    @overload
    def __call__(self, f: self.dom.ar[x, y]) -> self.cod.ar[self(x), self(y)]: ...

class Transformation:
    C: Category
    D: Category
    dom: Functor[C, D]
    cod: Functor[C, D]

    def __call__(self, x: C.ob) -> D.ar[self.dom(x), self.cod(x)]: ...
```

Note that annotations with dependent types are not supported by any Python implementation yet. We include them here to make the thesis clearer although they do not appear in the DisCoPy code. Since Python cannot statically check that arrow composition is well-typed, the best we can do is raise an `AxiomError` *at runtime.*

The data for a *functor* $F : C \to D$ between two categories C and D is given by a pair of overloaded functions $F : C_0 \to D_0$ and $F : C_1 \to D_1$ 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 $f, g \in C_1$.

Thus, implementing a functor in Python amounts to subclassing the `Functor` class of listing 1.1.1 (and then implementing software tests to check that the axioms hold).

The data for a *transformation* $\alpha : F \to G$ between two parallel functors $F, G : C \to D$ is given by a function from objects $x \in C_0$ to components $\alpha(x) : F(x) \to G(x)$ in D. A *natural transformation* is one where $\alpha(x) \fatsemi G(f) = F(f) \fatsemi \alpha(y)$ for all arrows $f : x \to y$ in C. The `Transformation` class is given in listing 1.1.1, checking that a transformation is natural cannot be done formally in Python. In the same way that there is a set Y^X of functions $X \to Y$ for any two sets X and Y, for any

two categories C and D there is a category D^C with functors $C \to D$ as objects and natural transformations as arrows.

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 directed multigraph with objects as nodes and arrows as edges, together with the list of equations between paths. A functor $F : C \to D$ from such a finite category C is called a* commutative diagram *in D. One commutative diagram can state a large number of equations, which can be read by* diagram chasing*.*

Example 1.1.4. *A monoid is the same as a category with one object, i.e. every arrow (element) can be composed with (multiplied by) every other. A preorder, i.e. a set with a reflexive transitive relation, is the same as a category with at most one arrow $x \leq y$ between any two objects x and y. Functors between monoids are the same as homomorphisms, functors between preorders are monotone functions.*

Example 1.1.5. *Just about any class of mathematical structures will be the objects of a category with the transformations between them as arrows: the category* **Set** *of sets and functions, the category* **Mon** *of monoids and homomorphisms, the category* **Preord** *of preorders and monotone functions, the category* **Cat** *of small categories and functors, etc. There are embedding (i.e. injective on objects and arrows) functors from* **Mon** *and* **Preord** *to* **Cat***, i.e. preorders and monoids form a subcategory of* **Cat***. There is a functor from* **Cat** *to* **Preord** *called the preorder collapse which sends a category C to the preorder given by $x \leq y$ iff there is an arrow $f \in C(x, y)$, i.e. we forget the difference between parallel arrows. There is a faithful (i.e. injective on homsets) functor $U :$ **Mon** $\to$ **Set** called the* forgetful functor *which sends monoids to their underlying set and homomorphisms to functions.*

1.1.2 Concrete categories

Example 1.1.6. *We can define the category* **Pyth** *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.*

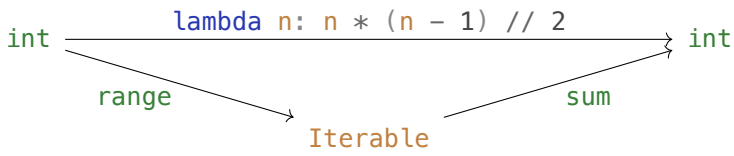
Endofunctors **Pyth** $\to$ **Pyth** *can be thought of as some kind of data containers. For example, we can define a* **List** *functor which sends a type* `t` *to* `List[t]` *and a function* `f` *to* `lambda *xs: map(f, xs)`*. There is a natural transformation* $\eta : \mathbf{Id} \to \mathbf{List}$ *from the obvious identity functor, implemented by the built-in function* `id`*. Its components send objects* `x : t` *of any type* `t` *to the singleton list* `[x] : List[t]`*.*

Listing 1.1.7. Implementation of the category **Pyth** with `type` as objects and `Function` as arrows.

```
class Composable:
    """ Allows syntactic sugar self >> other and self << other and n * self. """
    def cast(self, other):
        return other if isinstance(other, Composable) else self.id(other)

    __rshift__ = __llshift__ = lambda self, other: self.then(self.cast(other))
    __lshift__ = __lrshift__ = lambda self, other: self.cast(other).then(self)
    __rmul__ = lambda self, n: self.id(self.dom).then(*n * [self])

def inductive(method):
    """ Allows to apply a binary method to an arbitrary number of arguments. """
    def result(self, *others):
        if not others: return self
        if len(others) == 1: return method(self, others[0])
        if len(others) > 1: return method(method(self, others[0]), *others[1:])
    return result

@dataclass
class Function(Composable):
    inside: Callable
    dom: type
    cod: type

    @staticmethod
    def id(x: type) -> Function:
        return Function(lambda *xs: xs, x, x)

    @inductive
    def then(self, other: Function) -> Function:
        inside = lambda *xs: other(*tuple(self(*xs)))
        return Function(inside, self.dom, other.cod)
```

```
    def __call__(self, *xs): return self.inside(*xs)
```

Example 1.1.8. *The following commutative diagram denotes a functor* $3 \to$ **Pyth** *from the finite category* 3 *with three objects* $\{0, 1, 2\}$ *and three non-identity arrow* $f : 0 \to 1, g : 1 \to 2$ *and* $h : 0 \to 2$*, with the only non-trivial composition* $f \fatsemi g = h$*.*



It is read as the equation `sum(range(n)) = n * (n - 1) // 2`*.*

Example 1.1.9. *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 at least the structure of a* rig *(a riNg without Negatives, also called a* semiring*): a pair of monoids* $(\mathbb{S}, +, 0)$ *and* $(\mathbb{S}, \times, 1)$ *with the first one commutative and the second a homomorphism for the first, i.e.* $a \times 0 = 0 = 0 \times a$ *and* $(a + b) \times (c + d) = ac + ad + bc + bd$*. 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, the category* $\mathbf{Mat}_{\mathbb{B}}$ *is equivalent to the category of finite sets and relations. There is a faithful functor* $\mathbf{FinSet} \to \mathbf{Mat}_{\mathbb{B}}$ *which sends finite sets to their cardinality and functions to their graph.*

Listing 1.1.10. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ with `int` as objects and `Matrix[dtype]` as arrows.

```
@dataclass
class Matrix(Composable):
    dtype = int

    dom: int
    cod: int
    inside: list[list[dtype]]

    def __class_getitem__(cls, dtype):
        class C(cls): pass
        C.dtype = dtype; return C
```

```
    def __init__(self, inside: list[list[dtype]], dom: int, cod: int):
        self.dom, self.cod = dom, cod
        self.inside = [list(map(self.dtype, row)) for row in inside]

    @classmethod
    def id(cls, x: int) -> Matrix:
        return cls([[i == j for i in range(x)] for j in range(x)], x, x)

    @inductive
    def then(self, other: Matrix) -> Matrix:
        inside = [[sum([self[i][j] * other[j][k] for j in range(other.dom)])
                   for k in range(other.cod)] for i in range(self.dom)]
        return type(self)(inside, self.dom, other.cod)

    def __getitem__(self, key: int) -> Matrix:
        if self.dom == 1: return Matrix([[self.inside[0][key]]], 1, 1)
        return Matrix([self.inside[key]], 1, self.cod)

    for attr in ("__bool__", "__int__", "__float__", "__complex__"):
        def method(self):  # Downcasting a 1 by 1 Matrix to a scalar.
            assert self.dom == self.cod == 1
            return getattr(self[0][0], attr)()
        setattr(Matrix, attr, method)
```

Subscriptable types such as `list[list[int]]` *and the* `__class_getitem__` *method are a new feature of Python* `>=3.10`*. By default, we fix* `Matrix = Matrix[int]`*. We can get Boolean, real and complex matrices with* `Matrix[bool]`*,* `Matrix[float]` *and* `Matrix[complex]` *respectively. Note that this implementation is not meant to be efficient, rather it helps in making the thesis self-contained.*

Example 1.1.11. *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.*

1.1.3 Free categories

The main principles behind the implementation of DisCoPy follow from the concept of a *free object*. Let's start from a simple example. Given a set X, we can construct a monoid $X^\star$ with underlying set $\coprod_{n\in\mathbb{N}} X^n$ the set of all finite lists with elements in X. The associative multiplication is given by list concatenation $X^m \times X^n \to X^{m+n}$ and the unit is given by the empty list denoted $1 \in X^0$. Given a function

$f : X \to Y$, we can construct a homomorphism $f^\star : X^\star \to Y^\star$ defined by element-wise application of f (this is what the built-in `map` does in Python). We can easily check that $(f \fatsemi g)^\star = f^\star \fatsemi g^\star$ and $(\mathtt{id}_X)^\star = \mathtt{id}_{X^\star}$. Thus, we have defined a functor $F : \mathbf{Set} \to \mathbf{Mon}$.

Why is this functor so special? Because it is the *left adjoint* to the forgetful functor $U : \mathbf{Mon} \to \mathbf{Set}$. An *adjunction* $F \dashv U$ between two functors $F : C \to D$ and $U : D \to C$ is a pair of natural transformations $\eta : \mathtt{id}_C \to F \fatsemi U$ and $\epsilon : U \fatsemi F \to \mathtt{id}_D$ called the *unit* and *counit* respectively. In the case of lists, we already mentioned the unit in example 1.1.6: it is the function that sends every object to a singleton list. For a monoid M, the counit $\epsilon(M) : F(U(M)) \to M$ is the monoid homomorphism that takes lists of elements in M and multiplies them. We can easily check that these two transformations are indeed natural, thus we get that *lists are free monoids*. This may be taken as a mathematical explanation for why lists are so ubiquitous in programming. Another equivalent definition of adjunction is in terms of an isomorphism $C(x, U(y)) \simeq D(F(x), y)$ which is natural[1] in $x \in C_0$ and $y \in D_0$. In the adjunction for lists, functions $X \to U(M)$ from a set X to the underlying set of a monoid M are in a natural one-to-one correspondance with monoid homomorphisms $X^\star \to M$. To define a homomorphism from a free monoid, it is sufficient to define the image of each generating element.

Now we want to play the same game with categories instead of monoids. We can define a forgetful functor $U : \mathbf{Cat} \to \mathbf{Set}$ which sends a small category C to its set of objects C_0, and its left adjoint $F : \mathbf{Set} \to \mathbf{Cat}$ which sends a set to the *discrete category* with its elements as objects and only identity arrows. However, this is a rather boring construction because forgetting the arrows of a categories is too much: the forgetful functor U is not faithful. Instead, we need to replace the category of sets with the category of *signatures*. The data for a signature is given by a tuple $\Sigma = (\Sigma_0, \Sigma_1, \mathtt{dom}, \mathtt{cod})$ where:

- Σ_0 is a set of *generating objects*,
- Σ_1 is a set of *generating arrows*, which we will also call *boxes*,
- $\mathtt{dom}, \mathtt{cod} : \Sigma_1 \to \Sigma_0$ are the domain and codomain.

A morphism of signatures $f : \Sigma \to \Gamma$ is a pair of overloaded functions $f : \Sigma_0 \to \Gamma_0$ and $f : \Sigma_1 \to \Gamma_1$ such that $f \fatsemi \mathtt{dom} = \mathtt{dom} \fatsemi f$ and $f \fatsemi \mathtt{cod} = \mathtt{cod} \fatsemi f$. Thus, signatures and their morphisms form a category **Sig** and there is a faithful functor

[1]The isomorphism $C(x, U(y)) \simeq D(F(x), y)$ is natural in x if it is a natural transformation between the two functors $C(-, U(y)), D(F(-), y) : C \to \mathbf{Set}$.

$U : \mathbf{Cat} \to \mathbf{Sig}$ which sends a category to its underlying signature: it forgets the identity and composition. Signatures may be thought of as directed multigraphs *with an attitude* [nLa]. Given a signature Σ, we can define a category $F(\Sigma)$ with nodes as objects and *paths as arrows*. More precisely, an arrow $f : x \to y$ is given by a length $n \in \mathbb{N}$ and a list $f_1, \dots, f_n \in \Sigma_1$ with $\texttt{dom}(f_1) = x$, $\texttt{cod}(f_n) = y$ and $\texttt{cod}(f_i) = \texttt{dom}(f_{i+1})$ for all $i < n$. Given a morphism of signatures $f : \Sigma \to \Gamma$, we get a functor $F(f) : F(\Sigma) \to F(\Gamma)$ relabeling boxes in Σ by boxes in Γ. Thus, we have defined a functor $F : \mathbf{Sig} \to \mathbf{Cat}$, it remains to show that it indeed forms an adjunction $F \dashv U$. This is very similar to the monoid case: the unit sends a box in a signature to the path of just itself, the counit sends a path of arrows in a category to their composition. Equivalently, we have a natural isomorphism $\mathbf{Cat}(F(\Sigma), C) \simeq \mathbf{Sig}(\Sigma, U(C))$: to define a functor $F(\Sigma) \to C$ from a free category is the same as to define a morphism of signatures $\Sigma \to U(C)$.

If lists are such fundamental data structures because they are free monoids, we argue that the arrows of free categories should be just as fundamental: they capture the basic notion of *data pipelines*. Free categories are implemented in the most basic module of DisCoPy, `discopy.cat`, which is sketched in listing 1.1.12.

Listing 1.1.12. Implementation of the free category $F(\Sigma)$ with $\Sigma_0 =$ `Ob` and $\Sigma_1 =$ `Box`.

```python
@dataclass
class Ob:
    name: str
    __str__ = lambda self: self.name

@dataclass
class Arrow(Composable):
    inside: list[Arrow]
    dom: Ob
    cod: Ob

    @classmethod
    def upgrade(cls, old: Arrow) -> Arrow:
        if old if isinstance(old, cls) else cls(old.inside, old.dom, old.cod)

    @classmethod
    def id(cls, x: Ob) -> Arrow:
        return cls.upgrade(Arrow([], x, x))

    def then(self, *others: Arrow) -> Arrow:
```

```
        for f, g in zip((self, ) + others, others): assert f.cod == g.dom
        dom, cod = self.dom, others[-1].cod if others else self.cod
        inside = self.inside + sum([other.inside for other in others], [])
        return self.upgrade(Arrow(inside, dom, cod))

    __len__ = lambda self: len(self.inside)
    __str__ = lambda self: ' >> '.join(map(str, self.inside))\
        if self.inside else '{}.id({})'.format(type(self).__name__, self.dom)

@dataclass
class Box(Arrow):
    upgrade = Arrow.upgrade

    def __init__(self, name: str, dom: Ob, cod: Ob):
        self.name = name
        super().__init__([self], dom, cod)

    def __eq__(self, other):
        if isinstance(other, Box):
            return (self.name, self.dom, self.cod)\
                == (other.name, other.dom, other.cod)
        return isinstance(other, Arrow) and other.inside == [self]

    __str__ = lambda self: self.name
    __hash__ = lambda self: hash(repr(self))
```

The classes `Ob` and `Arrow` for objects and arrows are implemented in a straightforward way, using the built-in `dataclass` decorator to avoid the bureaucracy of defining initialisation, equality, etc. We define the method `__str__` so that `eval(str(f)) == f` for all `f: Arrow`, provided that the names of each object and box is in scope. The method `Arrow.then` accepts any number of arrows `others`, which will prove useful when defining functors. The `Box` class requires more attention: a box `f = Box('f', x, y)` is an arrow with the list of just itself as boxes, i.e. `f.inside == [f]`. For the axiom `f >> id(y) == f == id(x) >> f` to hold, we need to make sure that `f == Arrow(x, y, [f])`, i.e. a box is equal to the arrow with just itself as boxes. The main subtlety in the implementation is the class method `upgrade` which takes an `old: Arrow` as input and returns a new member of a given `cls`, subclass of `Arrow`. This allows the composition of arrows in a subclass to remain within the subclass, without having to rewrite the method `then`. This means we need to make `Arrow.id` a `classmethod` as well so that it can call `upgrade` and return an arrow of the appropriate subclass. We also need to fix

`Box.upgrade = Arrow.upgrade`: when we compose a box then an arrow, we want to return a new arrow object, not a box.

Example 1.1.13. *We can define* `Circuit` *as a subclass of* `Arrow` *and* `Gate` *as a subclass of* `Circuit` *and* `Box` *defined by a name and a number of qubits.*

```
class Circuit(Arrow): pass

class Gate(Box, Circuit):
    upgrade = Circuit.upgrade

    def __init__(self, name: str, n_qubits: int):
        dom, cod = Ob(str(n_qubits)), Ob(str(n_qubits))
        Box.__init__(self, name, dom, cod)

X, Y, Z, H = [Gate(name, n_qubits=1) for name in "XYZH"]

assert (X >> Y) >> Z == X >> (Y >> Z) and X >> Ob('1') == X == Ob('1') >> X
assert isinstance(Circuit.id(Ob('1')), Circuit) and isinstance(X >> Y, Circuit)
```

The `Functor` class listed in 1.1.14 has two mappings `ob` and `ar` as attributes, from objects to objects and from boxes to arrows respectively. The domain of the functor is implicitly defined as the free category generated by the domain of the `ob` and `ar` mappings. The optional arguments `dom` and `cod` allow to define functors with arbitrary categories as domain and codomain, a `Category` is nothing more than a pair of types for its objects and arrows. For now we only use `cod` to define the image of identity arrows, otherwise the (co)domain of the functor is defined implicitly as the (co)domain of the `ob` and `ar` mappings.

We have chosen to implement functors in terms of Python `dict` rather than functions mainly because the syntax looked better for small examples. However, nothing prevents us from making the most of Python's *duck typing*: if it quacks like a `dict` and if it has a `__getitem__` method, we can use it to define functors like a `dict`. Thus, we can define functors with domains that are not finitely generated, such as the identity functor or more concretely the evaluation functor for quantum gates parameterised by a continuous angle. An equivalent solution is to subclass `Functor` and override its `__call__` method directly. The only downside is that we cannot print, save or check equality for such functors, we can only apply them to objects and arrows.

Listing 1.1.14. Implementation of **Cat** with `Category` as objects and `Functor` as arrows.

```python
class DictOrCallable:
    def __class_getitem__(_, source, target):
        return dict[source, target] | Callable[[source], target]

@dataclass
class FakeDict:
    inside: Callable
    __getitem__ = lambda self, key: self.inside(key)

@dataclass
class Functor(Composable):
    ob: dictOrCallable[Ob, Ob]
    ar: dictOrCallable[Box, Ar]
    dom: Category = Category(Ob, Arrow)
    cod: Category = Category(Ob, Arrow)

    def __init__(self, ob, ar, dom, cod):
        ob, ar = (d if hasattr(d, "__getitem__") else FakeDict(d) for d in (ob, ar))
        self.ob, self.ar, self.dom, self.cod = ob, ar, dom, cod

    def __call__(self, other: Ob | Arrow) -> Ob | Arrow:
        if isinstance(other, Ob): return self.ob[other]
        if isinstance(other, Box):
            result = self.ar[other]
            if isinstance(result, self.cod.ar): return result
            # This allows some nice syntactic sugar for the ar mapping.
            return self.cod.ar(result, self(other.dom), self(other.cod))
        if isinstance(other, Arrow):
            base_case = self.cod.ar.id(self(other.dom))
            return base_case.then(*self(box) for box in other.inside)
        raise TypeError

    @classmethod
    def id(cls, x: Category) -> Functor:
        return cls(lambda obj: obj, lambda box: box, dom=x, cod=x)

    @inductive
    def then(self: Functor, other: Functor) -> Functor:
        assert self.cod == other.dom
        ob, ar = (
            {x: g[f[x]] for x in f} if hasattr(f, "__iter__") else (lambda x: g[f[x]])
            for f, g in [(self.ob, other.ob), (self.ar, other.ar)])
        return type(self)(ob, ar, self.dom, other.cod)
```

Example 1.1.15. *A typical DisCoPy script starts by defining objects and boxes:*

```
x, y, z = map(Ob, "xyz")
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', z, x)
```

We can define a simple relabeling functor from the free category to itself:

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

We can interpret our arrows as Python functions:

```
G = Functor(
    ob={x: int, y: Iterable, z: int},
    ar={f: range, g: sum, h: lambda n: n * (n - 1) // 2},
    cod=Category(type, Function))
assert G(f >> g)(42) == G(h)(42) == 861
```

We can interpret our arrows as matrices:

```
H = Functor(
    ob={x: 1, y: 2, z: 2},
    ar={f: [[0, 1]], g: [[0, 1], [1, 0]], h: [[1], [0]]},
    cod=Category(int, Matrix))
assert H(f >> g) == H(h).transpose()
```

We can even define functors into **Cat**, *i.e. interpret arrows as functors:*

```
I = Functor(
    ob={x: Category(Ob, Arrow), y: Category(Ob, Arrow), z: Category(int, Matrix)},
    ar={f: F, g: H},
    cod=Category(Category, Functor))
assert I(f >> g)(h) == H(F(h)) == H(f)
```

1.1.4 Quotient categories

After free objects, another concept behind DisCoPy is that of a *quotient object*. Again, let's start with the example of a monoid M. Suppose we're given a binary relation $R \subseteq M \times M$, then we can construct a quotient monoid M/R with underlying set the equivalence classes of the smallest congruence generated by R. That is, the smallest relation $(\sim_R) \subseteq M \times M$ such that:

- $x \sim_R y$ for all $(x, y) \in R$,
- $x \sim_R x$ and if $x \sim_R y$ and $y \sim_R z$ then $x \sim_R z$,

- if $x \sim_R x'$ and $y \sim_R y'$ then $x \times y \sim_R x' \times y'$.

The first point says that $R \subseteq (\sim_R)$. The second says that $(\sim_R)$ is an equivalence relation. The third says that $(\sim_R)$ is closed under products, it is equivalent to the substitution axiom: if $x \sim_R y$ then $axb \sim_R ayb$ for all $a, b \in M$. Explicitly, the congruence $(\sim_R)$ can be constructed in two steps: first, we define the rewriting relation $(\to_R) \subseteq M \times M$ where $axb \to_R ayb$ for all $(x, y) \in R$ and $a, b \in M$. Second, we define $(\sim_R)$ as the *symmetric, reflexive, transitive closure* of the rewriting relation, i.e. two elements $x, y \in M$ are equal in M/R iff they are in the same connected component of the undirected graph induced by $(\to_R) \subseteq M \times M$. Now there is a homomorphism $q : M \to M/R$ which sends monoid elements to their equivalence class with the following property: for any homomorphism $f : M \to N$ with $x \sim_R y$ implies $f(x) = f(y)$, there is a unique $f' : M/R \to N$ with $f = q \fatsemi f'$. Intuitively, a homomorphism from a quotient M/R is nothing more than a homomorphism from M which respects the axioms R. Up to isomorphism, we can construct any monoid M as the quotient $X^\star/R$ of a free monoid $X^\star$: take $X = U(M)$ and $R = \{(xy, z) \in X^\star \times X^\star \mid x \times y = z \in M\}$.

The pair $(X, R \subseteq X^\star \times X^\star)$ of a set of generating elements X and a binary relation R on its free monoid is called a *presentation* of the monoid $M \simeq X^\star/R$. Arguably, the most fundamental computational problem is the *word problem for monoids*: given a presentation (X, R) and a pair of lists $x, y \in X^\star$, decide whether $x = y$ in $X^\star/R$. As mentioned in the introduction, it was shown to be equivalent to Turing's halting problem, and thus undecidable, by Post [Pos47] and Markov [Mar47]. The proof is straightforward: we can encode the tape alphabet and the states of a Turing machine in the set X and its transition table into the relation R, then whether the machine halts reduces to deciding $x = y$ for x and y the initial and accepting configurations respectively: a proof of equality corresponds precisely to a run of the Turing machine.

The case of quotient categories is similar, only we need to take care of objects now. Given a category C and a family of binary relations $\{R_{x,y} \subseteq C(x, y) \times C(x, y)\}_{x,y \in C_0}$, we can construct a quotient category C/R with equivalence classes as arrows. There is a functor $Q : C \to C/R$ sending each arrow to its equivalence class, and for any functor $F : C \to D$ with $(f, g) \in R_{x,y}$ implies $F(f) = F(g)$, there is a unique $F' : C/R \to D$ with $F = Q \fatsemi F'$. Intuitively, a functor from a quotient category C/R is nothing more than a functor from C which respects the axioms R. Again, any small category C is isomorphic to the quotient $F(\Sigma)/R$ of a free category $F(\Sigma)$: take $\Sigma = U(C)$ and $R = \{(f \fatsemi g, h) \in F(\Sigma) \times F(\Sigma) \mid f \fatsemi g = h \in C\}$.

The pair $(\Sigma, R \subseteq \coprod_{x,y \in \Sigma_0} \Sigma(x,y) \times \Sigma(x,y))$ is called a presentation of the category $C \simeq F(\Sigma)/R$. Since monoids are just categories with one object, the word problem for categories will be just as undecidable as for monoids.

What does it mean to implement a quotient category in Python? Since presentations of categories are as expressive as Turing machines, we might as well avoid solving the halting problem and just use a Python function to define equality of arrows. Implementing a quotient category is nothing more than implementing a free category and an equality function that respects the axioms of a congruence. One straightforward way is to define equality of arrows f, g in a free category $F(\Sigma)$ to be the equality of their interpretation $[\![f]\!] = [\![g]\!]$ under a functor $[\![-]\!] : F(\Sigma) \to D$ into a concrete category D where equality is decidable. Another method is to define a *normal form* method which takes an arrow and returns the representative of its equivalence class, then identity of arrow is identity of their normal forms.

Example 1.1.16. *Take the signature Σ with one object $\Sigma_0 = \{1\}$ and four arrows $\Sigma_1 = \{Z, X, H, -1\}$ for the Z, X and Hadamard gate and the global (-1) phase. Let's define the relation R induced by:*

- $H \fatsemi X = Z \fatsemi H$ *and* $Z \fatsemi X = (-1) \fatsemi X \fatsemi Z$,
- $f \fatsemi f = \mathtt{id}(1)$ *and* $f \fatsemi (-1) = (-1) \fatsemi f$ *for all* $f \in \Sigma_1$.

The quotient $F(\Sigma)/R$ is a subcategory of the category **Circ** *of quantum circuits, it is isomorphic to the quotient induced by the interpretation $[\![-]\!] : F(\Sigma) \to \mathbf{Mat}_{\mathbb{C}}$. Suppose we're given a functor $\mathtt{cost} : F(\Sigma) \to \mathbb{R}^+$, we can define the normal form of a circuit f to be the representative of its equivalence class with the lowest cost. Thus, deciding equality of circuits reduces to solving circuit optimisation perfectly.*

1.1.5 Daggers, sums and bubbles

We conclude this section by discussing three extra pieces of implementation beyond the basics of category theory: daggers, sums and bubbles. A *dagger* for a category C can be thought of as a kind of time-reversal for arrows. More precisely, a dagger is a contravariant endofunctor $\dagger : C \to C^{op}$, i.e. from the category to its opposite with `dom` and `cod` swapped, which is the identity on objects and an involution, i.e. $(\dagger) \fatsemi (\dagger) = \mathtt{id}_{\mathbb{C}}$. A $\dagger$-functor is a functor between $\dagger$-categories that commutes with the dagger, thus we get a category $\dagger - \mathbf{Cat}$. The free $\dagger$-category is constructed as follows. Define the functor $\dagger : \mathbf{Sig} \to \mathbf{Sig}$ which sends a signature Σ to $\dagger(\Sigma)$ with $\dagger(\Sigma)_0 = \Sigma_0$ and $\dagger(\Sigma)_1 = \{-1, 1\} \times \Sigma_1$ with $\mathtt{dom}(b, f) = \mathtt{cod}(f)$ if $b = -1$ else $\mathtt{dom}(f)$

and symmetrically for `cod`. Then the free dagger category is the quotient category $F(\dagger(\Sigma))/R$ for the congruence generated by $(1, f) \fatsemi (-1, f) \to_R \texttt{id}(\texttt{dom}(f))$ and $(-1, f) \fatsemi (1, f) \to_R \texttt{id}(f.cod)$.

Example 1.1.17. *The conjugate transpose defines a dagger on the category* $\mathbf{Mat}_{\mathbb{C}}$*, the adjoint defines a dagger on the category* **Circ** *and the evaluation* **Circ** $\to$ $\mathbf{Mat}_{\mathbb{C}}$ *is a* $\dagger$*-functor. By extension, there is a dagger structure on* $\mathbf{Mat}_{\mathbb{S}}$ *for each rig anti-homomorphism* $\dagger : \mathbb{S} \to \mathbb{S}$*, i.e. a homomorphism for the commutative addition and an anti-homomorphism for the (non-necessarily commutative) product* $\dagger(a \times b) = \dagger(b) \times \dagger(a)$*. Thus, when* $\mathbb{S}$ *is a commutative rig such as the Boolean,* $\mathbf{Mat}_{\mathbb{S}}$ *is automatically a* $\dagger$*-category with the transpose as dagger and the identity as conjugation.*

DisCoPy implements free †-categories by adding an attribute `is_dagger: bool` to boxes and a method `Arrow.dagger`, shortened to the postfix operator `[::-1]`, which reverses the order of boxes and negates `is_dagger` elementwise. The normal form is computable in linear time but it has not been implemented yet. In order to implement the syntactic sugar `f[::-1] == f.dagger()`, we need to override the `__getitem__` method. In general, DisCoPy defines indexing `f[i]` and slicing `f[start:stop:step]` so that `f[key].inside == f.inside[key]` for any `key: int` and any `key: slice` with `key.step in (-1, 1, None)`. Although the case of negative indices (i.e. counting backwards from the end of the list) is implemented in DisCoPy, its interaction with list reversal is too complex to be listed here.

Listing 1.1.18. Implementation of free †-categories and †-functors.

```
class Arrow(cat.Arrow):
    def dagger(self):
        return self.upgrade(Arrow(
            self.cod, self.dom, [box.dagger() for box in self.inside[::-1]]))

    def __getitem__(self, key: int | slice) -> Arrow:
        if isinstance(key, int): return self.upgrade(self.inside[key])
        if key.step not in (-1, 1, None): raise IndexError
        if key.step == -1:
            for i in (key.start, key.stop):
                if i is not None and i < 0: ...
            stop, start = (
                None if i is None else i + 1 for i in (key.stop, key.start))
            return self[start:stop].dagger()
        dom, cod = self[key.start].dom, self[key.stop].cod
        return self.upgrade(Arrow(dom, cod, self.inside[key]))
```

```
class Box(cat.Box, Arrow):
    upgrade = Arrow.upgrade

    def __init__(self, name: str, dom: Ob, cod: Ob, is_dagger=False):
        self.is_dagger = is_dagger; cat.Box.__init__(self, name, dom, cod)

    def dagger(self):
        return Box(self.name, self.cod, self.dom, not self.is_dagger)

class Functor(cat.Functor):
    dom = cod = Category(Ob, Arrow)

    def __call__(self, other):
        if isinstance(other, Box) and other.is_dagger:
            return self(other.dagger()).dagger()
        return super().__call__(other)
```

Example 1.1.19. *We can show the dagger is indeed a contravariant endofunctor.*

```
x, y, z = map(Ty, "xyz")
f, g = Box('f', x, y), Box('g', y, z)

assert Diagram.id(x)[::-1] == Diagram.id(x)
assert (f >> g)[::-1] == g[::-1] >> f[::-1]
```

Listing 1.1.20. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ as a †-category.

```
def transpose(self: Matrix) -> Matrix:
    inside = [[self[j][i] for j in range(self.dom)] for i in range(self.cod)]
    return type(self)(inside, self.cod, self.dom)

def map(self: Matrix, func: Callable[[Number], Number]) -> Matrix:
    inside = [list(map(func, row)) for row in self.inside]
    return type(self)(inside, self.dom, self.cod)

Matrix.transpose, Matrix.map = transpose, map
Matrix.conjugate = lambda self: self.map(lambda x: x.conjugate())
Matrix.dagger = lambda self: self.conjugate().transpose()
```

Example 1.1.21. *We can implement a simulator for* 1*-qubit circuits as a* †*-functor.*

```
Circuit.eval = Functor(
    ob={Ob('1'): 2},
```

```
    ar={X: [[0, 1], [1, 0]],
        Y: [[0, -1j], [1j, 0]],
        Z: [[1, 0], [0, -1]],
        H: [[x / sqrt(2) for x in row] for row in [[1, 1], [1, -1]]]}
    cod=Category(int, Matrix[complex]))
```

We can check that every circuit is unitary*, i.e. its dagger is also its inverse.*

```
for c in [X, Y, Z, H, X >> Y >> Z >> H]:
    assert (c >> c[::-1]).eval() == Matrix.id(2) == (c[::-1] >> c).eval()
```

We can check the equations given in the presentation of example 1.1.16.

```
assert (Z >> H).eval() == (H >> X).eval()
assert (Z >> X).eval() == (X >> Z).eval().map(lambda x: -x)
for gate in [H, Z, X]: assert (gate >> gate).eval() == Matrix.id(2)
```

A category C has *sums*, or equivalently C is *commutative-monoid-enriched*, when it comes equipped with a commutative monoid $(+, 0)$ on each homset $C(x, y)$ such that $f \fatsemi 0 = 0 = 0 \fatsemi f$ and $(f + f') \fatsemi (g + g') = f \fatsemi g + f \fatsemi g' + f' \fatsemi g + f' \fatsemi g'$ for all arrows f, g, f', g'. A functor $F : C \to D$ between categories with sums is commutative-monoid-enriched when $F(0) = 0$ and $F(f + g) = F(f) + F(g)$. For example, the category $\mathbf{Mat}_{\mathbb{S}}$ has sums given by elementwise addition of matrices. A commutative-monoid-enriched category with one object is precisely a rig. Given a signature Σ, we construct the free category with sums $F^+(\Sigma)$ by taking the free commutative monoid over each homset of $F(\Sigma)$, i.e. arrows $f : x \to y$ in $F^+(\Sigma)$ are *bags* (also called *multisets*) of arrows $f_i : x \to y$ in $F(\Sigma)$.

In DisCoPy, free categories with sums are implemented by `Sum`, a subclass of `Box` with an attribute `terms: list[Arrow]` as well as its own `upgrade` method, which turns an arrow into the sum of just itself. It is attached to the arrow with `Arrow.sum = Sum`. The composition of a sum as the sum of the compositions of its terms, we also override `Arrow.then` so that `f >> (g + h) == Sum.upgrade(f) >> (g + h)` for any arrow `f`. We define equality so that `f == Sum.upgrade(f)`, equality of bags of terms is implemented as equality of lists sorted by an arbitrary ordering. DisCoPy functors are commutative-monoid-enriched, i.e. a formal sum of arrows can be interpreted as a concrete sum of matrices.

Listing 1.1.22. Implementation of free sum-enriched categories and functors.

```
class Arrow(cat.Arrow):
    def __eq__(self, other):
        return other.terms == [self] if isinstance(other, Sum)\
```

```
            else super().__eq__(other)

    def then(self, other: Arrow) -> Arrow:
        return self.sum.upgrade(self).then(other) if isinstance(other, Sum)\
            else super().then(other)

    @staticmethod
    def zero(dom: Ob, cod: Ob) -> Arrow: return Sum([], dom, cod)

    __add__ = lambda self, other: self.sum.upgrade(self) + other
    __lt__ = lambda self, other: hash(self) < hash(other)  # An arbitrary order.

class Sum(cat.Box, Arrow):
    def __init__(self, terms: list[Arrow], dom: Ob, cod: Ob):
        assert all(f.dom == dom and f.cod == cod for f in terms)
        self.terms, name = terms, "Sum({}, {}, [{}])".format(
            dom, cod, ", ".join(map(str, terms)))
        cat.Box.__init__(self, name, dom, cod)

    def __eq__(self, other):
        if isinstance(other, Sum):
            return (self.dom, self.cod, sorted(self.terms))\
                == (other.dom, other.cod, sorted(other.terms))
        return self.terms == [other]

    def __add__(self, other):
        if not isinstance(other, Sum): return self + self.upgrade(other)
        return Sum(self.terms + other.terms, self.dom, self.cod)

    @classmethod
    def upgrade(cls, old: cat.Arrow) -> Sum:
        return old if isinstance(old, cls) else cls([old], old.dom, old.cod)

    @inductive
    def then(self, other):
        terms = [f.then(g) for f in self.terms for g in self.upgrade(other).terms]
        return type(self)(terms, self.dom, other.cod)

Arrow.sum = Sum

class Functor(cat.Functor):
    dom = cod = Category(Ob, Arrow)

    def __call__(self, other):
```

```
        if isinstance(other, Sum):
            unit = self.cod.ar.zero(self(other.dom), self(other.cod))
            return sum([self(f) for f in other.terms], unit)
        return super().__call__(other)
```

Listing 1.1.23. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ as a category with sums.

```
class Matrix:
    ...
    def __add__(self: Matrix, other: Matrix) -> Matrix:
        inside = [[x + y for x, y in zip(u, v)]
                  for u, v in zip(self.inside, other.inside)]
        return type(self)(inside, self.dom, self.cod)

    @classmethod
    def zero(cls, dom: int, cod: int) -> Matrix:
        return cls([[0 for _ in range(cod)] for _ in range(dom)], dom, cod)
```

A *bubble* on a (subcategory of a) category C is a pair of unary operators $\beta_{\mathtt{dom}}, \beta_{\mathtt{cod}} : C_0 \to C_0$ on objects and a unary operator between homsets $\beta : C(x, y) \to C(\beta_{\mathtt{dom}}(x), \beta_{\mathtt{cod}}(y))$ for (some) pairs of objects $x, y \in C_0$. Given a signature Σ and a pair $\beta_{\mathtt{dom}}, \beta_{\mathtt{cod}} : C_0 \to C_0$, we construct the free category with bubbles $F(\Sigma^\beta)$ by induction on the maximum level n of bubble nesting: take the signature $\Sigma^\beta = \bigcup_{n\in\mathbb{N}} \Sigma^\beta_n$ for $\Sigma^\beta_0 = \Sigma$ and $\Sigma^\beta_{n+1} = \Sigma + \{\beta(f) \mid f \in F(\Sigma^\beta_n)\}$. That is, box in Σ^β is a box in Σ^β_n for some $n \in \mathbb{N}$. A box in Σ^β_n is either a box in Σ or an arrow $f : x \to y$ in $F(\Sigma^\beta_{n-1})$ that we have put inside a bubble $\beta(f) : \beta_{\mathtt{dom}}(x) \to \beta_{\mathtt{cod}}(y)$.

Example 1.1.24. *Any endofunctor $\beta : C \to C$ also defines a bubble, thus we can define a bubble-preserving functor $F(U(C)^\beta) \to C$ which interprets bubbles as functor application. Any functor between two categories C and D defines a bubble on their disjoint union $C + D$ (i.e. with objects $C_0 + D_0$ and arrows $C_1 + D_1$). These functor bubbles have also been called* functorial boxes *[Mel06].*

Example 1.1.25. *An* exponential rig *is a one-object category $\mathbb{S}$ with sums and a bubble* $\exp : \mathbb{S} \to \mathbb{S}$ *which is a homomorphism from sum to product, i.e.* $\exp(a+b) = \exp(a)\exp(b)$ *and* $\exp(0) = 1$. *Any rig $\mathbb{S}$ is an exponetial rig by taking* $\exp(a) = 1$ *for all $a \in \mathbb{S}$. Non-trivial examples include the complex numbers as well as the Boolean rig with negation. Thus, exponential rigs provide enough syntax to define the matrices of most quantum gates, as well as any propositional logic formula.*

Example 1.1.26. *Matrix exponential is a bubble on the subcategory of square matrices, with the property that* $\exp(f+g) = \exp(f) \fatsemi \exp(g)$ *whenever* $f \fatsemi g = g \fatsemi f$. *Also, any function* $\mathbb{S} \to \mathbb{S}$ *yields a bubble on* $\mathbf{Mat}_{\mathbb{S}}$ *given by element-wise application. For example, we can define a bubble on the category* $\mathbf{Mat}_{\mathbb{B}}$ *of Boolean matrices which sends each matrix* f *to its entrywise negation* $\bar{f}$.

DisCoPy implements free bubbles with `Bubble`, a subclass of `Box` which we attach to the arrow class with `Arrow.bubble = Bubble`. `Bubble` has attributes `inside: Arrow`, `dom: Ob` and `cod: Ob` as well as `name: str`. DisCoPy functors interpret bubbles with `name = "method"` as the application of `method` in the codomain category. The resulting syntax with bubbles is strictly more expressive than that of free categories alone. For example, element-wise negation cannot be expressed as a composition: there is no matrix $N : x \to x$ in $\mathbf{Mat}_{\mathbb{B}}$ such that $N \fatsemi f = \bar{f}$ for all $f : x \to y$. This is also the case for the element-wise application of any non-linear function such as the rectified linear units (ReLU) used in machine learning. As we will discuss in Chapter 2.5, differentiation of parameterised matrices cannot be expressed as a composition either, but it is a unary operator between homsets, i.e. a bubble.

Listing 1.1.27. Implementation of free categories with bubbles and their functors.

```
class Bubble(Box):
    def __init__(self, inside: Arrow, dom=None, cod=None, name="bubble"):
        self.inside, dom, cod = inside, dom or inside.dom, cod or inside.cod
        name = "Bubble({}, {}, {}, {})".format(inside, dom, cod, name)
        super().__init__(name, dom, cod)

    @property
    def is_id_on_objects(self) -> bool:
        return (self.dom, self.cod) == (self.inside.dom, self.inside.cod)

Arrow.bubble = Bubble

class Functor(cat.Functor):
    def __call__(self, other):
        if isinstance(other, Bubble):
            method = getattr(self.cod.ar, other.name)
            return method(self(other.inside)) if other.is_id_on_objects\
                else method(self(other.inside), self(other.dom), self(other.cod))
        return super().__call__(other)
```

Example 1.1.28. *We can encode the architecture of a neural network as an arrow with sums and bubbles, encoding vector addition and non-linear activation function respectively. The evaluation of the neural network on some input vector for some parameters is given by the application of a sum-and-bubble-preserving functor into* $\mathbf{Mat}_{\mathbb{R}}$. *The hyper-parameters (i.e. the number of neurons at each layer) are given by the image of the functor on objects.*

```
Matrix.ReLU = lambda self: self.map(lambda x: max(x, 0))

vector, bias = Box('vector', x, y), Box('bias', x, x)
ones, weights = Box('ones', x, y), Box('weights', y, z)
network = ((vector + (bias >> ones)) >> weights).bubble(name="ReLU")

F = Functor(
    ob={x: 1, y: 4, z: 2},
    ar={vector: [[1.2, -2.3, 3.4, -4.5]],
        bias: [[-3.14]], ones: [[1, 1, 1, 1]]
        weights: [[5.6, -6.7, 7.8, -8.9],
                  [9.0, -0.1, 2.3, -3.4]]},
    cod=Category(int, Matrix[float]))

assert F(network) == F(vector).map(lambda x: x + F(bias))\
                              .then(F(weights)).map(lambda x: max(0, x))
```

Example 1.1.29. *We can implement propositional logic with boxes as propositions, composition as conjunction, sum as disjunction and bubble as negation. The evaluation of a formula in a model corresponds to the application of a sum-and-bubble-preserving functor into* $\mathbf{Mat}_{\mathbb{B}}(1, 1)$.

```
Matrix._not = lambda self: self.map(lambda x: not x)

class Formula(Arrow):
    _not = lambda self: self.bubble(name="_not")

class Proposition(Box, Formula):
    def __init__(self, name): Box.__init__(self, name, Ob('x'), Ob('x'))

def model(data: dict[Proposition, bool]):
    return Functor(ob={Ob('x'): 1}, ar={p: [[data[p]]] for p in data},
                   dom=Category(Ob, Formula), cod=Category(int, Matrix[bool]))

p, q = map(Proposition, "pq")
p_implies_q = (q._not() >> p)._not()
```

```
not_p_or_q = p._not() + q

for a, b in itertools.product([0, 1], [0, 1]):
    F = model({p: a, q: b})
    assert F(p_implies_q) == not (not F(q) and F(p))\
        == F(not_p_or_q) == not F(p) or F(q)
```

Remark 1.1.30. *The constructions for dagger, sums and bubbles all commute with each other. Moreover, there is cube of faithful functors which embed free categories in free †-categories with sums and bubbles. Thus, they are all implemented by default in the same class* `Arrow`.

1.2 Diagrams in Python

1.2.1 Abstract monoidal categories

In the previous section, we introduced the idea of arrows in free categories as abstract data pipelines and functor application as their evaluation in concrete categories such as **Pyth**, **Mat** or **Circ** where the computation happens. For now, our pipelines are rather basic because they are linear: we cannot express functions of multiple arguments, nor tensors of order higher than 2, nor circuits with multiple qubits in any explicit way. In this section, we move from the one-dimensional syntax of arrows in free categories to the two-dimensional syntax of *string diagrams*, the arrows of free *monoidal categories*. The data for a (strict[1]) monoidal category C is that of a category together with: an object $1 \in C_0$ called the *unit* and a pair of overloaded binary operations called the *tensor* on objects $\otimes : C_0 \times C_0 \to C_0$ and on arrows $\otimes : C_1 \times C_1 \to C_1$, translated to `@` in Python. The axioms for monoidal categories are the following:

- $(C_0, \otimes, 1)$ and $(C_1, \otimes, \texttt{id}(1))$ are monoids,
- the tensor defines a functor $\otimes : C \times C \to C$, i.e. the following *interchange law* $(f \fatsemi f') \otimes (g \fatsemi g') = (f \otimes g) \fatsemi (f' \otimes g')$ holds for all arrows $f, f', g, g' \in C_1$.

We will use the following terminology: an object x is called a *system*, an arrow $f : 1 \to x$ from the unit is called a *state* of the system x, an arrow $f : x \to 1$ into the unit is called an *effect* of x and an arrow $a : 1 \to 1$ from the unit to itself is called a *scalar*.

[1] We will assume that our monoidal categories are strict, i.e. the axioms for monoids are equalities rather than natural isomorphisms subject to coherence conditions.

A functor $F : C \to D$ between monoidal categories C and D is (strict[1]) monoidal whenever it is also a monoid homomorphism on objects and arrows. Thus, monoidal categories themselves form a category **MonCat** with monoidal functors as arrows. A transformation $\alpha : F \to G$ between two monoidal functors $F, G : C \to D$ is monoidal itself when $\alpha(x \otimes y) = \alpha(x) \otimes \alpha(y)$ for all objects $x, y \in C$.

Example 1.2.1. *Every monoid M can also be seen as a discrete monoidal category, i.e. with only identity arrows.*

Example 1.2.2. *A monoidal category with one object is a commutative monoid. Indeed in any monoidal category, the interchange law implies that scalars form a commutative monoid, by the following Eckmann-Hilton argument:*

$$\begin{aligned} a \fatsemi b &= 1 \otimes a \fatsemi b \otimes 1 = (1 \fatsemi b) \otimes (a \fatsemi 1) = b \otimes a \\ &= (b \fatsemi 1) \otimes (1 \fatsemi a) = b \otimes 1 \fatsemi 1 \otimes a = b \fatsemi a \end{aligned}$$

Example 1.2.3. *A monoidal category with at most one arrow between any two objects is called a* preordered monoid. *The functoriality axiom implies that the preorder is in fact a pre-congruence, i.e. $a \leq b$ and $c \leq d$ implies $a \times c \leq b \times d$. Given the presentation of a monoid (X, R) with $R \subseteq X^\star \times X^\star$, we can construct a preordered monoid with $(\leq_R) = \bigcup_{n \in \mathbb{N}} R^n \subseteq X^\star \times X^\star$ the (non-symmetric) reflexive transitive closure of R. Thus, the inequality problem (i.e. given two lists $x, y \in X^\star$ and a presentation (X, R), decide whether $x \leq_R y$) is a generalisation of the word problem for monoids.*

Example 1.2.4. *The category* **FinSet** *is monoidal with the singleton 1 as unit and Cartesian product as tensor. Again, this is not a strict monoidal category but it is equivalent to one: take the category with natural numbers $m, n \in \mathbb{N}$ as objects and functions $[m] \to [n]$ as arrows for $[n] = \{0, 1, \ldots, n-1\}$. The states can be identified with elements and the only effect is discarding, i.e. the constant function into the singleton.* **FinSet** *is also monoidal with the empty set 0 as unit and disjoint union as tensor.*

Example 1.2.5. *For any category C, there is a monoidal category C^C where the objects are enfodunctors with composition as tensor and the arrows are natural transformations $\alpha : F \to F'$, $\beta : G \to G'$ with vertical composition $(\alpha \otimes \beta)(x) : G(F(x)) \to G'(F'(x))$ as tensor.*

[1]We will assume that our monoidal functors are strict, i.e. $F(x \otimes y) = F(x) \otimes F(y)$ and $F(1) = 1$ are equalities rather than natural transformations.

1.2.2 Concrete monoidal categories

Example 1.2.6. *The category* **Pyth** *is monoidal with unit* `()` *and* `tuple[t1, t2]` *as the tensor of types* `t1` *and* `t2`*. Given two functions* `f` *and* `g`*, we can define their tensor* `f @ g = lambda x, y: f(x), g(y)`*.*

There are two caveats however. First, **Pyth** *is not strict monoidal:* `(x, (y, z))` *is not strictly equal to* `((x, y), z)` *but only naturally isomorphic, similarly for* `((), x) != x != (x, ())`*. These natural isomorphisms are subject to coherence conditions which make sure that all the ways to rebracket* `(((x, y), z), w)` *into* `(x, (y, (z, w)))` *are the same. In practice, this bureaucracy of parenthesis does not pose any problem: MacLane's coherence theorem [Mac71, p. VII] makes sure that every monoidal category is monoidally equivalent*[1] *to a strict one.*

Second, the interchange law only holds for the subcategory of **Pyth** *with* pure functions *as arrows. Indeed, if the functions* `f` *and* `g` *are impure (e.g. they call* `random` *or* `print`*) then their tensor* `f @ g` *will depend on the order in which they are evaluated, i.e.* `f @ id >> id @ g != id @ g >> f @ id`*. As we will discuss in section 1.5,* **Pyth** *is in fact a* premonoidal category*. The states, i.e. the functions* `f : () -> t`*, can be identified with their value* `f(): t`*. There is only one pure effect, i.e. a unique pure function* `f : t -> ()` *called* discarding*, and thus a unique pure scalar. If we take all impure functions into account, the scalars form a non-commutative monoid of side-effects.*

Listing 1.2.7. Implementation of **Pyth** as a (non-strict pre)monoidal category with `tuple` as tensor.

```
class Function:
    ...
    def tuple(self, other: Function) -> Function:
        dom, cod = tuple[self.dom, other.dom], tuple[self.cod, other.cod]
        return Function(lambda x, y: (self(x), other(y)), dom, cod)
```

Example 1.2.8. *We can also make* **Pyth** *monoidal with the* tagged union *as tensor on objects and* `typing.NoReturn` *as unit. Given two types* `t0`*,* `t1`*, their tagged union* `t0 + t1` *is the union of the types* `tuple[0, t0]` *and* `tuple[1, t1]`[2]*, i.e. a term* `(b, x): t0 + t1` *is a pair of a Boolean* `b: bool` *and a term* `x: t0` *if* `b` *else a term* `x: t1`*. Given two functions* `f`*,* `g` *we can define their tensor as* `lambda b, x:(b, f(x) if b else g(x))`*.*

[1] An *equivalence* of categories is an adjunction where the unit and counit are in fact natural isomorphisms. It is a *monoidal equivalence* when they are also monoidal transformations.

[2] What we really mean is `tuple[Literal[0], t0] | tuple[Literal[1], t1]`.

Listing 1.2.9. Implementation of **Pyth** as a (non-strict) monoidal category with `tagged_union` as tensor.

```
class TaggedUnion:
    _name = "TaggedUnion"

    def __class_getitem__(cls, types):
        if not types: return NoReturn
        result = Union[tuple(tuple[Literal[i], t] for i, t in enumerate(types))]
        result.__origin__, result.__args__ = TaggedUnion, types; return result

class Function:
    ...
    def tagged_union(self: Function, other: Function) -> Function:
        dom, cod = TaggedUnion[self.dom, other.dom], TaggedUnion[self.cod, other.cod]
        return Function(lambda b, x: (b, self(x)) if b else (b, other(x)), dom, cod)
```

Example 1.2.10. *The category* $\mathbf{Mat}_{\mathbb{S}}$ *is monoidal with addition of natural numbers as tensor on objects and the* direct sum $f \oplus g = \begin{pmatrix} f & 0 \\ 0 & g \end{pmatrix}$ *as tensor on arrows. When the rig* $\mathbb{S}$ *is commutative,* $\mathbf{Mat}_{\mathbb{S}}$ *is also monoidal with multiplication of natural numbers as tensor on objects and the* Kronecker product *as tensor on arrows. The inclusion functor* $\mathbf{FinSet} \to \mathbf{Mat}_{\mathbb{B}}$ *is monoidal in two ways: it sends disjoint unions to direct sums and Cartesian products to Kronecker products.*

Listing 1.2.11. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ as a strict monoidal category with `direct_sum` and `Kronecker` as tensor.

```
class Matrix:
    ...
    def direct_sum(self, other: Matrix) -> Matrix:
        dom, cod = self.dom + other.dom, self.cod + other.cod
        left, right = (len(m.inside[0]) if m.inside else 0 for m in (self, other))
        inside = [row + right * [0] if i < len(self.inside) else left * [0] + row
                  for i, row in enumerate(self.inside + other.inside)]
        return type(self)(inside, dom, cod)

    def Kronecker(self, other: Matrix) -> Matrix:
        dom, cod = self.dom * other.dom, self.cod * other.cod
        inside = [[self[i_dom][i_cod] * other[j_dom][j_cod]
            for i_cod in range(self.cod) for j_cod in range(other.cod)]
            for i_dom in range(self.dom) for j_dom in range(other.dom)]
        return type(self)(inside, dom, cod)
```

Example 1.2.12. *The category* **Circ** *is monoidal with addition of natural numbers as tensor on objects and* parallel composition *of circuits as tensor on arrows. The evaluation functor* `eval` : **Circ** $\to$ $\mathbf{Mat}_{\mathbb{C}}$ *is monoidal: it sends the parallel composition of circuits to the Kronecker product of their unitary matrices.*

1.2.3 Free monoidal categories

Now, what does it mean to implement a monoidal category in Python? Again, nothing more than defining a pair of classes for objects and arrows with a `tensor` method that satisfies the axioms. Less trivially, we want to implement the arrows of *free monoidal categories* which can then be interpreted in arbitrary monoidal categories via the application of monoidal functors: this is the content of the `discopy.monoidal` module. As in the case of free categories, free monoidal categories will be the image of a functor F : **MonSig** $\to$ **MonCat**, the left adjoint to the forgetful functor U : **MonCat** $\to$ **MonSig** from monoidal categories to *monoidal signatures*. A monoidal signature Σ is a monoidal category without identity, composition or tensor: a pair of sets Σ_0, Σ_1 and a pair of functions $\texttt{dom}, \texttt{cod} : \Sigma_1 \to \Sigma_0^\star$ from boxes to lists of objects. A morphism of monoidal signatures $f : \Sigma \to \Gamma$ is a pair of functions $f : \Sigma_0 \to \Gamma_0$ and $f : \Sigma_1 \to \Gamma_1$ with $f \fatsemi \texttt{dom} = \texttt{dom} \fatsemi f^\star$ and $f \fatsemi \texttt{cod} = \texttt{cod} \fatsemi f^\star$. Thus, we have defined the category **MonSig** of monoidal signatures and their morphisms. In order to define the forgetful functor U : **MonCat** $\to$ **MonSig**, we will need the following technical lemma.

Definition 1.2.13. *A monoidal category C is* foo *(free on objects) when its monoid of objects $(C_0, \otimes, 1)$ is a free monoid $C_0 = X^\star$ generated by some set of objects X.*

Lemma 1.2.14. *Every monoidal category is monoidally equivalent to a foo one.*

Proof. Given a monoidal category C, we construct C' with objects $C_0^\star$ the free monoid over the objects of C and $C'(x, y) = C(\epsilon_{C_0^\star}(x), \epsilon_{C_0^\star}(y))$ for $\epsilon_{C_0} : C_0^\star \to C_0$ the counit of the list adjunction. That is, an arrow $f : x \to y$ between two lists $x, y \in C_0^\star$ in C' is an arrow $f : \epsilon_{C_0}(x) \to \epsilon_{C_0}(y)$ between their multiplication in C. From left to right, the monoidal equivalence $C \simeq C'$ sends every object $x \in C_0$ to its singleton list $x \in C_0^\star$ and every arrow to itself, from right to left it sends every list to its multiplication and every arrow to itself. □

This means we can take the data for a monoidal category C to be the following:

- a class C_0 of *generating objects* and a class C_1 of arrows,

- domain and codomain functions $\texttt{dom}, \texttt{cod} : C_1 \to C_0^\star$,
- a function $\texttt{id} : C_0^\star \to C_1$ and a (partial) operation $\texttt{then} : C_1 \times C_1 \to C_1$,
- an operation on arrows $\texttt{tensor} : C_1 \times C_1 \to C_1$ with $\texttt{dom}(f \otimes g) = \texttt{dom}(f)\texttt{dom}(g)$ and $\texttt{cod}(f \otimes g) = \texttt{cod}(f)\texttt{cod}(g)$.

The axioms for the objects to be a monoid now come for free, we only need to require that tensor on arrows is a monoid with the interchange law. With this definition of (free-on-objects) monoidal category, we can define the forgetful functor U : **MonCat** $\to$ **MonSig**: it forgets the identity, composition and tensor on arrows, but not the tensor on objects which is free.

Example 1.2.15. *Take a monoid M seen as a discrete monoidal category, we get an equivalent monoidal category M' with objects the free monoid $M^\star$ and an isomorphism $x_1 \dots x_n \to y_1 \dots y_m$ whenever $x_1 \times \cdots \times x_n = y_1 \times \cdots \times y_m$ in M.*

In the cases of monoidal categories where the objects are the natural numbers with addition as tensor, such as **FinSet** with disjoint union, $\mathbf{Mat}_{\mathbb{S}}$ with direct sum or **Circ**, the monoid of objects is already free: $(\mathbb{N}, +, 0)$ is the free monoid generated by the singleton set. These monoidal categories are also called *PROs* (for PROduct categories). When the objects are generated by a more-than-one-element set they are also called *coloured PROs*, this is the same thing as a foo-monoidal category. For example, we can take all the Python types as colours and define **Pyth** as a foo-monoidal category with `list[type]` as objects.

Listing 1.2.16. Implementation of the foo-monoidal category **Pyth** with `list[type]` as objects, `Function` as arrows and `tuple` as tensor.

```
class Tensorable:
    """ Allows syntactic sugar self @ other. """
    def cast(self, other):
        return other if isinstance(other, Tensorable) else self.id(other)

    __matmul__ = lambda self, other: self.tensor(self.cast(other))
    __rmatmul__ = lambda self, other: self.cast(other).tensor(self)

class Function(cat.Function, Tensorable):
    def __init__(self, inside: Callable, dom: list[type], cod: list[type]):
        super().__init__(inside, dom, cod)

    @inductive
    def tensor(self, other: Function) -> Function:
```

```
        def inside(*xs):
            left, right = xs[:len(self.dom)], xs[len(self.dom):]
            result = tuple(self(*left)) + tuple(other(*right))
            return result[0] if len(self.cod + other.cod) == 1 else result
        return Function(inside, self.dom + other.dom, self.cod + other.cod)
```

In the case of $\mathbf{Mat}_{\mathbb{S}}$ with Kronecker product as tensor, we can define an equivalent category $\mathbf{Tensor}_{\mathbb{S}}$ where the objects are lists of natural numbers and the arrows $f : x_1 \dots x_n \to y_1 \dots y_m$ are $(x_1 \times \cdots \times x_n) \times (y_1 \times \cdots \times y_m)$ matrices, i.e. tensors of order $m + n$. Note that we could define yet another equivalent category where the objects are lists of prime numbers instead.

Listing 1.2.17. Implementation of the foo-monoidal category $\mathbf{Tensor}_{\mathbb{S}} \simeq \mathbf{Mat}_{\mathbb{S}}$ with `list[int]` as objects and `Tensor[dtype]` as arrows.

```
def product(x, unit=1): return unit if not x else product(unit * x[0], x[1:])

class Tensor(Tensorable, Matrix):
    def __init__(self, inside: list[list], dom: list[int], cod: list[int]):
        Matrix.__init__(self, inside, dom, cod)

    def downgrade(self) -> Matrix:
        return Matrix[self.dtype](self.inside, product(self.dom), product(self.cod))

    @classmethod
    def id(cls, x: list[int] = []) -> Tensor:
        return cls(Matrix.id(product(x)).inside, x, x)

    @inductive
    def then(self, other: Tensor) -> Tensor:
        inside = Matrix.then(*map(Tensor.downgrade, (self, other))).inside
        return type(self)(inside, self.dom, other.cod)

    @inductive
    def tensor(self, other: Tensor) -> Tensor:
        inside = Matrix.Kronecker(*map(Tensor.downgrade, (self, other))).inside
        return type(self)(inside, self.dom + other.dom, self.cod + other.cod)

    def __getitem__(self, key : int | tuple) -> Tensor:
        if isinstance(key, tuple):
            key = sum(key[i] * product(self.dom[i + 1:]) for i in range(len(key)))
        inside = Matrix.__getitem__(self.downgrade(), key).inside
        dom, cod = ([], cod) if product(self.dom) == 1 else ([], [])
```

```
        return type(self)(inside, dom, cod)

    for attr in ("__bool__", "__int__", "__float__", "__complex__"):
        setattr(Tensor, attr, lambda self: getattr(self.downgrade(), attr)())
```

Now how do we go on constructing the left adjoint $F : \mathbf{MonSig} \to \mathbf{MonCat}$? In the same way that lists in the free monoid $X^\star$ can be defined as equivalence classes of expressions built from generators in X, product and unit, we can construct the arrows of the free monoidal category $F(\Sigma)$ as equivalence classes of expressions built from boxes in Σ_1, identity, composition and tensor. In order to find good representatives for these equivalence classes, we will need the following technical lemma.

Definition 1.2.18. *Given a monoidal signature Σ, we define a signature of* layers *$L(\Sigma)$ with $\Sigma_0^\star$ as objects and triples $(x, f, y) \in \Sigma_0^\star \times \Sigma_1 \times \Sigma_0^\star$ as boxes with $\mathtt{dom}(x, f, y) = x\mathtt{dom}(f)y$ and $\mathtt{cod}(x, f, y) = x\mathtt{cod}(f)y$. Given a morphism of monoidal signatures $f : \Sigma \to \Gamma$, we get a morphism between their signatures of layers $L(f) : L(\Sigma) \to L(\Gamma)$. Thus, we have defined a functor $L : \mathbf{MonSig} \to \mathbf{Sig}$.*

Lemma 1.2.19. *Fix a monoidal signature Σ. Every well-typed expression built from boxes in Σ_1, identity of objects in $\Sigma_0^\star$, composition and tensor is equal to:*

$$\mathtt{id}(x) \text{ for } x \in \Sigma_0^\star \quad \text{or} \quad \mathtt{id}(x_1) \otimes f_1 \otimes \mathtt{id}(y_1) \fatsemi \ldots \fatsemi \mathtt{id}(x_n) \otimes f_n \otimes \mathtt{id}(y_n)$$

for some list of layers $(x_1, f_1, y_1), \ldots, (x_n, f_n, y_n) \in L(\Sigma)$.

Proof. By induction on the structure of well-typed expressions. The only non-trivial case is for the tensor $f \otimes g$ of two expressions $f : x \to y$ and $g : z \to w$, where we need to apply the interchange law to push the tensor through the composition $f \otimes g \;=\; (f \fatsemi \mathtt{id}(y)) \otimes (\mathtt{id}(z) \fatsemi g) \;=\; f \otimes \mathtt{id}(z) \fatsemi \mathtt{id}(y) \otimes g$. □

We have all the ingredients to define the free monoidal category $F(\Sigma)$: it is a quotient $F(L(\Sigma))/R$ of the free category generated by the signature of layers $L(\Sigma)$. Its objects, which we call *types*, are lists in the free monoid $\Sigma_0^\star$. Its arrows, which we call *diagrams*, are paths with lists in $\Sigma_0^\star$ as nodes and layers $(x, f : s \to t, y) \in L(\Sigma)$ as edges $xsy \to xty$. The equality of diagrams is the smallest congruence generated by the *right interchanger*:

$$(axb,\ g,\ c) \fatsemi (a,\ f,\ bwc) \quad \to_R \quad (a,\ f,\ bzc) \fatsemi (ayb,\ g,\ c)$$

for all types $a, b, c \in \Sigma_0^\star$ and boxes $f : x \to y$ and $g : z \to w$. That is, we can interchange two consecutive layers whenever the output of the first box is not

connected to the input of the second, i.e. there is an identity arrow $\texttt{id}(b)$ separating them. Note that for an effect $f : x \to 1$ followed by a state $g : 1 \to y$, we have two options: we can apply the right interchanger $(1, f, 1) \fatsemi (1, g, 1) \to_R (1, g, x) \fatsemi (y, f, 1)$ or its opposite $(1, f, 1) \fatsemi (1, g, 1) \leftarrow_R (x, g, 1) \fatsemi (1, f, y)$. For the composition of two scalars $a : 1 \to 1$ and $b : 1 \to 1$, we can apply interchangers indefinitely $a \fatsemi b \to_R b \fatsemi a \to_R a \fatsemi b$: this is the Eckmann-Hilton argument. Delpeuch and Vicary [DV18] give a quadratic solution to the word problem for free monoidal categories, i.e. deciding when two diagrams are equal.

Theorem 1.2.20 ([DV18])**.** *The equality of diagrams is decidable in linear time in the connected case, and quadratic in the general case. The right interchanger is confluent and for connected diagrams, i.e. when the Eckmann-Hilton argument does not apply, it reaches a normal form in a cubic number of steps.*

Proof. The linear-time algorithm for the connected case works by reduction to isomorphism of planar maps. In the general case, computing the tree of connected components of the diagram takes quadratic time. The cubic worst-case for normal forms is given in example 1.3.7. □

We have defined the equality of diagrams, there remains to define the tensor operation. First, we define the *whiskering* $f \otimes z$ of a diagram f by an object $z \in \Sigma_0^\star$ on the right: we tensor z to the right-hand side of each layer (x_i, f_i, y_i), i.e. $f \otimes z = (x_1, f_1, y_1 z) \fatsemi \cdots \fatsemi (x_n, f_n, y_n z)$ and symmetrically for the whiskering $z \otimes f$ on the left. Then, we can define the tensor $f \otimes g$ of two diagrams $f : x \to y$ and $g : z \to w$ in terms of whiskering $f \otimes g = f \otimes z \fatsemi y \otimes g$. Note that we could have chosen to define $f \otimes g = x \otimes g \fatsemi f \otimes w$, the two definitions are equated by the interchanger.

Given a morphism of monoidal signatures $f : \Sigma \to \Gamma$, we get a monoidal functor $F(f) : F(\Sigma) \to F(\Gamma)$ by relabeling: we have defined a functor $F : \mathbf{MonSig} \to \mathbf{MonCat}$. We now have to show that it is indeed the left adjoint of $U : \mathbf{MonCat} \to \mathbf{MonSig}$. This is very similar to the monoid case. The unit $\eta_\Sigma : \Sigma \to U(F(\Sigma))$ sends objects to themselves and boxes $f : x \to y \in \Sigma$ to diagrams $(1, f, 1) \in L(\Sigma)$, i.e. the layer with empty lists on both sides of f. The counit $\epsilon_C : F(U(C)) \to C$ is the functor which sends diagrams with boxes in C to their evaluation, i.e. the formal composition and tensor of diagrams in $F(U(C))$ is sent to the concrete composition and tensor of arrows in C. In the next section, we will show that this construction is in fact equivalent to the topological definition of diagrams as labeled graphs embedded in the plane.

Listing 1.2.21. Outline of the class `monoidal.Ty`.

```
@dataclass
class Ty(Ob):
    inside: list[Ob | str] = []

    def __init__(self, inside=[]):
        self.inside = [x if isinstance(x, Ob) else Ob(x) for x in inside]
        name = ' @ '.join(map(str, inside)) if inside else "{}()".format(type(self).__name__)
        super().__init__(name)

    @classmethod
    def upgrade(cls, old: Ob) -> Ty:
        if isinstance(old, cls): return old
        return cls(old.inside) if isinstance(old, Ty) else cls(old)

    def tensor(self, *others: Ty) -> Ty:
        if all(isinstance(other, Ty) for other in others):
            return self.upgrade(Ty(
                self.inside + sum([other.inside for other in others], [])))
        return NotImplemented  # This will allow whiskering on the left.

    __matmul__ = tensor
    __getitem__ = lambda self, key: self.upgrade(Ty([self.inside[key]]))
    __pow__ = lambda self, n: self.upgrade(Ty(n * self.inside))
    __len__ = lambda self: len(self.inside)
```

The implementation of the class `Ty` for types (i.e. lists of objects) is straightforward, it is sketched in listing 1.2.21. The only subtlety is in the method `upgrade` which allows the user to subclass `Ty` in a way that the tensor of subclassed objects stays within the subclass, without having to redefine the `tensor` method.

Example 1.2.22. *We can define a* `Qubits` *subclass and be sure that the tensor of qubits is still an instance of* `Qubits`*, not merely* `Ty`*.*

```
class Qubits(Ty):
    def __init__(self, n: int): self.n = n
    objects = property(lambda self: n * [Ob('1')])
    upgrade = staticmethod(lambda old: Qubits(len(old)))
    __str__ = lambda self: "qubit ** {}".format(len(self))

qubit = Qubits(1)
assert qubits ** 0 == Qubits(0) and qubits ** 42 == Qubits(42)
assert isinstance(qubit ** 0, Qubits) and isinstance(qubit ** 42, Qubits)
```

The implementation of `Layer` as a subclass of `cat.Box` is sketched in listing 1.2.23. It has methods `__matmul__` and `__rmatmul__` for whiskering on the right and left respectively, and `upgrade` for turning boxes into layers with units on both sides. Instead of getting the units by calling `Ty()` directly, we use the domain of the box to slice empty types of the appropriate `Ty` subclass. This will prove useful in sections 1.4.1, 1.4.6 and 1.5.4 where we will subclass `Ty` to define the types for free rigid, closed and 2-categories respectively.

Listing 1.2.23. Outline of the class `monoidal.Layer`.

```
class Layer(cat.Box):
    def __init__(self, left: Ty, box: cat.Box, right: Ty):
        self.left, self.box, self.right = left, box, right
        name = ("{} @ ".format(left) if left else "") + box.name\
            + (" @ {}".format(right) if right else "")
        dom, cod = left @ box.dom @ right, left @ box.cod @ right
        super().__init__(name, dom, cod)

    def __matmul__(self, other: Ty) -> Layer:
        return Layer(self.left, self.box, self.right @ other)

    def __rmatmul__(self, other: Ty) -> Layer:
        return Layer(other @ self.left, self.box, self.right)

    @classmethod
    def upgrade(cls, old: cat.Box) -> Layer:
        left, box, right = old.dom[:0], box old.dom[len(old.dom):]
        return old if isinstance(old, cls) else cls(left, box, right)
```

Now we have all the ingredients to define `Diagram` as a subclass of `Arrow` with instances of `Layer` as boxes. The `tensor` method is defined in terms of left and right whiskering of layers. The `interchange` method takes an integer `i < len(self)` and returns the diagram with boxes `i` and `i + 1` interchanged, or raises an `AssertionError` if they are connected. It also takes an optional argument `left: bool` which allows to choose between left and right in case we're interchanging an effect then a state.

The `normal_form` method applies `interchange` until it reaches a normal form, or raises `NotImplementedError` if the diagram is disconnected. The `draw` method renders the diagram as an image, it implements the drawing algorithm discussed in the next section.

Listing 1.2.24. Outline of the class `monoidal.Diagram`.

```python
class Diagram(cat.Arrow, Tensorable):
    def __init__(self, inside: list[Layer], dom: Ty, cod: Ty):
        super().__init__(inside, dom, cod)

    @inductive
    def tensor(self, other: Diagram) -> Diagram:
        layers = [layer @ other.dom for layer in self.inside]
        layers += [self.cod @ layer for layer in other.inside]
        dom, cod = self.dom @ other.dom, self.cod @ other.cod
        return self.upgrade(Diagram(layers, dom, cod))

    def interchange(self, i: int, left=False) -> Diagram: ...
    def normal_form(self, left=False) -> Diagram: ...
    def draw(self, **params): ...
```

Again, we have a class method `upgrade` which takes an old `cat.Arrow` and turns it into a new object of type `cls`, a given subclass of `Diagram`. This means we do not need to repeat the code for identity or composition which is already implemented by `cat.Arrow`. In turn, when the user defines a subclass of `Diagram`, they do not need to repeat the code for identity, composition or tensor. The implementation of `monoidal.Box` as a subclass of `cat.Box` and `Diagram` is relatively straightforward, we only need to make sure that a box is equal to the diagram of just itself. We also want the `upgrade` method of `Box` to be that of `Diagram`.

Listing 1.2.25. Outline of the class `monoidal.Box`.

```python
class Box(cat.Box, Diagram):
    upgrade = Diagram.upgrade

    def __init__(self, name: str, dom: Ty, cod: Ty, **params):
        cat.Box.__init__(self, name, dom, cod, **params)
        Diagram.__init__(self, dom, cod, [Layer.upgrade(self)])

    def __eq__(self, other):
        if isinstance(other, Box): return cat.Box.__eq__(self, other)
        return isinstance(other, Diagram) and other.inside == [Layer.upgrade(self)]
```

Example 1.2.26. *We can define* `Circuit` *as a subclass of* `Diagram`. `Gate` *and* `Ket` *are subclasses of* `Circuit` *and* `Box`. *Now we can compose and tensor gates together and the result will be an instance of* `Circuit`.

```python
class Circuit(Diagram): pass
```

```
class Gate(Circuit, Box):
    def __init__(self, name: str, n_qubits: int):
        Box.__init__(self, name, Qubits(n_qubits), Qubits(n_qubits))

class Ket(Circuit, Box):
    def __init__(self, *bits: bool):
        self.bits, dom, cod = bits, Qubits(0), Qubits(len(bits))
        name = "Ket({})".format(', '.join(map(str, bits)))
        Box.__init__(self, name, dom, cod)

Gate.upgrade = Ket.upgrade = Circuit.upgrade

X, Y, Z, H = [Gate(name, n_qubits=1) for name in "XYZH"]
CX, sqrt2 = Gate("CX", n_qubits=2), Gate("sqrt(2)", n_qubits=0)
assert isinstance(sqrt2 @ Ket(0, 0) >> H @ qubit >> CX, Circuit)
```

The `monoidal.Functor` class is a subclass of `cat.Functor`. It overrides the `__call__` method to define the image of types and layers, and it delegates to its superclass for the image of objects, boxes and composition.

Listing 1.2.27. Implementation of monoidal functors.

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

    def __call__(self, other):
        if isinstance(other, Ty):
            return self.cod.ob(
                sum([self(x) for x in other.inside], self.ob()))
        if isinstance(other, Ob):
            result = self.ob[other] if other in self.ob else self.ob[Ty(other)]
            if isinstance(result, self.cod.ob): return result
            # This allows some nice syntactic sugar for the ob mapping.
            return self.cod.ob([result])
        if isinstance(other, Layer):
            return self(layer.left) @ self(layer.box) @ self(layer.right)
        return super().__call__(other)
```

Note that the domain the dictionary `ob` *can be either an* `Ob` *or a* `Ty` *of length* 1.

Example 1.2.28. *We can simulate quantum circuits by applying a functor from* `Circuit` *to* `Tensor`.

```
class Eval(Functor):
    def __init__(self):
```

```
        ob = {Ob('1'): 2}
        ar = {X: [[0, 1], [1, 0]], Y: [[0, -1j], [1j, 0]], Z: [[1, 0], [0, -1]],
              H: [[x / sqrt(2) for x in row] for row in [[1, 1], [1, -1]]],
              CX: [[i == j for j in [0, 1, 2, 3]] for i in [0, 1, 3, 2]],
              sqrt2: [[sqrt(2)]]}
        super().__init__(ob, ar, cod=Category(tuple[int], Tensor[complex]))

    def __call__(self, other):
        if isinstance(other, Ket):
            if not other.bits: return Tensor.id([])
            head, *tail = other.bits
            return Tensor[complex]([[head, not head]], [], [2]) @ self(Ket(*tail))
        return super().__call__(other)

Circuit.eval = Eval()
circuit = sqrt2 @ Ket(0, 0) >> H @ qubit >> CX
superposition = Ket(0, 0) + Ket(1, 1)
assert circuit.eval() == superposition.eval()
```

Remark 1.2.29. *DisCoPy uses a more compact encoding of diagrams than their list of layers. Indeed, a diagram is uniquely specified by a domain, a list of boxes and a list of* offsets*, i.e. the length of the type to the left of each box.*

```
Encoding = namedtuple("Encoding", ["dom", "boxes_and_offsets"])

Diagram.boxes = property(lambda self: [box for _, box, _ in self.inside])
Diagram.offsets = property(lambda self: [len(left) for left, _, _ in self.inside])

def encode(diagram: Diagram) -> Encoding:
    return Encoding(diagram.dom, zip(diagram.boxes, diagram.offsets))

def decode(encoding: Encoding) -> Diagram:
    diagram = Diagram.id(encoding.dom)
    for box, offset in encoding.boxes_and_offsets:
        left, right = result.cod[:offset], result.cod[offset + len(box.dom):]
        diagram >>= left @ box @ right
    return diagram

x, y, z = map(Ty, "xyz")
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', y @ z, x)
encoding = Encoding(dom=x @ y, boxes_and_offsets=[(f, 0), (g, 1), (h, 0)])
assert decode(encoding) == f @ g >> h and encode(f @ g >> h) == encoding
```

1.2.4 Quotient monoidal categories

Once we have defined freeness, we need to define quotients. The quotient C/R of a monoidal category C by a binary relation $R \subseteq \coprod_{x,y \in C_0^\star} C(x,y) \times C(x,y)$ has the same objects C_0 and arrows equivalence classes of arrows in C_1 under the smallest *monoidal congruence* containing R. A congruence $(\sim_R)$ is monoidal when $f \sim_R f'$ and $g \sim_R g'$ implies $f \otimes g \sim f' \otimes g'$. Explicitly, we can construct C/R as the quotient category for the rewriting relation $\to_R$ where:

$$u \fatsemi b \otimes f \otimes c \fatsemi v \quad \to_R \quad u \fatsemi b \otimes g \otimes c \fatsemi v$$

for all $(f, g) \in R$, $u : a \to b \otimes \mathtt{dom}(f) \otimes c$ and $v : b \otimes \mathtt{cod}(f) \otimes c \to d$. Intuitively, if we can equate f and g then we can equate them in any context, i.e. with any objects b and c tensored on the left and right and any arrows u and v composed above and below. A proof that two diagrams are equal in the quotient can itself be thought of as a diagram in three dimensions, i.e. the movie of a diagram being rewritten into another. These higher-dimensional diagrams will be mentioned in section 1.6. Again, every monoidal category C is isomorphic to the quotient of a free monoidal category $C = F(\Sigma)/R$: take $\Sigma = U(C)$ and the relation $R \subseteq F(U(C)) \times F(U(C))$ given by every binary composition and tensor.

The word problem for categories reduces to that of monoidal categories, indeed the signature of a category can be seen as a monoidal signature where boxes all have domain and codomain of length one. Thus, deciding equality of diagrams in arbitrary quotient monoidal categories is just as undecidable. The implementation of a quotient is nothing more than a subclass of `Diagram` with an equality method that respects the axioms of a monoidal congruence. The easy way is to define equality of diagrams to be equality of their evaluation by a monoidal functor into a category where equality is decidable. The hard way is to define a `normal_form` method which sends every diagram to a chosen representative of its equivalence class. DisCoPy provides some basic tools to define such a normal form: *pattern matching* and *substitution*.

Listing 1.2.30. Implementation of diagram pattern matching and substitution.

```
@dataclass
class Match:
    top: Diagram
    bottom: Diagram
    left: Ty
    right: Ty
```

```
    def subs(self, target):
        return self.top >> self.left @ target @ self.right >> self.bottom

def match(self: Diagram, pattern: Diagram) -> Iterator[Match]:
    for i, layer in enumerate(self.inside + [self.id(self.cod)]):
        for j in range(len(layer.dom) + 1):
            match = Match(
                top=self[:i], bottom=self[i + len(pattern):],
                left=self[i].dom[:j], right=self[i].dom[j + len(pattern.dom):])
            if self == match.subs(pattern): yield match
```

Now implementing a quotient reduces to implementing a *rewriting strategy*, i.e. a function which inputs diagrams and returns either a choice of match or `StopIteration`, then proving that it is *confluent* (i.e. the order in which we pick matches does not matter) and *terminating* (i.e. there are no infinite sequences of rewrites). Our simple pattern matching routine can be extended in several ways. First, it can only find matches on the nose: we could apply interchangers to the diagram until we find a match (with the cubic-time complexity that this implies). Second, it can only find and substitute one match at a time: we could iterate through lists of compatible matches and implement their simultaneous substitution. Third, instead of looking for `pattern` as a subdiagram of `self` directly, we could iterate through the functors `F` such that `F(pattern)` is a subdiagram. This would allow to implement infinite families of equations such as those between quantum gates parameterised by continuous phases.

Why should computer scientists care about such diagram rewriting? One reason is that diagrams are free data structures in the same sense that lists are free: they are a two-dimensional generalisation of lists. Another reason is that they allow an elegant definition of a Turing-complete problem: given a finite monoidal signature Σ and a pair of lists $x, y \in \Sigma_0^\star$, decide whether there is a diagram $f : x \to y$ in $F(\Sigma)$. Indeed, the word problem for monoids (which is equivalent to the halting problem for Turing machines) reduces to the existence problem for diagrams: given the presentation of a monoid $X^\star/R$, take objects $\Sigma_0 = X$ and boxes $\Sigma_1 = R$ with $\texttt{dom}, \texttt{cod} : \Sigma_1 \hookrightarrow X^\star \times X^\star \to X^\star$ the left and right hand-side of each related pair. For any pair $x, y \in X^\star$, we have that $x \leq_R y$ if and only if there is a diagram $f : x \to y$ in $F(\Sigma)$: the preordered monoid generated by the relation R is the preorder collapse of the free monoidal category $F(\Sigma)$. While monoid presentations define *decision problems* (i.e. with a Boolean output), free monoidal categories naturally define *function problems*: given a pair of types, output a diagram.

If we compose the two reductions together, we get a free monoidal category where diagrams are the possible runs of a given Turing machine. Moreover, a monoidal functor from the category of one machine to another corresponds to a reduction between the problems they solve, the domain machine being simulated by the codomain. Thus, we could very well take finite monoidal signatures as our definition of machine and diagrams as our definition of computation: algorithmic complexity is given by the size of signatures, time and space complexity are given by the length and width[1] of diagrams. Now if two-dimensional diagrams encode computations on one-dimensional lists, we can think of three-dimensional diagrams either as computations on two-dimensional data, or as higher-order computations. For example, the optimisation steps of a (classical or quantum) compiler can be thought of as a three-dimensional diagram, with (classical or quantum) circuits as domain and codomain.

Example 1.2.31. *We can simplify quantum circuits using pattern matching.*

```
def simplify(circuit, rules):
    for source, target in rules:
        for match in circuit.match(source):
            return simplify(match.subs(target), rules)
    return circuit

rules = [(Ket(b) >> X, Ket(not b))
         for b in [0, 1]] + [
         (Ket(b0) @ Ket(b1) >> CX, Ket(b0) @ Ket(not b1 if b0 else b1))
         for b0 in [0, 1] for b1 in [0, 1]]
circuit = Ket(0) @ Ket(0) >> X @ qubit >> CX >> qubit @ X

assert simplify(circuit, rules) == Ket(1) @ Ket(0)
```

1.2.5 Daggers, sums and bubbles

As in the previous section, we introduce three extra pieces of implementation: daggers, sums and bubbles. A †-monoidal category is a monoidal category with a dagger (i.e. an identity-on-objects involutive contravariant endofunctor) that is also a monoidal functor, a †-monoidal functor is both a †-functor and a monoidal functor. They are implemented by adding a `dagger` method to the `Layer` class.

[1]The width of a diagram is the maximum width of its layers, which is not preserved by interchangers. In the diagrams generated by Turing machines, we cannot apply interchangers anyway: every box is connected to the next by the head of the machine.

For example, $\mathbf{Tensor}_{\mathbb{S}}$ is †-monoidal with any conjugate transpose as dagger. The category $\mathbf{Mat}_{\mathbb{S}}$ with direct sum as tensor is also †-monoidal.

Listing 1.2.32. Implementation of free †-monoidal categories.

```
class Layer:
    ...
    def dagger(self) -> Layer:
        return Layer(self.left, self.box.dagger(), self.right)
```

A monoidal category is commutative-monoid-enriched when it has sums that distribute over the tensor, i.e. $(f+f')\otimes(g+g') = f\otimes g+f\otimes g'+f'\otimes g+f'\otimes g'$ and $f\otimes 0 = 0 = 0\otimes f$. They are implemented by a adding method a `tensor` method to `Sum`, as well as overriding `Diagram.tensor` so that `f @ (g + h) == Sum.upgrade(f) @ (g + h)` for all diagrams `f`.

Listing 1.2.33. Implementation of free monoidal categories with sums.

```
class Diagram(monoidal.Diagram):
    @inductive
    def tensor(self, other):
        return self.sum.upgrade(self).tensor(other)\
            if isinstance(other, Sum) else super().tensor(other)

class Sum(cat.Sum, Box):
    @inductive
    def tensor(self, other: Sum) -> Sum:
        terms = [f @ g for f in self.terms for g in self.upgrade(other).terms]
        return Sum(terms, self.dom @ other.dom, self.cod @ other.cod)

Diagram.sum = Sum
```

Bubbles for monoidal categories are the same as bubbles for categories, their implementation requires no extra work. As we mentioned in the previous section, bubbles do give us a strictly more expressive syntax however: they can encode operations on arrows that cannot be expressed in terms of composition or tensor.

Listing 1.2.34. Implementation of free monoidal categories with bubbles.

```
class Bubble(cat.Bubble, Box): pass

Diagram.bubble = Bubble
```

Example 1.2.35. *As in example 1.1.24, any monoidal endofunctor $\beta : C \to C$ also defines a bubble on the monoidal category C, we can define a bubble-preserving functor $F(U(C)^\beta) \to C$ which interprets bubbled diagrams as functor application. However, the disjoint union $C + D$ of two monoidal categories does not yield a well-defined monoidal category: we cannot tensor arrows of C with those of D. Thus, the case of monoidal functors $C \to D$ requires diagrams with different colours (for the inside and the outside of the bubble) which we will mention in section 1.4.*

Example 1.2.36. *We can implement the Born rule as a bubble on* `Circ` *interpreted as element-wise squared amplitude. We can also implement any classical post-processing as a bubble.*

```
Bra = lambda *bits: Ket(*bits)[::-1]
Born_rule = lambda x: abs(x) ** 2
Circuit.measure = lambda self: self.bubble(name="squared_amplitude")
Tensor.squared_amplitude = lambda self: self.map(Born_rule)

assert float((Ket(0) >> H >> Bra(0)).measure().eval()) == .5

biased_ReLU = lambda x: max(0, 2 * x - 1)
Circuit.post_process = lambda self: self.bubble(name="non_linearity")
Tensor.non_linearity = lambda self: self.map(biased_ReLU)

circuit = Ket(0, 0) >> H @ qubit >> CX >> Bra(0, 0)
post_processed_circuit = circuit.measure().post_process()
assert float(post_processed_circuit.eval())\
    == biased_ReLU(Born_rule(float(circuit.eval())))
```

Example 1.2.37. *We can implement the formulae of first-order logic using Peirce's* existential graphs. *They are the first historical examples of string diagrams as well as the first definition of first-order logic [BT98; BT00; MZ16; HS20]. Predicates of arity n are boxes with a codomain of length n, if there are more than one generating objects we get a many-sorted logic. The wires of the diagram correspond to variables, open wires in the domain and codomain are free variables, the others are existentially quantified. Thus, the composition of the diagrams $f : x \to y$ and $g : y \to z$ encodes the formula $\exists y\ f(x, y) \wedge g(y, z)$ with two free variables x, z and y bound. The diagram obtained by composing a predicate p with the dagger of a predicate q encodes the formula $\exists x\ p(x) \wedge q(x)$. Bubbles, which Peirce calls* cuts, *encode negation. The evaluation of a formula in a finite model corresponds to the application of a bubble-preserving functor into* $\mathbf{Mat}_{\mathbb{B}}$.

```python
class Formula(Diagram):
    cut = lambda self: self.bubble(name="_not")

class Predicate(Box, Formula):
    def __init__(self, name, dom): Box.__init__(self, name, Ty(), dom)

def model(size: dict[Ty, int], data: dict[Predicate, list[bool]]):
    return Functor(
        ob=size, ar={p: [data[p]] for p in data},
        dom=Category(Ty, Formula), cod=Category(list[int], Tensor[bool]))

x = Ty('x')
dog, god, mortal = [Predicate(name, dom=x) for name in ("dog", "god", "mortal")]
all_dogs_are_mortal = (dog.cut() >> mortal.dagger()).cut()
gods_are_not_mortal = (god >> mortal.dagger()).cut()
there_is_no_god_but_god = god >> (Formula.id(x).cut() >> god.dagger()).cut()

size = {x: 2}

for dogs, gods, mortals in itertools.product(*3 * [itertools.product(*size[x] * [[0, 1]])]):
    F = model(size, {dog: dogs, god: gods, mortal: mortals})
    assert F(all_dogs_are_mortal) == all(
        not F(dog)[i] or F(mortal)[i] for i in range(size[x]))
    assert F(gods_are_not_mortal) == not any(
        F(god)[i] and F(mortal)[i] for i in range(size[x]))
    assert F(there_is_no_god_but_god) == any(F(god)[i] and not any(
        F(god)[j] and j != i for j in range(size[x])) for i in range(size[x]))
```

Note that for now our syntax is somehow limited: we can only write formulae where each variable appears at most twice, once for the source and target of its wire. In section 1.4.3 we will introduce the diagrammatic syntax for arbitrary formulae, essentially by adding explicit boxes for equality.

1.2.6 From tacit to explicit programming

We get to the end of this section and the reader may have noticed that we have not drawn a single diagram yet: drawing will be the topic of the next section. This absence of drawing intends to demonstrate that diagrams are not only a great tool for visual reasoning, they can also be thought of as a *data structure for abstract pipelines*. Monoidal functors then allow to evaluate these abstract pipelines in terms of concrete computation, be it Python functions, tensor operations or

quantum circuits. This abstract programming style, defining programs in terms of composition rather than arguments-and-return-value, is called *point-free* or *tacit programming*. Because of the difficulty of writing any kind of complex program in that way, it has also been called the *pointless style*. DisCoPy provides a `@diagramize` decorator which allows the user to define diagrams using the standard *explicit* syntax for Python functions instead of the point-free syntax. Given `dom: Ty`, `cod: Ty` and `signature: list[Box]` as parameters, it adds to each box a `__call__` method which takes the objects of its domain as input and returns the objects of its codomain.

Example 1.2.38. *We can define quantum circuits as Python functions on qubits.*

```
kets0 = Ket(0, 0)

@diagramize(dom=Qubits(2), cod=Qubits(2), signature=[sqrt2, kets0, H, CX])
def circuit():
    sqrt2(); qubit0, qubit1 = kets0
    return CX(H(qubit0), qubit1)

assert circuit == sqrt2 @ kets0 >> H @ qubit >> CX
```

The underlying algorithm constructs a graph with nodes for each object of the domain and the codomain of each box, as well as of the whole diagram. There is an edge from a codomain node of a box (or a domain node of the whole diagram) to the domain node of another (or a codomain node of the whole diagram) whenever they are connected. There is also a node for each box and an edge from that box node to its domain and codomain nodes. First, we initialise the graph of the identity diagram and feed the objects of its codomain as input to the decorated function. When a box is applied to a list of nodes, it adds edges going into each object of its domain and returns nodes for each object of its codomain. Finally, the return value of the decorated function is taken as the codomain of the whole diagram.

Listing 1.2.39. Translation from `Diagram` to `Graph`.

We use the graph data structure from NetworkX [HSS08].

```
from networkx import Graph

Node = namedtuple('Node', ['kind', 'label', 'i', 'j'])

def diagram2graph(diagram: Diagram) -> Graph:
    graph = Graph()
```

```
    scan = [Node('dom', x, i, -1) for i, x in enumerate(diagram.dom)]
    graph.add_edges(zip(scan, scan))
    for j, (left, box, _) in enumerate(diagram.inside):
        box_node = Node('box', box, -1, j)
        dom_nodes = [Node('dom', x, i, j) for i in enumerate(box.dom)]
        cod_nodes = [Node('cod', x, i, j) for i in enumerate(box.cod)]
        graph.add_edges(zip(scan[len(left): len(left @ box.dom)], dom_nodes))
        graph.add_edges(zip(dom_nodes, len(box.dom) * [box_node]))
        graph.add_edges(zip(len(box.cod) * [box_node], cod_nodes))
        scan = scan[len(left):] + cod_nodes + scan[len(left @ box.dom):]
    graph.add_edges(zip(scan, [
        Node('cod', x, i, len(diagram)) for i, x in enumerate(diagram.cod)]))
    return graph
```

The `graph2diagram` algorithm which translates the resulting graph into a diagram will be covered in the next section. It will allow to automatically read *pictures of diagrams* (i.e. matrices of pixels) and translate them into `Diagram` objects. Note that in order to construct a `monoidal.Diagram` we need to assume *plane graphs* as input, i.e. graphs with an embedding in the plane. This means the `diagramize` method cannot accept functions which swap the order of variables such as `lambda x, y: y, x`. We also need to assume that every codomain node is connected to exactly one domain node. In terms of Python functions, this means we have to use every variable exactly once. In section 1.4 we will discuss the case of diagrams induced by non-planar graphs, with potentially multiple edges between domain and codomain nodes. Listing 1.2.39 shows the implementation of the inverse translation `diagram2graph` which outputs only planar graphs as we will show in the next section by constructing their embedding in the plane, i.e. their drawing.

1.3 Drawing & reading

The previous section defined diagrams as a data structure based on lists of layers, in this section we define *pictures of diagrams*. Concretely, such a picture will be encoded in a computer memory as a bitmap, i.e. a matrix of colour values. Abstractly, we will define these pictures in terms of topological subsets of the Cartesian plane. We first recall the topological definition from Joyal's and Street's unpublished manuscript *Planar diagrams and tensor algebra* [JS88] and then discuss the isomorphism between the two definitions. In one direction, the isomorphism sends a `Diagram` object to its drawing. In the other direction, it reads the picture

of a diagram and translates it into a `Diagram` object, i.e. its domain, codomain and list of layers.

1.3.1 Labeled generic progressive plane graphs

A *topological graph*, also called 1-dimensional cell complex, is a tuple (G, G_0, G_1) of a Hausdorff space G and a pair of a closed subset $G_0 \subseteq G$ and a set of open subsets $G_1 \subseteq P(G)$ called *nodes* and *wires* respectively, such that:

- G_0 is discrete and $G - G_0 = \bigcup G_1$,
- each wire $e \in G_1$ is homeomorphic to an open interval and its boundary is contained in the nodes $\partial e \subseteq G_0$.

From a topological graph G, one can construct an undirected graph in the usual sense by forgetting the space G, taking G_0 as nodes and edges $(x, y) \in G_0 \times G_0$ for each $e \in G_1$ with $\partial e = \{x, y\}$. A topological graph is finite (planar) if its undirected graph is finite (planar, i.e. there is some embedding in the plane).

A *plane graph* between two real numbers $a < b$ is a finite, planar topological graph G with an embedding in $\mathbb{R} \times [a, b]$. We define the domain $\texttt{dom}(G) = G_0 \cap \mathbb{R} \times \{a\}$, the codomain $\texttt{cod}(G) = G_0 \cap \mathbb{R} \times \{b\}$ as lists of nodes ordered by horizontal coordinates and the set $\texttt{boxes}(G) = G_0 \cap \mathbb{R} \times (a, b)$. We require that:

- $G \cap \mathbb{R} \times \{a\} = \texttt{dom}(G)$ and $G \cap \mathbb{R} \times \{b\} = \texttt{cod}(G)$, i.e. the graph touches the horizontal boundaries only at domain and codomain nodes,
- every domain and codomain node $x \in G \cap \mathbb{R} \times \{a, b\}$ is in the boundary of exactly one wire $e \in G_1$, i.e. wires can only meet at box nodes.

A plane graph is *generic* when the projection on the vertical axis $p_1 : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is injective on $G_0 - \mathbb{R} \times \{a, b\}$, i.e. no two box nodes are at the same height. From a generic plane graph, we can get a list $\texttt{boxes}(G) \in G_0^\star$ ordered by height. A plane graph is *progressive* (also called *recumbent* by Joyal and Street) when p_1 is injective on each wire $e \in G_1$, i.e. wires go from top to bottom and do not bend backwards.

From a progressive plane graph G, one can construct a directed graph by forgetting the space G, taking G_0 as nodes and edges $(x, y) \in G_0 \times G_0$ for each $e \in G_1$ with $\partial e = \{x, y\}$ and $p_1(x) < p_1(y)$. We can also define the domain and the codomain of each box node $\texttt{dom}, \texttt{cod} : \texttt{boxes}(G) \to G_1^\star$ with $\texttt{dom}(x) = \{e \in G_1 \mid \partial e = \{x, y\}, p_1(x) < p_1(y)\}$ the wires coming in from the top and $\texttt{cod}(x) = \{e \in G_1 \mid \partial e = \{x, y\}, p_1(x) > p_1(y)\}$ the wires going out to the

bottom, these sets are linearly ordered as follows. Take some $\epsilon > 0$ such that the horizontal line at height $p_1(x) - \epsilon$ crosses each of the wires in the domain. Then list $\texttt{dom}(x) \in G_1^\star$ in order of horizontal coordinates of their intersection points, i.e. $e < e'$ if $p_0(y) < p_0(y')$ for the projection $p_0 : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $y^{(\prime)} = e^{(\prime)} \cap \{p_1(x) - \epsilon\} \times \mathbb{R}$. Symmetrically we define the list of codomain nodes $\texttt{cod}(x) \in G_1^\star$ with a horizontal line at $p_1 + \epsilon$.

A *labeling* of progressive plane graph G by a monoidal signature Σ is a pair of functions from wires to objects $\lambda : G_1 \to \Sigma_0$ and from boxes to boxes $\lambda : \texttt{boxes}(G) \to \Sigma_1$ which commutes with the domain and codomain. From an lgpp (*labeled generic progressive plane*) graph, one can construct a `Diagram`.

Listing 1.3.1. Translation from labeled generic progressive plane graphs to `Diagram`.

```
def read( G,  λ : G_1 → Ty,  λ : boxes(G) → Box ) -> Diagram:
    dom = [ λ(e) for x ∈ dom(G) for e ∈ G_1 if x ∈ ∂e ]
    boxes = [ λ(x) for x ∈ boxes(G) ]
    offsets = [len( G_1 ∩ {p_0(x)} × ℝ ) for x ∈ boxes(G) ]
    return decode(dom, zip(boxes, offsets))
```

1.3.2 From diagrams to graphs and back

In the other direction, there are many possible ways to draw a given `Diagram` as a lgpp graph, i.e. to embed its graph into the plane. Vicary and Delpeuch [DV18] give a linear-time algorithm to compute such an embedding with the following disadvantage: the drawing of a tensor $f \otimes g$ does not necessarily look like the horizontal juxtaposition of the drawings for f and g. For example, if we tensor an identity with a scalar, the wire representing the identity will wiggle around the node representing the scalar. DisCoPy uses a quadratic-time drawing algorithm with the following design decision: we make every wire a straight line and as vertical as possible. We first initialise the lgpp graph of the identity with a constant spacing between each wire, then for each layer we update the embedding so that there is enough space for the output wires of the box before we add it to the graph. The resulting plane graph is then either plotted on the screen using Matplotlib [Hun07] or translated to TikZ [Tan13] code that can be integrated to a LaTeXdocument. All the diagrams in this thesis were drawn using DisCoPy together with TikZiT[1]

[1] `https://tikzit.github.io/index.html`

for manual editing.

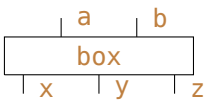
Listing 1.3.2. Outline of `Diagram.draw` from `Diagram` to `PlaneGraph`.

```
Embedding = dict[Node, tuple[float, float]]
PlaneGraph = tuple[Graph, Embedding]

def draw(self: Diagram) -> PlaneGraph:
    graph = diagram2graph(self)
    def make_space(scan: list[Node], box: Box, offset: int) -> float:
        """ Update the graph to make space and return the left of the box. """
    box_nodes = [Node('box', box, -1, j) for j, box in enumerate(self.boxes)]
    dom_nodes = scan = [Node('dom', x, i, -1) for i, x in enumerate(self.dom)]
    position = {node: (i, -1) for i, node in enumerate(dom_nodes)}
    for j, (left, box, _) in enumerate(self.inside):
        box_node, left_of_box = Node('box', box, -1, j), make_space(scan, box, offset)
        position[box_node] = (left_of_box + max(len(box.dom), len(box.cod)) / 2, j)
        for kind, epsilon in (('dom', -.1), ('cod', .1)):
            for i, x in enumerate(getattr(box, kind)):
                position[Node(kind, x, i, j)] = (left_of_box + i, j + epsilon)
        box_cod_nodes = [Node('cod', x, i, j) for i, x in enumerate(box.cod)]
        scan = scan[:len(left)] + box_cod_nodes + scan[len(left @ box.dom):]
    for i, x in enumerate(self.cod):
        position[Node('cod', x, i, len(self))] = (position[scan[i]][0], len(self))
    return graph, position

Diagram.draw = draw
```

Note that when we draw the plane graph for a diagram, we do not usually draw the box nodes as points. Instead, we draw them as boxes, i.e. a box node $x \in \texttt{boxes}(G)$ is depicted as the rectangle with corners $(l, p_1(x) \pm \epsilon)$ and $(r, p_1(x) \pm \epsilon)$ for $l, r \in \mathbb{R}$ the left- and right-most coordinate of its domain and codomain nodes. In this way, we do not need to draw the in- and out-going wires of the box node: they are hidden by the rectangle. Exceptions include *spider boxes* where we draw the box node (the head) and its outgoing wires (the legs of the spider) as well as *swap, cup and cap boxes* where we do not draw the box node at all, only its outgoing wires which are drawn as Bézier curves to look like swaps, cups and caps respectively. These special boxes will be discussed, and drawn, in section 1.4.

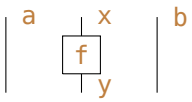
Example 1.3.3. Drawing of a box, an identity, a layer, a composition and a tensor.

```
a, b, c, x, y, z, w = map(Ty, "abcxyzw")
Box('box', a @ b, x @ y @ z).draw()
```

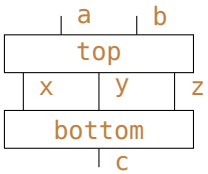



```
Diagram.id(x @ y @ z).draw()
```

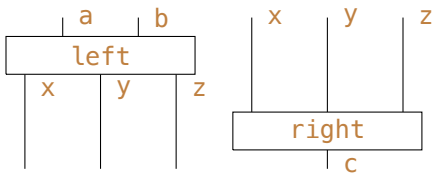

```
layer = a @ Box('f', x, y) @ b
layer.draw()
```




```
top, bottom = Box('top', a @ b, x @ y @ z), Box('bottom', x @ y @ z, c)
(top >> bottom).draw()
```

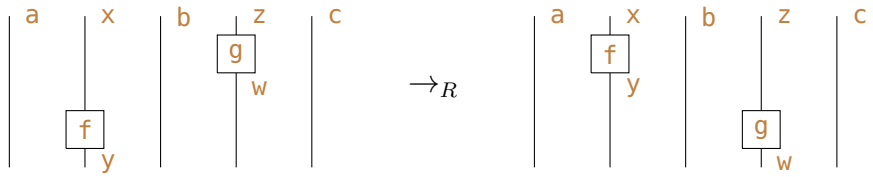



```
left, right = Box('left', a @ b, x @ y @ z), Box('right', x @ y @ z, c)
(left @ right).draw()
```



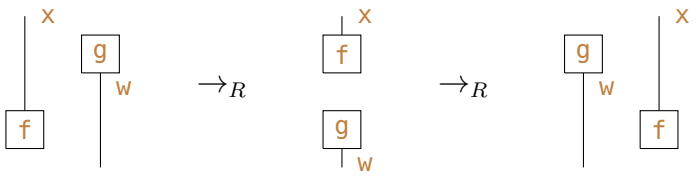
Example 1.3.4. Drawing of the interchanger in the general case.

```
f, g = Box('f', x, y), Box('g', z, w)
(a @ f @ b @ g @ c).interchange(0).draw(); (a @ f @ b @ g @ c).draw()
```



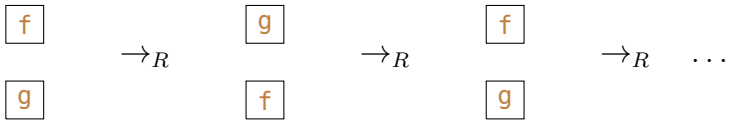
Example 1.3.5. Drawing of the interchangers for an effect then a state.

```
f, g = Box('f', x, Ty()), Box('g', Ty(), w)
(f >> g).interchange(0).draw()
(f >> g).draw()
(f >> g).interchange(0, left=True).draw()
```



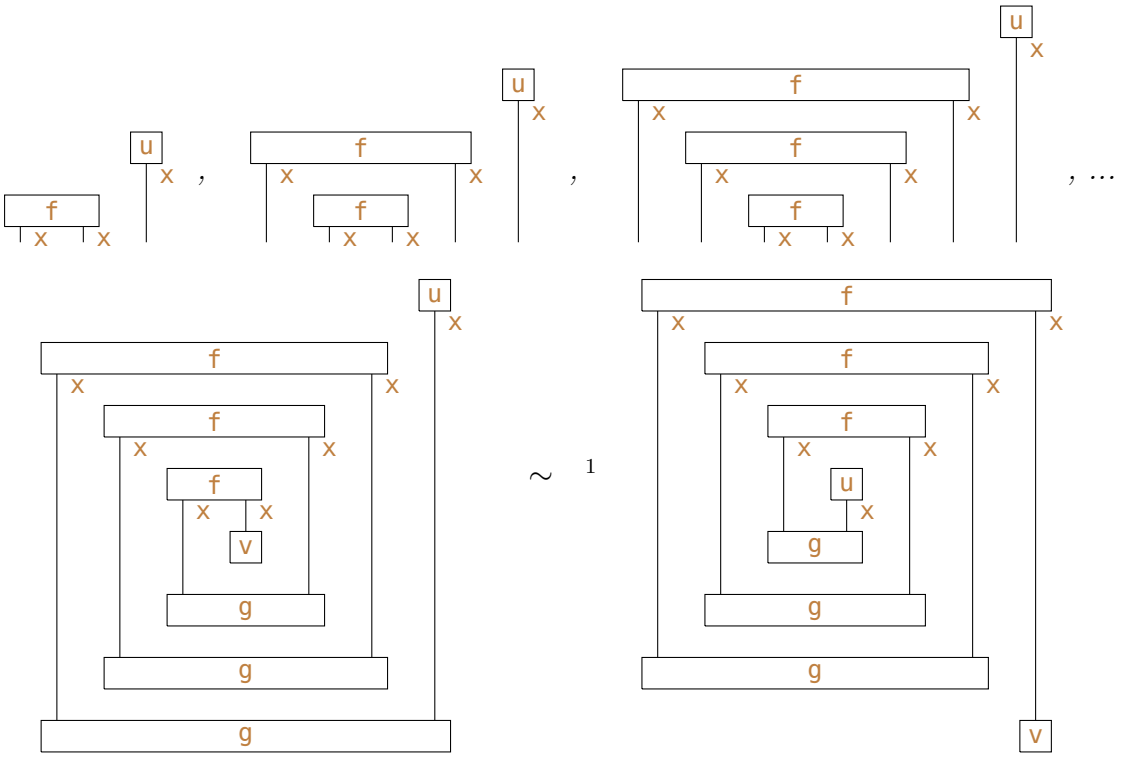
Example 1.3.6. Drawing of the Eckmann-Hilton argument.

```
f, g = Box('f', Ty(), Ty()), Box('g', Ty(), Ty())
(f @ g).draw()
(f @ g).interchange(0).draw()
(f @ g).interchange(0).interchange(0).draw()
```



Example 1.3.7. *The following spiral diagram is the cubic worst-case for interchanger normal form. It is also the quadratic worst-case for drawing, at each layer of the first half we need to update the position of every preceding layer in order to make space for the output wires.*

```
x = Ty('x')
f, g = Box('f', Ty(), x @ x), Box('g', x @ x, Ty())
u, v = Box('u', Ty(), x), Box('v', x, Ty())

def spiral(length: int) -> Diagram:
    diagram, n = u, length // 2 - 1
    for i in range(n): diagram >>= Id(x ** i) @ f @ Id(x ** (i + 1))
    diagram >>= Id(x ** n) @ v @ Id(x ** n)
    for i in range(n): diagram >>= Id(x ** (n - i - 1)) @ g @ Id(x ** (n - i - 1))
    return diagram

diagram = spiral(8)
for i in [1, 2, 3]: diagram[:i + 1].draw()
diagram.draw(); diagram.normal_form().draw()
```



Next, we define the inverse translation `graph2diagram`.

Listing 1.3.8. Translation from `PlaneGraph` to `Diagram`.

```
def graph2diagram(graph: Graph, position: Embedding) -> Diagram:
    dom = Ty(*[node.label for node in graph.nodes if node.kind == 'dom' and node.j == -1])
    boxes = [node.label for node in graph.nodes if node.kind == 'box']
    scan, offsets = [Node('dom', x, i, -1) for i, x in enumerate(dom)], []
    for j, box in enumerate(boxes):
        left_of_box = position[Node('dom', box.dom[0], 0, j)][0]\
            if box.dom else position[Node('box', box, -1, j)][0]
        offset = len([node for node in scan if position[node][0] < left_of_box])
        box_cod_nodes = [Node('cod', x, i, j) for i, x in enumerate(box.cod)]
        scan = scan[:offset] + box_cod_nodes + scan[offset + len(box.dom):]
        offsets.append(offset)
    return decode(dom, zip(boxes, offsets))
```

Theorem 1.3.9. `graph2diagram(self.draw()) == self` *for all* `self: Diagram`.

Proof. By induction on `n = len(self)`. If `len(self) == 0` we get that `dom == self.dom` and `boxes == offsets == []`. If the theorem holds for `self`, it holds for `self >> Layer(left, box,` Indeed, we have:

[1] See `https://github.com/oxford-quantum-group/discopy/blob/main/docs/_static/imgs/spiral.gif` for a proof.

- `dom == self.dom and boxes == self.boxes + [box]`
- `(x, Node('cod', self.cod[i], i, n)) in graph for i, x in enumerate(scan)`

Moreover, the horizontal coordinates of the nodes in `scan` are strictly increasing, thus we get the desired `offsets == self.offsets + [len(left)]`. □

From a labeled generic progressive plane graph, we get a unique diagram *up to deformation*. A deformation $h : G \to G'$ between two labeled plane graphs G, G' is a continuous map $h : G \times [0,1] \to \mathbb{R} \times \mathbb{R}$ such that:

- $h(G,t)$ is a plane graph for all $t \in [0,1]$, $h(G,0) = G$ and $h(G,1) = G'$,
- $x \in \texttt{boxes}(G)$ implies $h(x,t) \in \texttt{boxes}(h(G,t))$ for all $t \in [0,1]$,
- $h(G,t) \fatsemi \lambda = \lambda$ for all $t \in [0,1]$, i.e. the labels are preserved throughout.

A deformation is progressive (generic) when $h(G,t)$ is progressive (generic) for all $t \in [0,1]$. We write $G \sim G'$ when there exists some deformation $h : G \to G'$, this defines an equivalence relation.

Theorem 1.3.10. *For all lgpp graphs G,* `Diagram.draw(graph2diagram( G ))` $\sim G$ *up to generic progressive deformation.*

Proof. By induction on the length of $\texttt{boxes}(G)$. If there are no boxes, G is the graph of the identity and we can deform it so that each wire is vertical with constant spacing. If there is one box, G is the graph of a layer and we can cut it in three vertical slices with the box node and its outgoing wires in the middle. We can apply the case of the identity to the left and right slices, for the middle slice we make the wires straight with a constant spacing between the domain and codomain. Because G is generic, we can cut a graph with $n > 2$ boxes in two horizontal slices between the last and the one-before-last box, then apply the case for layers and the induction hypothesis. To glue the two slices back together while keeping the wires straight, we need to make space for the wires going out of the box.

This deformation is indeed progressive, i.e. we never bend wires we only make them straight. It is also generic, i.e. we never move a box node past another. □

Theorem 1.3.11. *There is a progressive deformation $h : G \to G'$ between two lgpp graphs iff* `graph2diagram( G ) == graph2diagram( G' )` *up to interchanger.*

Proof. By induction on the number n of *coincidences*, the times at which the deformation h fails to be generic, i.e. two or more boxes are at the same height. WLOG (i.e. up to continuous deformation of deformations) this happens at a discrete number of time steps $t_1, \dots, t_n \in [0, 1]$. Again WLOG at each time step there is at most two boxes at the same height, e.g. if there are two boxes moving below a third at the same time, we deform the deformation so that they move one after the other. The list of boxes and offsets is preserved under generic deformation, thus if $n = 0$ then `graph2diagram(` G `) == graph2diagram(` G' `)` on the nose. If $n = 1$, take `i: int` the index of the box for which the coincidence happens and `left: bool` whether it is a left or right interchanger, then `graph2diagram(` G `).interchange(i, left) == graph2diagram(` G' `)`. Given a deformation with $n+1$ coincidences, we can cut it in two time slices with 1 and n coincidences respectively then apply the cases for $n = 1$ and the induction hypothesis.

For the converse, a proof of `graph2diagram(` G `) == graph2diagram(` G' `)`, i.e. a sequence of n interchangers, translates into a deformation with n coincidences. DisCoPy can output these proofs as videos using `Diagram.normalize` to iterate through the rewriting steps and `Diagram.to_gif` to produce a `.gif` file. □

1.3.3 A natural isomorphism

We have established an isomorphism between the class of lgpp graphs (up to progressive deformation) and the class of `Diagram` objects (up to interchanger). It remains to show that this actually forms an isomorphism of monoidal categories. That is for every monoidal signature Σ, there is a monoidal category $G(\Sigma)$ with objects $\Sigma_0^\star$ and arrows the equivalence classes of lgpp graphs with labels in Σ. The domain and codomain of an arrow is given by the labels of the domain and codomain of the graph. The identity $\texttt{id}(x_1 \dots x_n)$ is the graph with wires $(i, a) \to (i, b)$ for $i \leq n$ and $a, b \in \mathbb{R}$ the horizontal boundaries. The tensor of two graphs G and G' is given by horizontal juxtaposition, i.e. take $w = \max(p_0(G)) + 1$ the right-most point of G plus a margin and set $G \otimes G' = G \cup \{(p_0(x) + w, p_1(x)) \mid x \in G'\}$. The composition $G \fatsemi G'$ is given by vertical juxtaposition and connecting the codomain nodes of G to the domain nodes of G'. That is, $G \fatsemi G' = s^+(G) \cup s^-(G') \cup E$ for $s^\pm(x) = \left(p_0(x), \frac{p_1(x) \pm (b-a)}{2}\right)$ and wires $s^+(\texttt{cod}(G)_i) \to s^-(\texttt{dom}(G')_i) \in E$ for each $i \leq \texttt{len}(\texttt{cod}(G)) = \texttt{len}(\texttt{dom}(G'))$.

The deformations for the unitality axioms are straightforward: there is a deformation $G \fatsemi \texttt{id}(\texttt{cod}(G)) \sim G \sim \texttt{id}(\texttt{dom}(G)) \fatsemi G$ which contracts the wires of the identity graph, the unit of the tensor is the empty diagram so we have an equality

$G \otimes \mathtt{id}(1) = G = \mathtt{id}(1) \otimes G$. The deformations for the associativity axioms are better described by the hand-drawn diagrams of Joyal and Street in figure 1.1.

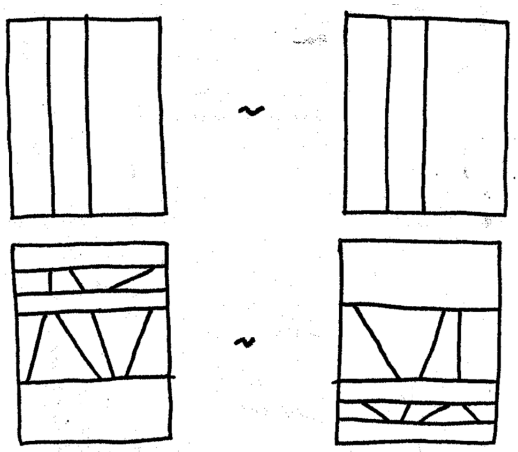


Figure 1.1: Deformations for the associativity of tensor and composition.

The interchange law holds on the nose, i.e. $(G \otimes G') \fatsemi (H \otimes H') = (G \fatsemi H) \otimes (G' \fatsemi H')$, as witnessed by figure 1.2, the hand-drawn diagram which is the result of both sides.

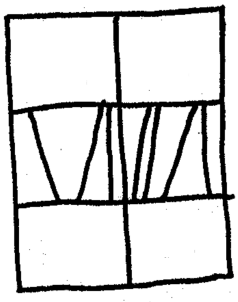


Figure 1.2: The graph of the interchange law.

Thus, we have defined a monoidal category $G(\Sigma)$. Given a morphism of monoidal signatures $f : \Sigma \to \Gamma$, there is a functor $G(f) : G(\Sigma) \to G(\Gamma)$ which sends a graph to itself relabeled with $f \fatsemi \lambda$, its image on arrows is given in listing 1.3.12. Hence, we have defined a functor $G : \mathbf{Monsig} \to \mathbf{MonCat}$ which we claim is naturally isomorphic to the free functor $F : \mathbf{Monsig} \to \mathbf{MonCat}$ defined in the previous section.

Listing 1.3.12. Implementation of the functor $G : \mathbf{Monsig} \to \mathbf{MonCat}$.

```
SigMorph = tuple[dict[Ob, Ob], dict[Box, Box]]


def G(f: SigMorph) -> Callable[[Graph], Graph]:
    def G_of_f(graph: Graph) -> Graph:
```

```
        relabel = lambda node: Node('box', f[1][node.label], node.i, node.j)\
            if node.kind == 'box'\
            else Node(node.kind, f[0][node.label], node.i, node.j)
        return Graph(map(relabel, graph.edges))
    return G_of_f
```

Theorem 1.3.13. *There is a natural isomorphism $F \simeq G$.*

Proof. From theorems 1.3.10 and 1.3.11, we have an isomorphism between `Diagram` and `PlaneGraph` given by `d2g = Diagram.draw` and `g2d = graph2diagram`, up to deformation and interchanger respectively. Now define the image of F on arrows `F = lambda f: Functor(ob=f[0], ar=f[1])`. Given a morphism of monoidal signatures `f: SigMorph` we have the following two naturality squares.

□

1.3.4 Daggers, sums and bubbles

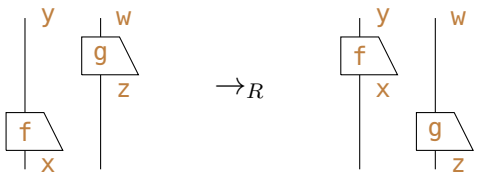
As in the previous sections, we now discuss the drawing of daggers, sums and bubbles. When we draw a diagram in the free †-monoidal category, we add some asymmetry to the drawing of each box so that it looks like the vertical reflection of its dagger.

Example 1.3.14. Drawing of the axiom for unitaries.

```
f, g = Box('f', x, y), Box('g', z, w)
(f >> f[::-1]).draw(); Diagram.id(x).draw(); (f[::-1] >> f).draw(); Diagram.id(y).draw()
```

Example 1.3.15. Drawing of the axiom for †-monoidal categories.

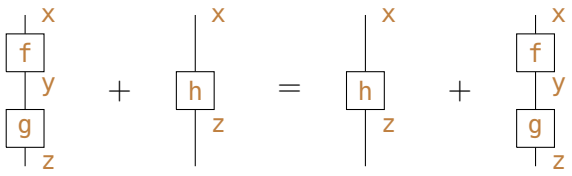
```
f, g = Box('f', x, y), Box('g', z, w)
(f @ g)[::-1].draw(); (f[::-1] @ g[::-1]).draw()
assert (f @ g)[::-1].normal_form() == f[::-1] @ g[::-1]
```



When we draw a sum, we just draw each term with an addition symbol in between. More generally, `drawing.equation` allows to draw any list of diagrams and `drawing.Equation` allows to draw equations within equations.

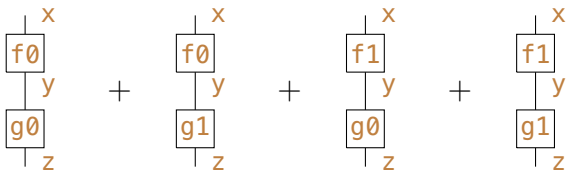
Example 1.3.16. Drawing of a commutativity equation.

```
from discopy import drawing

f, g, h = Box('f', x, y), Box('g', y, z), Box('h', x, z)
drawing.equation(drawing.Equation(f >> g, h, symbol='$+$'),
                 drawing.Equation(h, f >> g, symbol='$+$'))
```

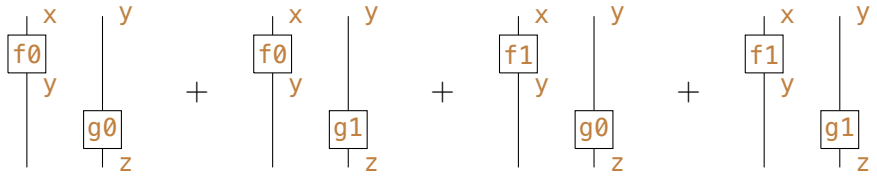


Example 1.3.17. Drawing a composition and tensor of sums.

```
f0, g0 = Box("f0", x, y), Box("g0", y, z)
f1, g1 = Box("f1", x, y), Box("g1", y, z)
((f0 + f1) >> (g0 + g1)).draw()
```



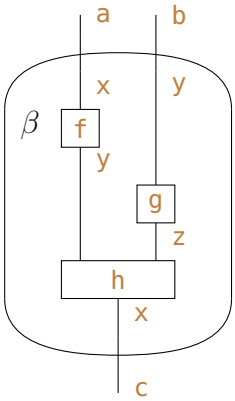

```
((f0 + f1) @ (g0 + g1)).draw()
```



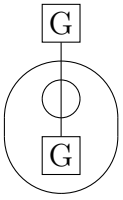
The case of drawing bubbles is more interesting. One solution would be to draw the bubble as a rectangle like any other box, then draw the content of the bubble inside the rectangle. However, this would require some clever scaling so that the boxes of the diagram inside the bubble have the same size as the boxes outside, i.e. we would need to add more complexity to our drawing algorithm. The solution implemented in DisCoPy is to apply a faithful functor `downgrade` : $F(\Sigma^\beta) \to F(\Sigma \cup \texttt{open}^\beta \cup \texttt{close}^\beta)$ from the free monoidal category with bubbles $F(\Sigma^\beta)$ to the free monoidal category generated by the following signature. Take the objects $\texttt{open}_0^\beta = \texttt{close}_0^\beta = \Sigma_0 + \{\bullet\}$ and boxes for opening $\texttt{open}^\beta(x) : \beta_{\texttt{dom}}(x) \to \bullet \otimes x \otimes \bullet$ and closing $\texttt{close}^\beta(x) : \bullet \otimes x \otimes \bullet \to \beta_{\texttt{cod}}(x)$ bubbles for each type $x \in \Sigma_0^\star$. Now define `downgrade(f.bubble())` $= \texttt{open}^\beta(\texttt{f.dom}) \fatsemi (\bullet \otimes \texttt{f} \otimes \bullet) \fatsemi \texttt{close}^\beta(\texttt{f.cod})$ for any diagram `f` inside a bubble. That is, we draw a bubble as its opening, its inside with identity wires on both sides then its closing. The $\bullet$-labeled wires are drawn with Bézier curves so that the bubble looks a bit closer to a circle than a rectangle. In the case of bubbles that are length-preserving on objects, we also want to override the drawing of its opening and closing boxes so that the wires go straight through the bubble rather than meeting at the box node.

Example 1.3.18. Drawing of a bubbled diagram and a first-order logic formula.

```
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', y @ z, x)
(f @ g >> h).bubble(dom=a @ b, cod=c, name="$\\beta$").draw()
```




```
god = Predicate("G", x)
formula = god >> (Formula.id(x).cut() >> god.dagger()).cut()
formula.draw()
```



1.3.5 Automatic diagram recognition

We conclude this section with an application of theorem 1.3.13 to *automatic diagram recognition*: turning pictures of diagrams into diagrams. In listing 1.3.1, we described an abstract reading algorithm which took lgpp graphs as input and returned diagrams. We make it a concrete algorithm by taking *bitmaps* as input: grids of Boolean pixels describing a black-and-white picture. The algorithm `read` listed below takes as input a pair of bitmaps for the box nodes and the wires of the plane graph, it returns a `Diagram`. It is more general than the `graph2diagram` algorithm of listing 1.3.1 where we assumed that the embedding of the graph looked like the output of `Diagram.draw`. i.e. that edges are straight vertical lines. Indeed, our reading algorithm will accept *any bitmaps* as input and always return a valid diagram, however bended the edges are. If the bitmaps indeed represent a progressive generic plane graph G, then we get `read( G ).draw()` $\sim G$ up to progressive generic deformation. If not, the output will still be a diagram but its drawing may not look anything like the input.

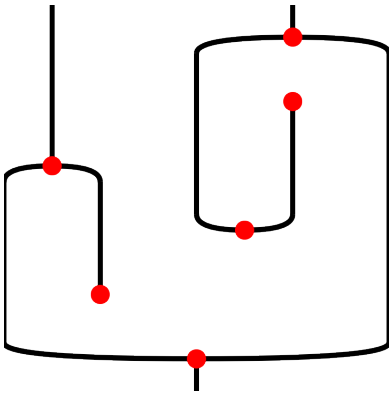
Listing 1.3.19. Implementation of the abstract reading algorithm of listing 1.3.1.

```
from numpy import array, argmin
from skimage.measure import regionprops, label

def read(box_pixels: array, wire_pixels: array) -> Diagram:
    connected_components = lambda img: regionprops(label(img))
    box_nodes, wires = map(connected_components, (box_pixels, wire_pixels))
    source, target, length, width = [], [], len(box_pixels), len(box_pixels[0])
    critical_heights = [0] + [int(node.centroid[0]) for node in box_nodes] + [length]
    for wire, region in enumerate(wires):
        top, bottom = (minmax(i for i, _ in region.coords) for minmax in (min, max))
        source.append(argmin(abs(array(critical_heights) - top)))
        target.append(argmin(abs(array(critical_heights) - bottom)))
    scan = [wire for wire, node in enumerate(source) if node == 0]
    dom, boxes_and_offsets = Ty('x') ** len(scan), []
    for depth, box_node in enumerate(box_nodes):
        input_wires = [wire for wire in scan if target[wire] == depth + 1]
        output_wires = [wire for wire, node in enumerate(source) if node == depth + 1]
        dom, cod = Ty('x') ** len(input_wires), Ty('x') ** len(output_wires)
        box = Box('box_{}_{}'.format(len(dom), len(cod)), dom, cod)
        height, left = map(int, box_node.centroid)
        left_of_box = [wire for wire in scan if wire not in input_wires
                       and dict(wires[wire].coords).get(height, width) < left]
        offset = max(len(left_of_box), 0)
        boxes_and_offsets.append((box, offset))
```

```
            scan = scan[:offset] + output_wires + scan[offset + len(input_wires):]
    return decode(dom, boxes_and_offsets)
```

We use the `array` data structure of NumPy [vdWCV11] for bitmaps. We compute the connected components of box and wire pixels with Scikit-Image [Wal+14], using their default ordering by lexicographic order of top-left pixel. We then define a list of critical heights: the top of the picture, the height of the centroid of each box component, then the bottom of the picture. For each wire component, we define its source and target as the closest critical height to its top-most and bottom-most pixel. We define the domain of the diagram as the list of wires with the domain as source. We then scan through the picture top to bottom, keeping a list `scan` of the open wires at each height. For each box, we find its input wires in this list and define the offset as the number of wires left of the box node that are not inputs, then we update `scan` with the output wires. We get an encoding `dom, boxes_and_offsets` which yields a valid diagram by construction.

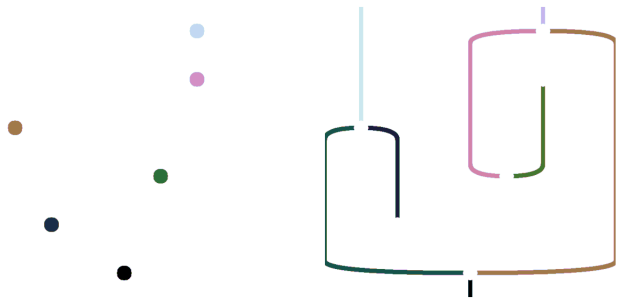
Example 1.3.20. *Suppose we take the following picture of a diagram as input, where the red pixels are boxes and the black pixels are wires:*



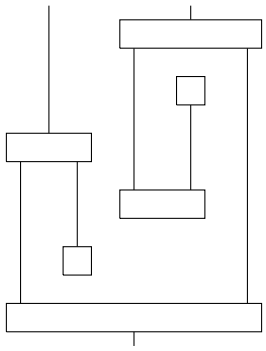
We count $\{1, \dots, 6\}$ *boxes and 8 wires*

$$\{0 \to 3,\ \ 0 \to 1,\ \ 1 \to 4,\ \ 1 \to 6,\ \ 2 \to 4,\ \ 3 \to 6,\ \ 3 \to 5,\ \ 6 \to 7\}$$

for 0 *and* 7 *the domain and codomain of the whole diagram respectively.*

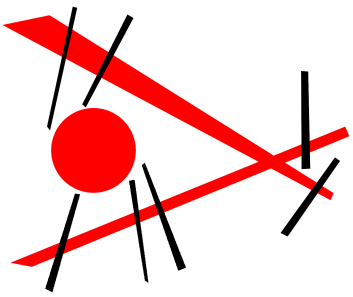


From this, we reconstruct the following diagram by scanning top to bottom:



which is indeed equal to the input picture, up to generic progressive deformation.

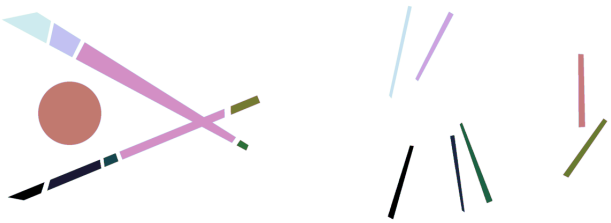
Example 1.3.21. *Suppose we start from the following pastiche of Kandinsky's* Punkt und linie zu fläche *(point and line on the plane):*



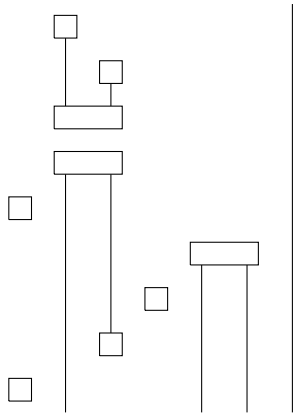
We count $\{1, \ldots, 9\}$ *boxes and 7 wires*

$$\{0 \to 3,\ \ 1 \to 3,\ \ 2 \to 4,\ \ 4 \to 8,\ \ 4 \to 9,\ \ 6 \to 10,\ \ 6 \to 10\}$$

for 0 *and* 10 *the domain and codomain of the whole diagram respectively.*



From this, we reconstruct the following diagram by scanning top to bottom:



which looks nothing like the input because Kandinsky's abstract paintings are not generic progressive plane graphs.

What could be the applications of such a reading algorithm? Of course, real-world pictures do not come in clean red and black bitmaps. We need some more computer vision to label each pixel as belonging either to a box, a wire or the background. This can be achieved by training a convolutional neural network on a large number of example pictures, each annotated with their red and black bitmaps. Once we have trained the machine to classify whether pixels belong to a box or wire, we can also train it to output the label of each pixel, i.e. the label of the box or wire it belongs to. With enough training data, we could automatically turn pictures of Bob's blackboard into circuits that we can execute on a quantum computer. Less trivially, this could be applied to *document layout analysis*: from the picture of say, a hand-drawn calendar from the middle ages, we could train our recognition algorithm to output a diagram that encodes the structure of the calendar. In previous work [Bor+19], we trained convolutional neural networks to extract lines of text in medieval manuscripts, but we had to exclude calendars from our analysis due to the complexity of their layouts. Our diagram recognition machines would enhance the automated analysis of such hand-drawn documents with structured layouts.

Our simple algorithm is robust to any deformation of lgpp graph, but there is much room for improvement. The easiest assumption to remove is genericity, i.e. boxes need not be at distinct heights. Non-generic progressive plane graphs, i.e. with potentially horizontal wires between boxes at the same height, have been characterised as the arrows of free *double categories*. Delpeuch [Del20b] shows how they can be encoded as lgpp graphs with an extra label on each wire for whether it is horizontal or vertical. We can also remove the progressivity assumption, i.e. wires can bend backwards. We have two options: either a) we write a geometric

algorithm than can find the endpoints of any bended wire, or b) we train another neural network to detect each point of non-progressivity, i.e. the cups and caps where the vertical derivative of a wire changes sign. Such non-progressive plane graphs can be encoded as lgpp graphs with *cups and caps boxes*, they are the arrows of free *pivotal categories* which we discuss in section 1.4.1.

Next, we can get rid of the planarity assumption: the projection of the topological graph onto the plane need not be an embedding, i.e. wires can cross. We can improve our reading algorithm in a similar way: either a) some geometric algorithm or b) some black-box neural network that can detect the points of non-planarity, i.e. the intersection of wires. Such non-planar progressive graphs can be encoded as lgpp graphs with *swap boxes*, they are the arrows of free *symmetric monoidal categories* which we discuss in section 1.4.2. Non-planar non-progressive graphs, i.e. where wires can bend and swap, are the arrows of free *compact closed categories*, which play the starring role in categorical quantum mechanics.

Finally, we can even remove the graph assumption: wires need not be homeomorphic to an open interval. We merely require that they are one-dimensional open subsets of the plane, i.e. wires can split and merge. We do not even need to assume that their boundary is in the nodes of the graph, i.e. wires can start or end anywhere in the plane. Again, we can improve our reading algorithm by encoding such hypergraphs (i.e. where wires can have any number of sources and targets) as lgpp graphs with *spider boxes* for the splits, merges, starts and ends of each wire. Labeled hypergraphs are the arrows of free *hypergraph categories*, which we discuss in section 1.4.3. In section 1.5, we discuss the relationship between this definition of hypergraph diagrams as (equivalence classes of) planar diagrams with swap and spider boxes and the more traditional graph-based definition.

1.4 Adding extra structure

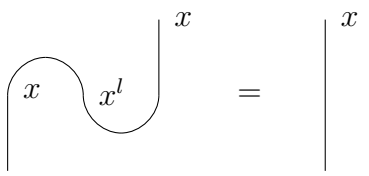
This section looks at the implementation of monoidal categories with extra structure: rigid, symmetric, hypergraph and cartesian closed categories.

1.4.1 Rigid categories & wire bending

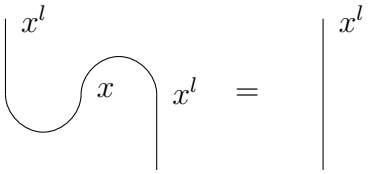
In sections 1.1 and 1.2 we discussed the fundamental notion of *adjunction* with the example of free-forgetful functors. The definition of left and right adjoints in terms of unit and counit natural transformations makes sense in **Cat**, but it can be translated in the context of any monoidal category C. An object $x^l \in C_0$ is

the left adjoint of $x \in C_0$ whenever there are two arrows $\texttt{cup}(x) : x^l \otimes x \to 1$ and $\texttt{cap}(x) : 1 \to x \otimes x^l$ (also called counit and unit) such that:

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



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

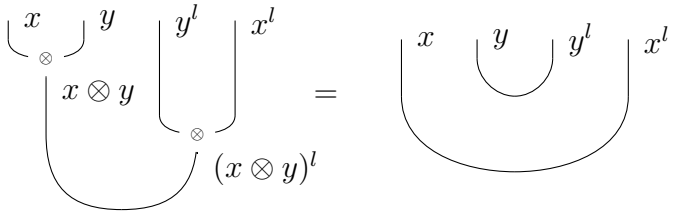


This is equivalent to the condition that the functor $x^l \otimes - : C \to C$ is the left adjoint of $x \otimes - : C \to C$. Symmetrically, $x^r \in C_0$ is the right-adjoint of $x \in C_0$ if x is its left adjoint. We say that C is *rigid* (also called *autonomous*) if every object has a left and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

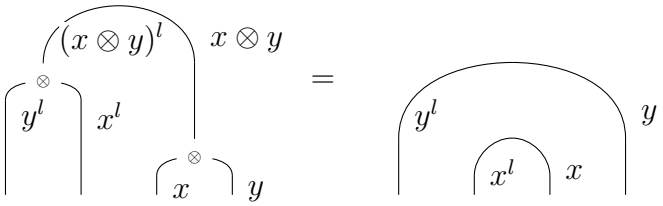
- adjoints are unique up to isomorphism,
- adjoints are monoid anti-homomorphisms, i.e. $(x \otimes y)^l \simeq y^l \otimes x^l$ and $1^l \simeq 1$,
- left and right adjoints cancel, i.e. $(x^l)^r \simeq x \simeq (x^r)^l$,

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

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



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

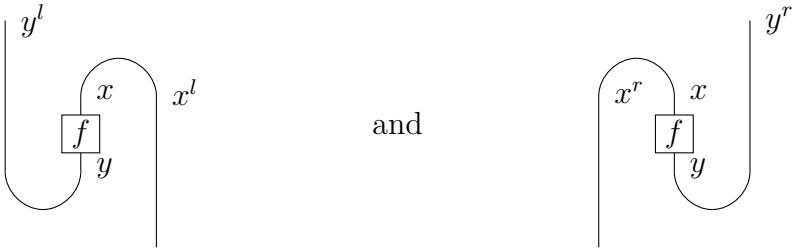


- $\texttt{cup}(1) = \texttt{cap}(1) = \texttt{id}(1)$, drawn as the equality of three empty diagrams.

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

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

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



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

Example 1.4.1. *Recall from example 1.2.5 that for any category C, the category C^C of endofunctors and natural transformations is monoidal. Its subcategory with endofunctors that have both left and right adjoints is rigid. Its subcategory with endofunctors that have equal left and right adjoints is pivotal.*

Example 1.4.2. Tensor$_{\mathbb{S}}$ *is †-pivotal with left and right adjoints given by list reversal, cups and caps by the Kronecker delta* $\texttt{cup}(n)(i,j) = \texttt{cap}(n)(i,j) = 1$ *if* $i = j$ *else* 0*. Note that for tensors of order greater than 2, the* diagrammatic transpose *defined in this way differs from the usual* algebraic transpose*: the former reverses list order while the latter is the identity on objects.*

Example 1.4.3. Circ *is †-pivotal with the preparation of the Bell state as cap and the post-selected Bell measurement as cup (both are scaled by* $\sqrt{2}$*). The snake equations yield a proof of correctness for the (post-selected) quantum teleportation protocol.*

Example 1.4.4. *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 2.1. A commutative pregroup is a (preordered) group: left and right adjoints coincide with the multiplicative inverse.*

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

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

There is a forgetful functor $U :$ **RigidCat** $\to$ **RigidSig** which sends any strictly-rigid foo-monoidal category to its underlying rigid signature. We now describe its left-adjoint $F^r :$ **RigidSig** $\to$ **RigidCat**. Given a rigid signature Σ, we define a

[1] In his original definition [Lam99b], 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 $x = x \otimes x^l \otimes x$.

monoidal signature $\Sigma^r = \Sigma \cup \{\texttt{cup}(x)\}_{x \in \Sigma_0} \cup \{\texttt{cap}(x)\}_{x \in \Sigma_0}$. The free rigid category is the quotient $F^r(\Sigma) = F(\Sigma^r)/R$ of the free monoidal category by the snake equations R. That is, the objects are lists of simple types $(\Sigma_0 \times \mathbb{Z})^\star$, the arrows are equivalence classes of diagrams with cup and cap boxes. This is implemented in the `rigid` module of DisCoPy as outlined below.

Listing 1.4.5. Implementation of objects and types of free rigid categories.

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

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

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

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

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

Example 1.4.6. *We can check the axioms for objects in rigid categories hold on the nose.*

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

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

Listing 1.4.7. Implementation of the arrows of free rigid categories.

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

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

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

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

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

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

The *snake removal* algorithm listed below computes the normal form of diagrams in rigid categories. It is a concrete implementation of the abstract algorithm

described in pictures by Dunn and Vicary [DV19, p. 2.12]. First, we implement a subroutine `follow_wire` takes a codomain node (given by the index `i` of its box and the index `j` of itself in the box's codomain) follows the wire till it finds either the domain of another box or the codomain of the diagram. When we follow a wire, we compute two lists of *obstructions*, the index of each box on its left and right. The `find_snake` function calls `follow_wire` for each `Cap` in the diagram until it finds one that is connected to a `Cup`, or returns `None` otherwise. A `Yankable` snake is given by the index of its cup and cap, the two lists of obstructions on each side and whether it is a left or right snake. `unsnake` applies `interchange` repeatedly to remove the obstructions, i.e. to make the cup and cap consecutive boxes in the diagram, then returns the diagram with the snake removed. Each snake removed reduces the length n of the diagram by 2, hence the `snake_removal` algorithm makes at most $n/2$ calls to `find_snake`. Finally, we call `monoidal.Diagram.normal_form` which takes at most cubic time. Finding a snake takes quadratic time (for each cap we need to follow the wire at each layer) as well as removing it (for each obstruction we make a linear number of calls to `interchange`). Thus, we can compute normal forms for diagrams in free rigid categories in cubic time. We conjecture that we can in fact solve the word problem (i.e. deciding whether two diagrams are equal) in quadratic time using the same reduction to planar map isomorphism as in theorem 1.2.20.

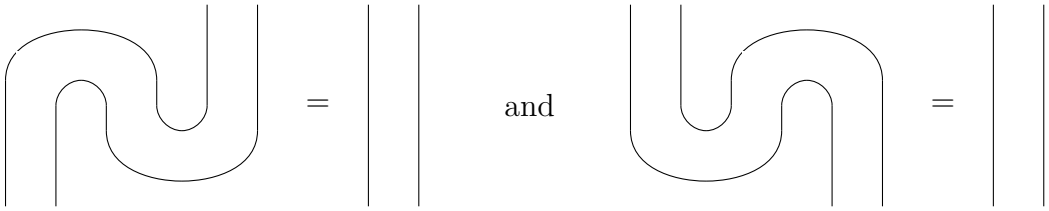
Listing 1.4.8. Outline of the snake removal algorithm.

```
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 1.4.9. *We can check that the snake equations hold up to normal form.*

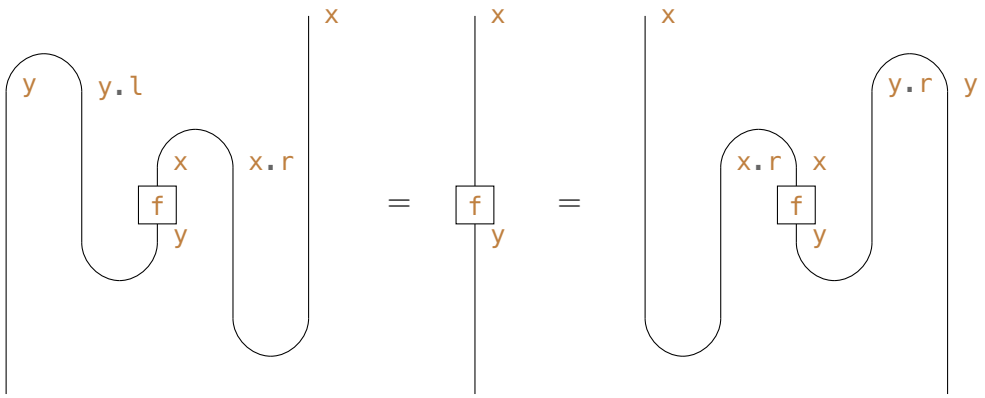
```
t = x @ y

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

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

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



Example 1.4.10. *We can check that left and right transpose cancel up to normal form.*

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

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

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



Listing 1.4.11. Implementation of **Circ** as a pivotal category.

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

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

Example 1.4.12. *We can verify the teleportation protocol for two qubits.*

```
two_qubit_Bell_state = Circuit.caps(qubit ** 2)
two_qubit_Bell_effect = Circuit.cups(qubit ** 2)

assert (two_qubit_Bell_state @ qubit ** 2 >> qubit ** 2 @ two_qubit_Bell_effect).eval()\
    == (qubit ** 2).eval()\
    == (qubit ** 2 @ two_qubit_Bell_state >> two_qubit_Bell_effect @ qubit ** 2).eval()
```

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

Listing 1.4.13. Implementation of strict rigid functors.

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

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

Listing 1.4.14. Implementation of $\mathbf{Tensor}_{\mathbb{S}}$ as a pivotal category.

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

Example 1.4.15. *We can check that* $\mathbf{Tensor}_{\mathbb{S}}$ *is indeed pivotal.*

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

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

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

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

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

Listing 1.4.16. Implementation of free †-pivotal categories.

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

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

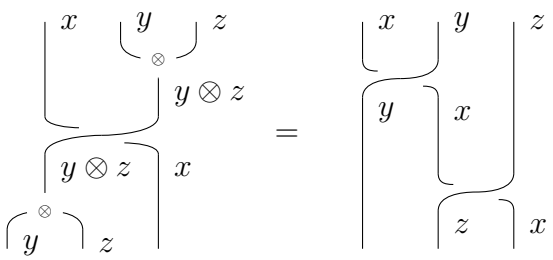
```
class Diagram(rigid.Diagram): pass

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

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

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

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

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

1.4.2 Braided categories & wire crossing

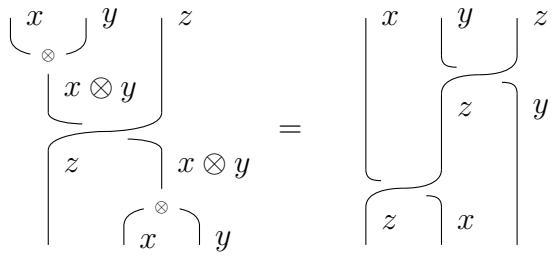
With rigid and pivotal categories, we have removed the assumption that diagrams are progressive: we can bend wires. With braided and symmetric monoidal categories, we now remove the planarity assumption: wires can cross.

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

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



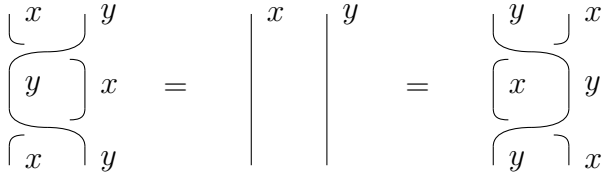
- $B(x \otimes y, z) \quad = \quad x \otimes B(y, z) \ \fatsemi \ B(x, z) \otimes y$



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

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

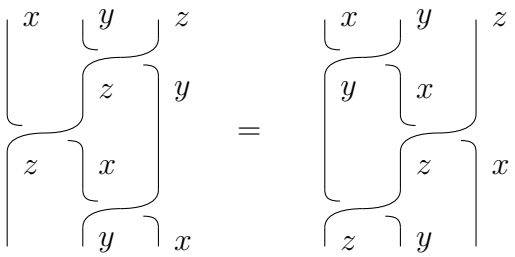
- $B(x, y) \fatsemi B^{-1}(x, y) \;=\; \mathtt{id}(x \otimes y) \;=\; B^{-1}(x, y) \fatsemi B(x, y)$,



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

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

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



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

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

Remark 1.4.17. *A symmetric (braided) category with one generating object is called a PROP (PROB) for PROduct and Permutation (Braid). Indeed, the arrows of the free PROP with no generating boxes (i.e. only swaps) are* permutations*, the arrows of the free braided PRO with no boxes are called* braids*. Both are* groupoids*, i.e. all their arrows are isomorphisms, which also implies that they are †-braided with the dagger given by the inverse. For every $n \in \mathbb{N}$, the arrows $f : x^n \to x^n$ in the free PROP (PROB) are the elements of the n-th symmetric group S_n (braid group B_n).*

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

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

Listing 1.4.18. Implementation of free †-braided and †-symmetric categories.

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

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

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

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

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

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

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

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

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

Diagram.naturality = naturality

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

Example 1.4.19. *We can check the hexagon equations hold on the nose.*

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

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

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

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

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

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

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

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

Listing 1.4.20. Implementation of **Pyth** and $\mathbf{Tensor}_{\mathbb{S}}$ as symmetric categories.

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

Function.swap = Function.braid = function_swap

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

Tensor.swap = Tensor.braid = tensor_swap
```

Example 1.4.21. *We can check the axioms for symmetric categories hold in* $\mathbf{Tensor}_{\mathbb{S}}$ *and* **Pyth**.

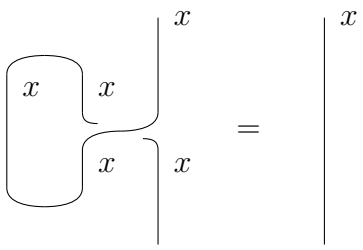
```
swap_twice = Diagram.swap(x, y @ z) >> Diagram.swap(y @ z, x)

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

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

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

assert G(f @ x >> Swap(b, x))(1j, 2) == G(Swap(a, x) >> x @ f)(1j, 2) == (2, f(1j))
```

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

Remark 1.4.22. *Note that the naturality axioms in* **Pyth** *hold only for its subcategory of pure functions, as we will see in section 1.5* **Pyth** *is in fact a symmetric* premonoidal *category. This is also the case for* $\mathbf{Tensor}_{\mathbb{S}}$ *when the rig* $\mathbb{S}$ *is non-commutative.*

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

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



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

The *unknotting problem* (given a knot, can it be untied, i.e. continuously deformed to a circle?) is a candidate NP-intermediate problem: it is decidable [Hak61] and in NP [Lac15], but there is neither a proof of it being NP-complete nor a polynomial-time algorithm. Delpeuch and Vicary [DV21] proved that the word

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

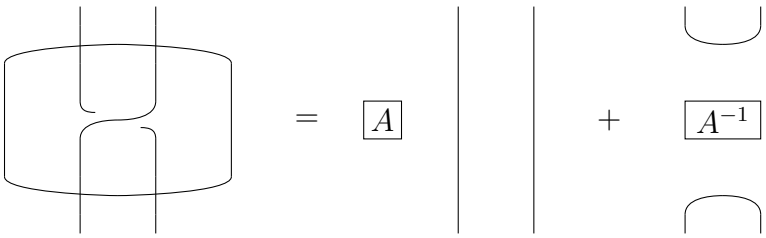
problem for free braided categories is unknotting-hard. Hence, there is little hope of finding a simple polynomial-time algorithm for computing normal forms of braided diagrams. It is not known whether it is even decidable.

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

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

Example 1.4.23. *We can define knot polynomials such as the Kauffman bracket using pivotal functors into a category where braids are defined as a weighted sum of diagrams.*

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

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

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

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



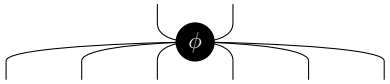
1.4.3 Hypergraph categories & wire splitting

With compact closed and tortile categories, we have removed both the progressivity and the planarity assumptions: wires can bend and cross. With *hypergraph categories* we remove the assumption that diagrams are graphs: wires can split and merge, they need not be homeomorphic to an open interval. A hypergraph category is a symmetric category with *coherent special commutative spiders*, let's spell out what this means.

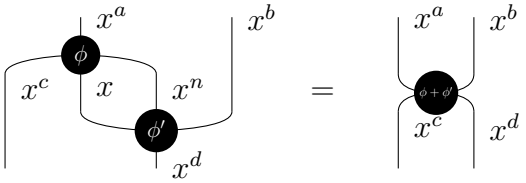
An object x in a monoidal category C has *spiders* with *phases* in a monoid $(\Phi, +, 0)$ if it comes equipped with a family of arrows $\texttt{spider}_{\phi,a,b}(x) : x^a \to x^b$ for every phase $\phi \in \Phi$ and pair of natural numbers $a, b \in \mathbb{N}$, such that the following *spider fusion*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

$$\texttt{spider}_{\phi,a,c+n+1}(x) \otimes x^b \ \fatsemi \ x^c \otimes \texttt{spider}_{\phi',c+n+1,d}(x) \ = \ \texttt{spider}_{\phi+\phi',a+b,c+d}(x)$$

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



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

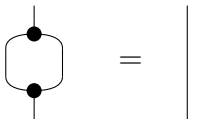


From spider fusion, we can deduce the following properties:

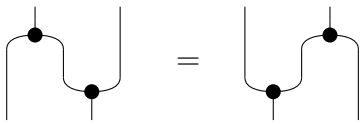
- $\texttt{merge}(x) = \texttt{spider}_{0,2,1}(x)$ and $\texttt{unit}(x) = \texttt{spider}_{0,0,1}(x)$ form a monoid,

- $\texttt{split}(x) = \texttt{spider}_{0,1,2}(x)$ and $\texttt{counit}(x) = \texttt{spider}_{0,1,0}(x)$ form a comonoid,

- $\texttt{split}(x) \fatsemi \texttt{merge}(x) = \texttt{id}(x)$, called the *special* condition,

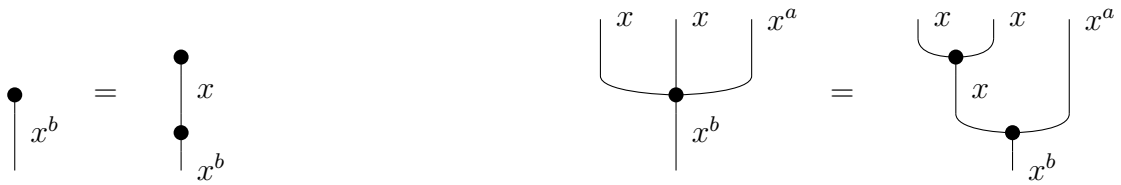


- $\texttt{merge}(x) \otimes x \fatsemi x \otimes \texttt{split}(x) = x \otimes \texttt{merge}(x) \fatsemi \texttt{split}(x) \otimes x$, called the *Frobenius law*.



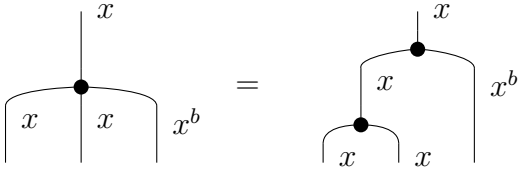
In fact, when the phases are trivial $\Phi = \{0\}$ these four axioms are sufficient to deduce spider fusion, spiders are also called *special Frobenius algebras*. Indeed, given a monoid $\texttt{merge}(x) : x \otimes x \to x$, $\texttt{unit}(x) : 1 \to x$ and a comonoid $\texttt{split}(x) : x \to x \otimes x$, $\texttt{counit}(x) : x \to 1$ subject to the Frobenius law, we can construct $\texttt{spider}_{a,b}(x) : x^a \to x^b$ by induction on the number of legs. The base case is given by the special condition $\texttt{spider}_{1,1}(x) = \texttt{id}(x)$. Then we define spiders with $a \in \mathbb{N}$ input legs for $a \neq 1$:

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



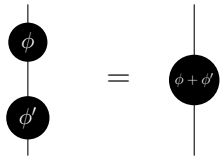
Finally we define spiders with one input leg by induction on the output legs $b \in \mathbb{N}$:

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

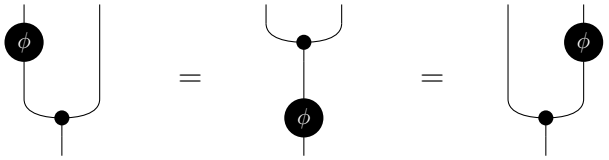


One can show that this satisfies the spider fusion law, again by induction on the legs [HV19, Lemma 5.20]. In this way, we can construct an infinite family of spiders from just the four boxes $\texttt{merge}(x), \texttt{unit}(x), \texttt{split}(x), \texttt{counit}(x)$ and a finite set of equations: a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

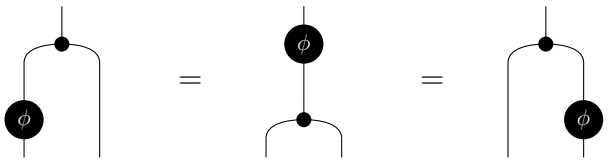
- $\texttt{shift}_\_(x)$ is a monoid homomorphism $\Phi \to C(x,x)$, i.e. $\texttt{shift}_0(x) = \texttt{id}(x)$ and $\texttt{shift}_\phi(x) \fatsemi \texttt{shift}_{\phi'} = \texttt{shift}_{\phi+\phi'}(x)$,



- phase shifts commute with the product, $\texttt{shift}_\phi(x) \otimes x \fatsemi \texttt{merge}(x) = \texttt{merge}(x) \fatsemi \texttt{shift}_\phi(x) = x \otimes \texttt{shift}_\phi(x) \fatsemi \texttt{merge}(x)$,

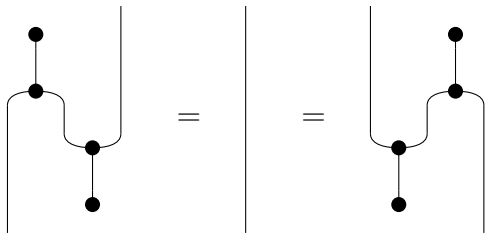


- phase shifts commute with the coproduct, $\texttt{split}(x) \fatsemi \texttt{shift}_\phi(x) \otimes x = \texttt{shift}_\phi(x) \fatsemi \texttt{split}(x) = x \otimes \texttt{split}(x) \fatsemi \texttt{shift}_\phi(x)$.



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

Remark 1.4.24. *Given any Frobenius algebra on an object x, we can show that x is its own left and right adjoint. Indeed, take $\texttt{cup}(x) = \texttt{unit}(x) \fatsemi \texttt{split}(x)$ and $\texttt{cap}(x) = \texttt{merge}(x) \fatsemi \texttt{counit}(x)$, then the Frobenius law and the (co)unit law of the (co)monoid implies the snake equations. Thus, a category with (not-necessarily special) spiders on every object is automatically a pivotal category.*



Example 1.4.25. *In any pivotal category, there is a Frobenius algebra for every object of the form $x^\star \otimes x$ given by:*

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

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

Example 1.4.26. *The category* $\mathbf{Tensor}_{\mathbb{S}}$ *has spiders for every dimension $n \in \mathbb{N}$ with phases in any submonoid of $\phi \in (\mathbb{S}, \times, 1)^n$. They are given by* $\texttt{spider}_{\phi,a,b}(n) = \sum_{i \leq n} \phi_i |i\rangle^{\otimes a} \langle i|^{\otimes b}$ *where $|i\rangle$ ($\langle i|$) is the i-th basis row (column) vector.*

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

When $\mathbb{S}$ is a field, we can divide every ϕ_i by ϕ_0, or equivalently require that $\phi_0 = 1$. Indeed, we can represent any spider with $\phi_0 \neq 1$ as a spider with $\phi_0 = 1$

multiplied by the scalar ϕ_0, which is called a global phase. *When $\mathbb{S} = \mathbb{C}$ and $n = 2$, we usually take the monoid of phases to be the unit circle and write it in terms of addition of angles.*

Example 1.4.27. *In the category* **Circ** *of quantum circuits, if we allow post-selected measurements then we can construct spiders with the unit circle as phases. The spiders with no inputs legs are called the (generalised) GHZ states:*

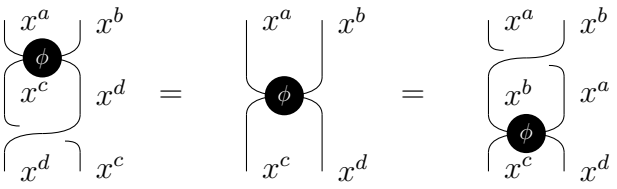
$$\texttt{spider}_{\alpha,0,b} = |0\rangle^{\otimes b} + e^{i\alpha}|1\rangle^{\otimes b}$$

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

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

If the monoidal category C is braided, we can remove the planarity assumption and define *commutative spiders* as those where the monoid and comonoid are commutative, i.e.

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

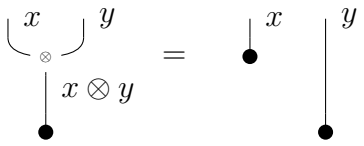


Together with spider fusion, this implies that the monoid of phases is also commutative. The PROB of commutative Frobenius algebras (without the special

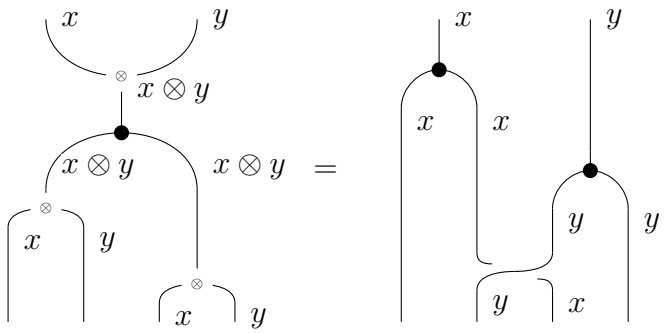
condition), i.e. diagrams with only spiders and braids, defines a notion of "well-behaved" 1d subspaces of 3d space, up to continuous deformation. When the category is furthermore symmetric, the PROP of commutative spiders defines a notion of "well-behaved" 1d spaces up to diffeomorphism, or equivalently 1d subspaces of 4d space, i.e. one where wires can pass through each other and all knots untie. It is equivalent to the category of two-dimensional *cobordisms* [Abr96], i.e. oriented 2d manifolds with a disjoint union of circles as boundary. Intuitively, a 2d cobordism can be thought of as a (non-planar) wire with a width, i.e. one that we can draw.

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

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



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



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

Thus we get to our definition: a *hypergraph category* is a symmetric category with coherent special commutative spiders on each object. We can take the data to be that of a foo-monoidal category C together with a function $\texttt{spider} : \mathbb{N} \times \mathbb{N} \times C_0 \to C_1$ or equivalenty, with four functions $\texttt{merge}, \texttt{unit}, \texttt{split}, \texttt{counit} : C_0 \to C_1$. Once we fix the spiders for generating objects, we get spiders for any type (i.e. list of objects). A hypergraph functor is a symmetric functor $F : C \to D$ between hypergraph categories such that $F \fatsemi \texttt{spider}_{a,b} = \texttt{spider}_{a,b} \fatsemi F$. Thus

we get a category **HypCat** with a forgetful functor U : **HypCat** $\to$ **MonSig**. Its left adjoint F^H : **MonSig** $\to$ **HypCat** is defined as a quotient $F^S(\Sigma^H)/R$ of the free symmetric category generated by $\Sigma^H = \Sigma\texttt{spider}$ and the relation R given by the equations for commutative spiders. Equivalently, we can take $\Sigma^H = \bigcup\{\Sigma, \texttt{merge}, \texttt{unit}, \texttt{split}, \texttt{counit}\}$ and R given by the equations for special commutative Frobenius algebras. A †-*hypergraph category* is a †-symmetric category (i.e. the swaps are unitaries) where the dagger is a hypergraph functor. We also require that the monoid of phases is in fact a group with the dagger as inverse or equivalently, that phase shifts are unitaries.

Example 1.4.28. *Every commutative rig* $\mathbb{S}$ *induces a* †*-hypergraph category* $\mathbf{Tensor}_{\mathbb{S}}$ *with the transpose as dagger. Arguably, special commutative Frobenius algebras were first defined by Peirce [Pei06] with their interpretation in the category of relations, or equivalently* $\mathbf{Tensor}_{\mathbb{B}}$*. Indeed, they correspond to what Peirce calls* lines of identity*: they express in two dimensions what one-dimensional first-order logic would express with equality symbols. For example, take a binary predicate encoded as a box* $p : 1 \to x^2$ *(interpreted as the formula* $\exists\ a \cdot \exists\ b \cdot p(a, b)$*) then the diagram* $p \fatsemi \texttt{merge}(x)$ *is interpreted as the formula* $\exists\ a \cdot \exists\ b \cdot p(a, b) \wedge a = b$ *or equivalently* $\exists\ a \cdot p(a, a)$*. Thus, every first-order logic formula can be written as a diagram with boxes for predicates, spiders for identity and bubbles for negation. The equivalence of formulae can be defined as a quotient of a free hypergraph category with bubbles, i.e. all the rules of first-order logic can be given in terms of diagrams.*

Example 1.4.29. *The category of complex tensors* $\mathbf{Tensor}_{\mathbb{C}}$ *is* †*-hypergraph with the spiders given in example 1.4.26. Any unitary matrix* $U : n \to n$ *defines another family of spiders* $U^{\otimes a} \fatsemi \texttt{spider}_{\phi,a,b}(n) \fatsemi (U^\dagger)^{\otimes b}$*. In fact, every unitary arises in this way, see Heunen and Vicary [HV19, Corollary 5.32]. Thus, the axioms for spiders allow us to define any orthonormal basis without ever mentioning basis vectors: they are merely the states* $v : 1 \to n$ *for which the comonoid is natural, i.e.* $v \fatsemi \texttt{split}(x) = v \otimes v$ *and* $v \fatsemi \texttt{counit}(x) = \texttt{id}(1)$*.*

Example 1.4.30. *With the spiders defined in example 1.4.27,* **Circ** *is a* †*-hypergraph category and the evaluation functor* $\mathbf{Circ} \to \mathbf{Tensor}_{\mathbb{C}}$ *is a* †*-hypergraph functor.*

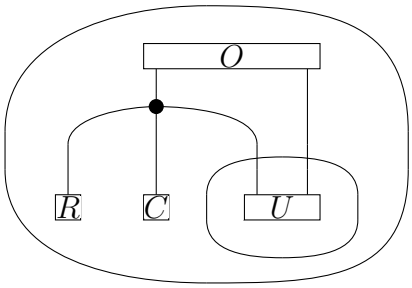
DisCoPy implements spiders for types of length one (i.e. generating objects) as a subclass of `Box` and spiders for arbitrary types as a method `Diagram.spiders`. That is, we make every free category and every functor †-hypergraph by default.

Listing 1.4.31. Implementation of †-hypergraph categories and functors.

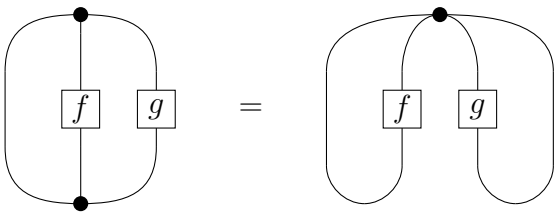
```python
class Spider(Box):
    def __init__(self, a: int, b: int, x: Ty, phase=None):
        assert len(x) == 1
        self.object, self.phase = x, phase or 0
        name = "Spider({})".format(', '.join(map(str, (a, b, x, phase))))
        super().__init__(name, dom=x ** a, cod=x ** b)

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

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

Diagram.spiders = coherence(Spider)
Diagram.cups = nesting(lambda x, _: Spider(0, 2, x))
Diagram.caps = nesting(lambda x, _: Spider(2, 0, x))

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

Example 1.4.32. *We can now extend example* **??** *to arbitrary formulae of first-order logic. Every variable that appears exactly twice is encoded as a wire (possibly with cups and caps), every variable that appears $n \neq 2$ is encoded as an n-legged spider. For example, the formula $\forall c\ \forall o\ O(c, o) \wedge R(c) \wedge C(c) \implies U(c, o)$ (interpreted as "every object of a rigid cartesian category is also its unit") can be encoded as a diagram with a wire for o and a four-legged spider for c.*

```
class Formula(Diagram):
    cut = lambda self: self.bubble(name="_not")

class Predicate(Box, Formula):
    def __init__(self, name, dom): Box.__init__(self, name, Ty(), dom)

def model(size: dict[Ty, int], data: dict[Predicate, list[bool]]):
    return Functor(ob=size, ar={p: [data[p]] for p in data},
                   dom=Category(Ty, Formula), cod=Category(list[int], Tensor[bool]))

objects, categories = Ty('o'), Ty('c')
has_object, has_unit = [Predicate(name, categories @ objects) for name in "OU"]
is_rigid, is_cartesian = [Predicate(name, categories) for name in "RC"]

rigid_cartesian_implies_trivial = (
    has_object >> Formula.spiders(1, 3, categories) @ objects
    >> (is_rigid @ is_cartesian @ has_unit.cut()).dagger()).cut()

size = {objects: 2, categories: 2}
predicate_values = itertools.product(*size[categories] * [[0, 1]])
relation_values = itertools.product(*size[categories] * size[o  bjects] * [[0, 1]])

for O, U, R, C in itertools.product(2 * [predicate_values] + 2 * [relation_values]):
    F = model(size, {has_object: O, has_unit: U, is_rigid: R, is_cartesian: C})
    is_rigid_cartesian_and_has_object = lambda i, j:\
        F(has_object)[i, j] and F(is_rigid)[i] and F(is_cartesian)[i]
    assert F(rigid_cartesian_implies_trivial) == all(
        not is_rigid_cartesian_and_has_object(i, j) or F(has_unit)[i, j]
        for i in range(size[categories]) for j in range(size[objects]))

rigid_cartesian_implies_trivial.draw()
```



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



1.4.4 Products & coproducts

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

Now if we go back to the intuition of diagrams as pipelines and their wires as carrying data, not all might be lost about spiders. Indeed, it makes sense to split a

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

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

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

for all pure functions `f` and inputs `xs` with `n = len(xs)`. Once drawn as a diagram, the naturality equations for comonoids allows us to either copy or delete boxes by passing them through either the coproduct or the counit.

A *cartesian category* is a symmetric category with coherent, natural commutative comonoids. The category **Pyth** is an example of cartesian category, as well as the categories **Set**, **Mon**, **Cat**, **MonCat**, etc. The category $\mathbf{Mat}_{\mathbb{S}}$ is also a cartesian category with the direct sum as tensor. Our definition of cartesian is convenient if we want to draw string diagrams and interpret them as functions but it is rather cumbersome: checking that a given category fits the definition involves a lot of structure (tensor, swaps and comonoids) and many axioms relating them. In practice, we usually take an equivalent definition: a category C is cartesian if

it has *categorical products* and a *terminal object*. An object $1 \in C_0$ is terminal if there is a unique arrow $\texttt{counit}(x) : x \to 1$ from each object C_0. An object $x_0 \times x_1 \in C_0$ is the product of two objects $x_0, x_1 \in C_0$ if it comes equipped with a pair of arrows $\pi_0 : x_0 \times x_1 \to x_0$ and $\pi_1 : x_0 \times x_1 \to x_1$ such that for all pairs of arrows $f_0 : y \to x_0$ and $f_1 : y \to x_1$ there is a unique $f = \langle f_0, f_1 \rangle : y \to x_0 \times x_1$ such that $f \fatsemi \pi_0 = f_0$ and $f \fatsemi \pi_1 = f_1$. These definitions are usually drawn as commutative diagrams where the full lines are universally quantified and the dotted line is uniquely existentially quantified.

From these two *universal properties* we can deduce that terminal objects and categorical products are unique up to a unique isomorphism and that they form a monoid on objects $(C_0, \times, 1)$, up to natural isomorphism. Given two arrows $f : a \to b$ and $g : c \to d$ we have two arrows $\pi_0 \fatsemi f : a \times c \to b$ and $\pi_1 \fatsemi g : a \times c \to d$, thus there is a unique $f \times g = \langle \pi_0 \fatsemi f, \pi_1 \fatsemi g \rangle : a \times b \to c \times d$. One can show that this makes the category C a monoidal category, i.e. $(C_1, \times, \texttt{id}(1))$ is a monoid and the interchange law holds. Furthermore, we can show C is symmetric with the swaps given by $S(x, y) = \langle \pi_1, \pi_0 \rangle : x \times y \to y \times x$. Finally, we can show C has coherent natural commutative comonoids given by $\texttt{split}(x) = \langle \texttt{id}(x), \texttt{id}(x) \rangle : x \to x \times x$ and $\texttt{counit}(x) : x \to 1$.

In the other direction, if C has coherent natural commutative comonoids we can deduce that 1 a terminal object from the naturality of the counit. For any arrows $f_0 : y \to x_0$ and $f_1 : y \to x_1$ we can define

- $\langle f_0, f_1 \rangle = \texttt{split}(y) \fatsemi f_0 \otimes f_1$,
- $\pi_0 = \texttt{id}(x_0) \otimes \texttt{counit}(x_1)$ and $\pi_1 = \texttt{counit}(x_0) \otimes \texttt{id}(x_1)$,

and show that $\otimes = \times$ is in fact a categorical product, see Selinger's survey [Sel10, Section 6.1]. A functor is cartesian when it preserves the categorical product, or equivalently if it is a symmetric functor that preserves the comonoid. This defines a category **CCat** of cartesian categories and functors. From lemma 1.2.14 we can assume that cartesian categories are free-on-objects, i.e. the monoid axioms for objects are equalities rather than natural transformations. Thus, we get a forgetful functor $U : \textbf{CCat} \to \textbf{MonSig}$ with its left-adjoint given by a quotient of the free symmetric category $F^C(\Sigma) = F^S(\Sigma \cup \texttt{split} \cup \texttt{counit})/R$ with the relations R given by the naturality equations for each box.

Taking the opposite definition, a *cocartesian category* is one with a categorical coproduct, or equivalently with a coherent natural commutative monoid. For example, the category **Set** is cocartesian with the disjoint union, the category **Pyth**

is cocartesian with the tagged union: the merging function takes a tagged element of an n-fold disjoint union and forgets the tag. While cartesian structures can be thought of in terms of product types and data copying, cocartesian structures formalise tagged unions and conditional branching. Indeed, when we interpret cocartesian diagrams in **Pyth** parallel wires encode the different branches of a program, merging two wires of the same type means forgetting the difference between two branches.

DisCoPy implements free (co)cartesian categories with a subclass of `Box` for making (merging) `n` copies of a type `x` of length one. The class method `Diagram.copy` allows to make `n` copies of an arbitrary type `x` by calling the `coherence` subroutine of the previous section, it takes an optional argument `is_dagger` that allows to merge copies instead. Cartesian functors take `Copy` boxes of its domain to the `copy` method of its codomain.

Listing 1.4.33. Implementation of free (co)cartesian categories and functors.

```
class Copyable:
    @classmethod
    def discard(cls, x: Ty, is_dagger=False) -> Copyable:
        return cls.copy(x, 0, is_dagger)

    @classmethod
    def merge(cls, x: Ty, n=2, is_dagger=False) -> Copyable:
        return cls.copy(x, n, not is_dagger)

    @classmethod
    def unit(cls, x: Ty): return cls.discard(x, is_dagger=True)

class Diagram(Copyable, symmetric.Diagram):
    @classmethod
    def copy(cls, x: Ty, n=2, is_dagger=False) -> Diagram:
        def factory(a, b, y):
            assert b == 1 if is_dagger else a == 1
            return Copy(a, y)[::-1] if is_dagger else Copy(b, x)
        a, b = (n, 1) if is_dagger else (1, n)
        return coherence(factory)(cls, a, b, x)

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

class Copy(Box):
    def __init__(self, x: Ty, n: int = 2, is_dagger=False):
        assert len(x) == 1
```

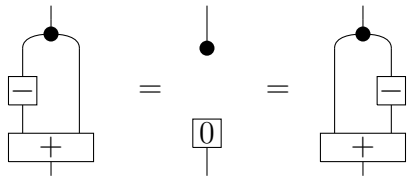
```
        name = "Copy({}, {}){}".format(x, n, "[::-1]" if is_dagger else "")
        dom, cod = (x ** n, x) if is_dagger else (x, x ** n)
        super().__init__(name, dom, cod, is_dagger)

    def dagger(self):
        x = self.cod if self.is_dagger else self.dom
        n = len(self.dom) if self.is_dagger else len(self.cod)
        return Copy(x, n, not self.is_dagger)

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

    def __call__(self, other):
        if isinstance(other, Copy):
            factory = getattr(self.cod.ar, "merge" if other.is_dagger else "copy")
            return factory(self(other.dom), len(other.cod))
        return super().__call__(other)
```

Example 1.4.34. *In a cartesian category, every monoid is automatically a* bialgebra*, i.e. the monoid is a comonoid homomorphism or equivalently, the comonoid is a homomorphism for the monoid. When furthermore the monoid has an inverse, then it is automatically a* Hopf algebra*, the generalisation of groups to arbitrary monoidal categories.*

```
x = Ty('x')
add, minus, zero = Box('+', x @ x, x), Box('-', x, x), Box('0', Ty(), x)
copy, discard = Diagram.copy(x), Diagram.discard(x)

drawing.equation(add >> copy, copy @ copy >> x @ Braid(x, x) @ x >> add @ add)
drawing.equation(zero >> copy, zero @ zero)
```

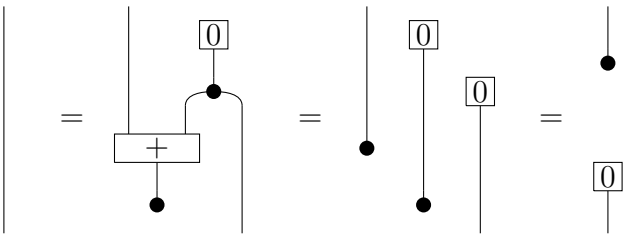

```
drawing.equation(add >> discard, discard @ discard)
drawing.equation(zero >> discard, Diagram.id(Ty()))
```

```
drawing.equation(copy >> minus @ x >> add, discard >> zero, copy >> x @ minus >> add)
```



A bialgebra which is also a Frobenius algebra is necessarily trivial, i.e. isomorphic to the unit.

```
drawing.equation(
    Diagram.id(x),
    x @ zero >> x @ copy >> add @ x >> discard @ x,
    x @ zero @ zero >> discard @ discard @ x,
    discard >> zero)
```



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

fixed objects) can also be seen as the functors from this Lawvere theory to **Set**, functors can also be seen as the natural transformations between such functors.

The free Lawvere theory with no boxes (only swaps, coproducts and counits) is equivalent to $\mathbf{FinSet}^{op}$, the opposite category to finite sets and functions, with the disjoint union as tensor. Indeed, a cartesian diagram $f : x^n \to x^m$ can be seen as the graph of a function from the m to n elements. This free Lawvere theory is also called the *theory of equality*, a functor from it to **Set** (i.e. an algebra for the theory) is just a set with its equality relation, a natural transformation between those functors is just a function. Thus, equality of cartesian diagrams with no boxes reduces to equality of functions between finite sets, which can be implemented as equality of finite dictionaries in Python. It is easy to show that the rules for naturality are confluent and terminating when applied from left to right, i.e. we copy (delete) every box by passing it down through all possible coproducts (counits). At each rewrite step there is one fewer box node above a comonoid node, thus we can reduce the word problem for free cartesian categories to that of free symmetric categories and then to graph isomorphism.

Listing 1.4.35. *Implementation of the free cocartesian category* **FinSet** *with* `int` *as objects and* `Dict` *as arrows.*

```
@dataclass
class Dict(Composable, Tensorable):
    inside: dict[int, int]
    dom: int
    cod: int

    __getitem__ = lambda self, key: self.inside[key]

    def __init__(self, inside, dom, cod):
        assert set(inside.keys()) == set(range(dom))
        assert set(inside.values()) <= set(range(cod))
        self.inside, self.dom, self.cod = inside, dom, cod

    @staticmethod
    def id(x: int = 0): return Dict({i: i for i in range(x)}, x, x)

    @inductive
    def then(self, other: Dict) -> Dict:
        return Dict({i: other[self[i]] for i in range(self.dom)}, self.dom, other.cod)

    @inductive
    def tensor(self, other: Dict) -> Dict:
```

```
        inside = {i: self[i] for i in range(self.dom)}
        inside.update({len(self.dom) + i: len(self.cod) + other[i]
                       for i in range(other.dom)})
        return Dict(inside, self.dom + other.dom, self.cod + other.cod)

    @staticmethod
    def merge(x: int, n: int) -> Dict:
        return Dict({i: i % x for i in range(n * x)}, n * x, x)
```

Example 1.4.36. *We can check equality of cartesian diagrams with one generating object and no boxes.*

```
x = Ty('x')
copy, discard, swap = Diagram.copy(x), Diagram.discard(x), Diagram.swap(x, x)
F = lambda f: Functor({x: 1}, {}, cod=Category(int, Dict))(f.dagger())

assert F(copy >> discard @ x) == F(Diagram.id(x)) == F(copy >> x @ discard)
assert F(copy >> copy @ x) == F(Diagram.copy(x, 3)) == F(copy >> x @ copy)
assert F(copy >> swap) == F(copy)
```

A category with distinct cartesian and cocartesian structures is an example of a *rig category*, i.e. it has two monoidal structures $\oplus$ and $\otimes$ that satisfy the equations of a rig up to natural isomorphism. This is the case for the category **Set** as well as for **Pyth**. The Kronecker product is not a cartesian product for $\mathbf{Mat}_{\mathbb{S}}$ (since this role is taken by direct sums) but it does form a rig category with the direct sum. The arrows of free rig categories can be described as (equivalence classes of) three-dimensional *sheet diagrams* where composition, additive and multiplicative tensor are encoded in three orthogonal axes [CDH20]. These 3d diagrams are a complete language for *dataflow programming* [Del20a], they are not implemented in DisCoPy yet.

The implementation of **Pyth** as a cartesian category with `tuple` and a cocartesian category with `tagged_union` is straightforward. What is more challenging is to make both monoidal structures fit in the same class, i.e. implementing a rig category. Indeed, if we take (**Pyth**, `tuple`) as the foo-monoidal category implemented in section 1.2, `dom: list[type]` and `cod: list[type]` are interpreted as a `tuple`. On the other hand, if we implement (**Pyth**, `tagged_union`) as a foo-monoidal category, lists of types will be interpreted as a `TaggedUnion` instead. Hence, we need to add an optional argument `is_additive: bool` which tells us how we should interpret `dom` and `cod`, and a pair of methods for converting additive into multiplicative functions. In order to keep the syntax simple, we keep the

notation `f @ g` for `tuple` and override the vertical line symbol `f | g` to denote the `tagged_union`. We use the same syntax for the direct sum of two tensors.

Listing 1.4.37. Implementation of **Pyth** as a cartesian category with `tuple` and a cocartesian category with `tagged_union`.

```
is_tuple = lambda typ: hasattr(typ, "__origin__") and typ.__origin__ is tuple
is_union = lambda typ: hasattr(typ, "__origin__") and typ.__origin__ is TaggedUnion

class Function(monoidal.Function):
    is_additive = False

    @classmethod
    @property
    def additive(cls):
        class C(cls): pass
        C.is_additive = True; return C

    def __repr__(self): super().__repr__().replace(
        "Function", "Function.additive" if self.is_additive else "Function")

    def make_additive(self) -> Function:
        if self.is_additive: return self
        dom, cod = (
            [tuple[types]] if len(types) != 1
            else list(types[0].__args__) if is_union(types[0])
            else types for types in (self.dom, self.cod))
        return Function.additive(self.inside, dom, cod)

    def make_multiplicative(self) -> Function:
        if not self.is_additive: return self
        dom, cod = (
            [TaggedUnion[types]] if len(types) != 1
            else list(types[0].__args__) if is_tuple(types[0])
            else types for types in (self.dom, self.cod))
        return Function(self.inside, dom, cod)

    def tuple(self, other: Function) -> Function:
        if self.is_additive: return self.make_multiplicative().tuple(other)
        if other.is_additive: return self.tuple(other.make_multiplicative())
        return super().tuple(other)

    def tagged_union(self, other: Function) -> Function:
        if not self.is_additive: return self.make_additive().tagged_union(other)
        if not other.is_additive: return self.tagged_union(other.make_additive())
```

```
        inside = self.inside if len(self.dom) == 1 and len(other.dom) == 0\
            else other.inside if len(self.dom) == 0 and len(other.dom) == 1\
            else lambda i, x: self(i, x) if i < len(self.dom) else other(i - len(self.dom), x)
        dom, cod = self.dom + other.dom, self.cod + other.cod
        return Function(inside, dom, cod)

    @staticmethod
    def copy(x: list[type], n: int, is_dagger=False):
        if is_dagger: return Function.merge(x, n)
        if n == 1: return Function.id(x)
        return Function(lambda *xs: n * xs, dom=x, cod=x ** n)

    @staticmethod
    def merge(x: list[type], n: int, is_dagger=False):
        if is_dagger: return Function.copy(x, n)
        if n == 1: return Function.additive.id(x)
        return Function.additive(lambda _, xs: xs, dom=x ** n, cod=x)

    @inductive
    def tensor(self, other: Function) -> Function:
        return self.tagged_union(other) if self.is_additive else self.tuple(other)

    __or__ = tagged_union
```

Example 1.4.38. *We can implement the architecture of a neural network as a cartesian diagram and its evaluation as a functor to* `Function`*.*

```
add = lambda x, n: Box('$+$', x ** n, x)
ReLU = lambda x: Box('$\sigma$', x, x)
weights = [Box('w{}{}'.format(i, j), x, x) for i in range(2) for j in range(2)]
bias = Box('b', Ty(), x)

network = Diagram.copy(x @ x, 2) >> Diagram.tensor(*weights) @ bias >> add(x, 5) >> ReLU(x)

F = Functor(ob={x: int}, ar={
    add(x, 5): lambda *xs: sum(xs),
    ReLU(x): lambda x: max(0, x),
    bias: lambda: -1, **dict(zip(weights, [0, 1, 2, 3]))},
    cod=Category(list[type], Function))

assert F(network)(42, -43) == max(0, sum([42 * 0, -43 * 1, 42 * 2, -43 * 3, -1]))
```

We could implement a neural network that can operate on different kinds of data (e.g. discrete versus continuous) using a cocartesian diagram instead.

1.4.5 Biproducts

A category has *biproducts* if the cartesian and cocartesian structures coincide, a †-category has †-*biproducts* when furthermore the monoid is the dagger of the comonoid. This is the case in the †-category $\mathbf{Mat}_{\mathbb{S}}$ with direct sum $\oplus$ as tensor.

Listing 1.4.39. Implementation of †-biproducts for $\mathbf{Mat}_{\mathbb{S}}$ and $\mathbf{Tensor}_{\mathbb{S}}$ as a rig category with direct sum and Kronecker product.

```
@Copyable
class Matrix:
    ...
    @classmethod
    def copy(cls, x: int, n: int, is_dagger=False) -> Matrix:
        dom, cod = (x ** n, x) if is_dagger else (x, x ** n)
        inside = [[i % n == j for j in range(x)] for i in range(x ** n)] if is_dagger\
            else [[i == j % n for j in range(x ** n)] for i in range(x)]
        return cls(inside, dom, cod)

    @classmethod
    def merge(cls, x: int, n: int, is_dagger=False) -> Matrix:
        return cls.copy(x, n, not is_dagger)

    @classmethod
    def basis(cls, x: int, i: int, is_dagger=False) -> Matrix:
        inside = [i % j == 0 for j in range(x)]
        inside = [[val] for val in inside] if is_dagger else [inside]
        dom, cod = (x, x ** 0) if is_dagger else (x ** 0, x)
        return cls(inside, dom, cod)

class Tensor:
    ...
    @inductive
    def direct_sum(self, other: Tensor) -> Tensor:
        old = self.downgrade().direct_sum(other.downgrade())
        return type(self)(old.inside, [old.dom], [old.cod])

    __or__ = direct_sum
```

Biproducts and matrices happen to be intimately related. Indeed, given any category C with sums we can construct its *free biproduct completion* $\mathbf{Mat}_C$ as the monoidal category with objects given by $C_0^\star$ and arrows $f : x \to y$ given by matrices $f_{ij} : x_i \to y_j$ of arrows in C_1. Composition in $\mathbf{Mat}_C$ is an extension of the usual

matrix multiplication with composition as product. In particular, if $C = \mathbb{S}$ is a rig, i.e. a one-object category with sums, then this definition coincides with the usual one. Any category with biproducts also has sums $f + g : x \to y$ given by $\texttt{split}(x) \fatsemi f \oplus g \fatsemi \texttt{merge}(x)$ and a zero morphism $0 = \texttt{counit}(x) \fatsemi \texttt{unit}(y)$. One can verify that the free completion is indeed the left-adjoint to the forgetful functor from biproducts to sums, see [Mac71, Exercise VIII.2.6]. Similarly, if C is a †-category then $\mathbf{Mat}_C$ is its free †-biproduct completion with the element-wise dagger of the transpose. If C is also a monoidal category, then $\mathbf{Mat}_C$ is a rig category with the tensor given by an extension of the usual Kronecker product with tensor as product.

DisCoPy implements free †-biproduct completion as a subclass of `Matrix[Diagram]` with `list[Ty]` as objects and addition given by the formal sum of diagrams. We use Python's duck typing to lift the code for composition, tensor and direct sum from `int` to `list[Ty]`. We also use a `contextmanager` to temporarily replace the multiplication of two `Diagram` entries. Again, we override equality so that diagrams are equal to the matrix of just themselves. The implementation of biproduct-preserving functors requires some work: given a matrix of diagrams, we construct its image as the sum of the images of its entries, pre- and post-composed by the basis row and column vectors. The `basis` subroutine constructs basis vectors for an arbitrary class using only its methods `id`, `zero`, `copy` and `direct_sum`.

Listing 1.4.40. Implementation of free †-biproduct completion.

```
@dataclass
class FakeInt:
    inside: list[Ty] = [Ty()]

    __int__ = lambda self: len(self.inside)
    __iter__ = property(lambda self: self.inside.__iter__)
    __add__ = lambda self, other: FakeInt(self.inside + other.inside)
    __mul__ = lambda self, other: FakeInt([x0 @ x1 for x0 in self.inside for x1 in other])
    __pow__ = lambda self, n: product(n * [self], unit=FakeInt())

class Biproduct(Matrix):
    dtype = Diagram

    def __init__(self, inside: list[list[Diagram]], dom: list[Ty], cod: list[Ty]):
        self.dom, self.cod, self.inside = list(dom), list(cod), [[
            cls.dtype.id(x) if val is 1 and x == y
            else cls.dtype.zero(x, y) if val is 0
            else val for y, val in zip(cod, row)] for x, row in zip(dom, inside)]
```

```
    @contextmanager
    def fake_multiplication(self, method):
        tmp, self.dtype.__mul__ = getattr(self.dtype, "__mul__", None), method
        self.dom, self.cod = map(FakeInt, (self.dom, self.cod))
        yield self
        self.dom, self.cod = map(list, (self.dom, self.cod))
        delattr(self.dtype, "__mul__") if tmp is None else setattr(self.dtype, "__mul__", tmp)

    @classmethod
    def upgrade(cls, old: Diagram):
        if isinstance(old, cls): return old
        return cls([[old]], [old.dom], [old.cod])

    @inductive
    def then(self, other: Biproduct | Diagram) -> Biproduct:
        with self.fake_multiplication(self.dtype.then) as self:
            return Matrix.then(self, self.upgrade(other))

    @inductive
    def tensor(self, other: Biproduct | Diagram) -> Biproduct:
        with self.fake_multiplication(self.dtype.tensor) as self:
            return Matrix.Kronecker(self, self.upgrade(other))

    @classmethod
    def copy(cls, x: list[Ty], n: int, is_dagger=False):
        with cls.id(x).fake_multiplication(cls.then) as biproduct:
            return Matrix.copy(biproduct.dom, n, is_dagger)

    @classmethod
    def discard(cls, x, is_dagger=False): return cls.copy(x, 0, is_dagger)

    dagger = lambda self: self.transpose().map(lambda f: f.dagger())
    __or__ = lambda self, other: self.direct_sum(self.upgrade(other))
    __eq__ = lambda self, other: Matrix.__eq__(self, self.upgrade(other))

for method in ("self", "other"):
    setattr(Diagram, method, inductive(
        lambda self, other: getattr(self.biproduct.upgrade(self), method)(other)
        if isinstance(other, Biproduct) else getattr(monoidal.Diagram, method)(other)))
Diagram.direct_sum = lambda self, *others: self.biproduct.upgrade(self).direct_sum(*other)
Diagram.__or__ = Diagram.direct_sum
Diagram.__eq__ = lambda self, other:\
    other.inside = [[self]] if isinstance(other, Biproduct)\
    else monoidal.Diagram.__eq__(self, other)
```

```
@classmethod
def basis(cls, x: list[Ty], i: int):
    terms = [cls.id(x[i]) if i == j else cls.zero(x[i], x[j]) for j in range(len(x))]
    return cls.copy(x[i], len(x)) >> cls.direct_sum(*terms)

class Functor(monoidal.Functor):
    dom = cod = Category(list[Ty], Biproduct)

    def __call__(self, other):
        if isinstance(other, Biproduct):
            result = self(other.zero(other.dom, other.cod))
            for i, row in enumerate(other.inside):
                effect = basis(self.cod.ar, list(map(self, other.dom)), i)[::-1]
                for j, diagram in enumerate(row):
                    state = basis(self.cod.ar, list(map(self, other.cod)), j)
                    result += effect >> self(diagram) >> state
            return result
        return super().__call__(other)
```

Example 1.4.41. *We can define the object* `bit` *as the list of two empty types, with* `true` *and* `false` *the two basis states then we can implement conditional expressions as biproducts.*

```
bit = 2 * [Ty()]
true = Biproduct.copy(Ty(), 2) >> Diagram.id(Ty()) | Diagram.zero(Ty(), Ty())
false = Biproduct.copy(Ty(), 2) >> Diagram.zero(Ty(), Ty()) | Diagram.id(Ty())

x, y = Ty('x'), Ty('y')
f, g = Box('f', x, y), Box('g', x, y)
conditional = f | g >> Biproduct.merge(y, 2)

assert true @ x >> conditional == f and false @ x >> conditional == g
```

Example 1.4.42. *We can implement classical control as a biproduct of two quantum states and measurement as a biproduct of two quantum effects. When we compose classical control with measurement, we get a matrix where the entries are scalar diagrams. The squared amplitude of the evaluation of these scalars give us the measurement probabilities for each classical choice of state.*

```
control = Biproduct([[Ket(0)], [Ket(1)]], bit, [qubit])
measure = Biproduct([[Bra(0), Bra(1)]], [qubit], bit)
```

```
F = Functor(
    ob={qubit: 2}, ar=lambda f: f.eval(),
    cod=Category(list[int], Tensor[complex]))

assert F(control >> H >> measure).inside\
    == [[(Ket(i) >> H >> Bra(j)).eval() for j in [0, 1]] for i in [0, 1]]
```

As we mentioned at the end of section 1.2, DisCoPy uses a *point-free* syntax and it can be rather tedious to define any complex diagram in this way. It is straightforward to extend the `diagramize` method to cartesian diagrams, so that they can be defined using the standard syntax for Python functions, where we can use arguments any number of times in any order. Extending it to cocartesian diagrams so that they can be defined using the standard Python syntax for conditionals will likely be more challenging. Given enough engineering, it would be possible to turn any pure Python function into a diagram, however this will require more structure than just (co)cartesian categories. Functions with side effects can be seen as arrows in *premonoidal categories* which are the topic of section 1.5, while recursive functions are arrows in *traced categories*, both will be discussed in section 1.5. Higher-order functions are modeled as arrows in *closed categories*, the topic of the next section.

1.4.6 Closed categories

As we have seen in sections 1.4.3 and 1.4.4, cartesian categories like **Pyth** and hypergraph categories like **Tensor** are two orthogonal extensions of monoidal categories. The former have natural comonoids on each object, the latter have spiders on each object, an object that has both is necessarily trivial. Nevertheless, the category **Pyth** does share a common structure with rigid categories beyond being monoidal: both are *closed* monoidal categories. A monoidal category C is left-closed if for every object $x \in C_0$, the functor $x \otimes - : C \to C$ has a right adjoint $x/- : C \to C$ called x *over* $-$. Symmetrically, C is right-closed if the functor $- \otimes x : C \to C$ has a right adjoint $-\backslash x : C \to C$ called $-$ *under* x. A *closed (monoidal) category* is one that is closed on the left and the right. For example, a rigid category is a closed category where the over and under types have the form $x/y = x^r \otimes y$ and $y/x = y \otimes x^l$. When the category is symmetric, over and under types coincide, they are called *exponentials* $x/y = y\backslash x = y^x$.

Example 1.4.43. *A discrete monoidal category (i.e. a monoid) is closed if and only if it is a group. A closed preordered monoid (i.e. a closed category with at*

most one arrow between any two objects) is also called a residuated monoid *[Coe13], their application to NLP will be discussed in section 2.1.*

As the name suggests, a *cartesian closed category* is a cartesian category that is also closed. Examples of cartesian closed categories include **Set** with the exponential Y^X given by the set of functions from X to Y and **Cat** with D^C the category of functors from C to D with natural transformations as arrows. The category **Pyth** with `list[type]` as objects and pure functions between tuples as arrows is also cartesian closed, the exponential of two lists of types `x, y` is given by `Callable[x, tuple[y]]`. The natural isomorphism $\Lambda : \mathbf{Pyth}(x \times y, z) \to \mathbf{Pyth}(y, z^x)$ is called *currying*, after the founding father of functional programming Haskell Curry. A function of two arguments $x \times y \to z$ is the same as a one-argument higher-order function $y \to z^x$. Taking the equivalent definition of adjunctions, the unit $\eta_y : y \to (y \times x)^x$ is given by concatenation, i.e. `lambda *ys: lambda *xs: ys + xs`, while the counit $\epsilon_y : y^x \times x \to x$ is given by evaluation, i.e. `lambda f, *xs: f(*xs)`. In fact, we can take the data for a cartesian closed category to be that of a cartesian category C together with:

- an operation $\exp : C_0^\star \times C_0^\star \to C_0$ sending every pair of types to a generating object,
- an operation $\mathtt{ev} : C_0^\star \times C_0^\star \to C_1$ sending every pair of types x, y to an arrow $\mathtt{ev}(x, y) : \exp(y, x) \times x \to y$,
- an operation $\Lambda_n : C_1 \to C_1$ for each $n \in \mathbb{N}$, sending every arrow $f : x \times y \to z$ with x of length n to an arrow $\Lambda_n(f) : y \to \exp(z, x)$.

We can take the axioms to be those of cartesian categories together with $\Lambda_n(f \times x \fatsemi \mathtt{ev}(z, x)) = f$ for all $f : y \to \exp(z, x)$. Intuitively, if we take a higher-order function, evaluate it then abstract away the result, we get back to where you started, i.e. $\Lambda_n^{-1}(f) = f \times x \fatsemi \mathtt{ev}(z, x)$.

Listing 1.4.44. Implementation of **Pyth** as a cartesian closed category.

```
def exp(base: list[type], exponent: list[type]) -> list[type]:
    return [Callable[list(exponent), tuple[tuple(base)]]]

class Function:
    ...
    def curry(self, n=1, left=True) -> Function:
        inside = lambda *xs: lambda *ys: self(*(ys + xs) if left else (xs + ys))
        dom, cod = (self.dom[n:], exp(self.cod, self.dom[:n])) if left\
```

```
            else (self.dom[:len(self.dom) - n], exp(self.cod, self.dom[len(diagram) - n:]))
        return Function(inside, dom, cod)

    @staticmethod
    def ev(base: list[type], exponent: list[type], left=True) -> Function:
        inside = (lambda *xs: xs[-1](*xs[:-1])) if left else (lambda f, *xs: f(*xs))
        dom, cod = (exponent + exp(base, exponent), base) if left\
            else (exp(base, exponent) + exponent, base)
        return Function(inside, dom, cod)

    def uncurry(self, left=True) -> Function:
        if len(self.cod) != 1 or not self.cod[0]._name == "Callable": return self
        *exponent, output = self.cod[0].__args__
        base = output.__args__ if is_tuple(output) else [output]
        return exponent @ self >> Function.ev(base, exponent) if left\
            else self @ exponent >> Function.ev(base, exponent, left=False)

    under, over = staticmethod(exp), staticmethod(lambda x, y: exp(y, x))
```

Example 1.4.45. *We can check the axioms for cartesian closed categories hold in* **Pyth**.

```
x, y, z = map(list, (complex, bool, float))
f = Function(dom=y, cod=exp(z, x), inside=lambda b: lambda a: abs(a) ** 2 if b else 0)

assert (x @ f >> Function.ev(z, x)).curry()(True)(1j) == f(True)(1j)\
    == (f @ x >> Function.ev(z, x, left=False)).curry(left=False)(True)(1j)
```

A (strict) cartesian closed functor is a cartesian functor F which respects the exponential, i.e. $F(y^x) = F(y)^{F(x)}$. Thus, we get a category **CCCat** of cartesian closed categories and functors, with a forgetful functor $U : \mathbf{CCCat} \to \mathbf{MonSig}$. Its left adjoint $F^{CC} : \mathbf{MonSig} \to \mathbf{CCCat}$ can be constructed in two steps. First, we define a *closed signature* as a monoidal signature Σ with a pair of binary operators $(-/-)$, $(-\backslash-) : \Sigma_0^\star \times \Sigma_0^\star \to \Sigma_0$, i.e. for every pair of types $x, y \in \Sigma_0^\star$ there are two generating objects x/y, $y\backslash x \in \Sigma_0$. Morphisms of closed signatures are defined in the obvious way, thus we get a category **CSig** with two forgetful functors $U : \mathbf{CCCat} \to \mathbf{CSig}$ and $U : \mathbf{CSig} \to \mathbf{MonSig}$. Its left-adjoint $F^C : \mathbf{MonSig} \to \mathbf{CSig}$ sends a monoidal signature to the union of itself with extra objects for the over and under slashes, it can be defined by induction on the number of nested slashes. Now we can define the left adjoint $F : \mathbf{CSig} \to \mathbf{CCCat}$ as a quotient of the free cartesian category with boxes for evaluations, bubbles

for currying (i.e. by induction on the number of nested curryings) and relations given by the axioms for natural isomorphisms. Composing the two adjunctions we get $F^{CC} : \mathbf{MonSig} \to \mathbf{CCCat}$.

Diagrams in free cartesian closed categories can also be seen as terms of the *simply typed lambda calculus* (up to $\beta\eta$-equivalence) or as proofs in *minimal logic* (the fragment of propositional logic with only conjunction and implication). This is called the *Curry-Howard-Lambek correspondance*, see Abramsky and Tzevelekos [AT10] for an introduction. The problem of deciding the $\beta\eta$-equivalence of a given pair of simply typed lambda terms (or equivalently the equivalence of two minimal logic formulae) is decidable [Tai67] but not elementary recursive [Sta79], i.e. its time complexity is not bounded by any tower of exponentials. As such, the word problem for free cartesian closed categories is as intractable as it gets. If we remove the copying and discarding, the word problem for free symmetric closed categories is decidable in linear time [Vor77]. The algorithm is based on a variant of Gentzen's cut-elimination theorem for a *substructural logic*, one where the *weakening* and *contraction* rules are omitted, i.e. we cannot discard or copy assumptions. To the best of our knowledge, the case of (non-symmetric) closed monoidal categories is still open. We conjecture it can be solved with a variant of cut-elimination for a *non-commutative logic* omitting the *exchange* rule, i.e. we cannot swap assumptions.

DisCoPy implements the types of the closed diagrams with a subclass `Exp` of `Ty` for exponentials. `Over` and `Under` are two subclasses of `Exp`, shortened to `x >> y` and `y << x`, and attached to the `Diagram` class as two static methods `over` and `under`. We need to initalise the type `self: Exp` with some list of objects, our only choice is `self.inside=[self]`. Thus, we need to override the equality and printing methods so that we don't fall into infinite recursion. We also need to override the `upgrade` method so that the unit law is satisfied, i.e. `self @ Ty() == self == Ty() @ self`.

Listing 1.4.46. Implementation of the types in free closed categories.

```
class Ty(monoidal.Ty):
    @classmethod
    def upgrade(cls, old: monoidal.Ty) -> Ty:
        return old[0] if len(old) == 1 and isinstance(old[0], Exp) else cls(*old.inside)

    def __pow__(self, other):
        return Exp(self, other) if isinstance(other, Ty) else super().__pow__(other)

class Exp(Ty):
    upgrade = Ty.upgrade
```

```
    def __init__(self, base, exponent):
        self.base, self.exponent = base, exponent
        super().__init__(inside=[self])

    def __eq__(self, other):
        return isinstance(other, type(self))\
            and (self.base, self.exponent) == (other.base, other.exponent)

    __str__ = lambda self: "({}) ** ({})".format(self.base, self.exponent)

class Under(Exp):
    __str__ = lambda self: "({} << {})".format(self.base, self.exponent)

class Over(Exp):
    __str__ = lambda self: "({} >> {})".format(self.exponent, self.base)

Ty.__lshift__, Ty.__rshift__ = Under, lambda self, other: Over(other, self)
Diagram.over, Diagram.under = map(staticmethod, (Over, Under))
```

Closed diagrams are implemented with two subclasses of `Box` for currying and evaluation, which are attached to the `Diagram` class as two static methods `curry` and `ev`. We shorten `Diagram.ev(base, exponent, left)` to `Ev(exponent >> base)` if `left` else `Ev(base << exponent)`. Closed functors map `Exp` types to the `over` and `under` methods of their codomain, similarly for `Curry` and `Uncurry` boxes.

Listing 1.4.47. Implementation of free closed categories and functors.

```
class Ev(Box):
    def __init__(self, inside: Exp):
        self.base, self.exponent, self.left\
            = inside.base, inside.exponent, isinstance(inside, Over)
        dom, cod = self.exponent @ inside, self.base if self.left\
            else inside @ self.exponent, self.base
        super().__init__("Ev" + str(exp), dom, cod)

class Curry(Box):
    def __init__(self, inside: Diagram, n=1, left=True):
        self.inside, self.n, self.left = inside, n, left
        name = "Curry({}, {}, {})".format(diagram, n, left)
        dom, cod = (inside.dom[n:], inside.dom[:n] >> inside.cod) if left\
            else (inside.dom[:len(inside.dom) - n],
                  inside.cod << inside.dom[len(inside.dom) - n:])
        super().__init__(name, dom, cod)
```

```
def uncurry(self: Diagram, left=True) -> Diagram:
    if not isinstance(self.cod, (Over if left else Under)): return self
    base, exponent = self.cod.base, self.cod.exponent
    return exponent @ self >> Ev(exponent >> base) if left\
        else self @ exponent >> Ev(base << exponent)

@staticmethod
def ev(base: Ty, exponent: Ty, left=True) -> Ev:
    return Ev(exponent >> base if left else base << exponent)

Diagram.curry, Diagram.uncurry, Diagram.ev = Curry, uncurry, ev

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

    def __call__(self, other):
        if isinstance(other, Over):
            return self.cod.ar.over(self(other.exponent), self(other.base))
        if isinstance(other, Under):
            return self.cod.ar.over(self(other.base), self(other.exponent))
        if isinstance(other, Curry):
            return self.cod.ar.curry(
                self(other.inside), len(self(other.cod.exponent)), other.left)
        if isinstance(other, Ev):
            return self.cod.ar.ev(self(other.base), self(other.exponent), other.left)
        return super().__call__(other)
```

Example 1.4.48. *We can check that the axioms hold by applying functors into* **Pyth**.

```
x, y, z = map(Ty, "xyz")
f, g = Box('f', y, z << x), Box('g', y, z >> x)

F = Functor(
    ob={x: complex, y: bool, z: float},
    ar={f: lambda b: lambda a: abs(a) ** 2 if b else 0,
        g: lambda b: lambda a: a + 1j if b else -1j}
    cod=Category(list[type], Function))

assert F((f @ x >> Ev(z << x)).curry(left=False))(True)(1j) == F(f)(True)(1j)
assert F((x @ g >> Ev(z >> x)).curry())(True)(1.2) == F(g)(True)(1j)
```

Understanding the relationship between closed and rigid categories will also explain how we draw closed diagrams, using the bubble notation introduced by

Baez and Stay [BS09, Section 2.6]. Indeed, in a free rigid category the exponentials are given as a tensor of a type and an adjoint, thus we can draw them as two wires side by side. On the other hand, the exponentials of a free closed category are defined as generating objects, they ought to be drawn as one wire but we can decide to draw them as two, inseparable wires. This constraint can be materialised by a *clasp* that binds the two wires together. Similarly, in free rigid categories we can draw evaluation and currying as diagrams with cups and caps while in a free closed category they are defined as generating boxes, which ought to be drawn as black boxes. We can decide to draw them the same way as in a rigid category, with a bubble surrounding them to prohibit illicit rewrites. Once drawn in this way, the equations for currying become a special case of the snake equations, although in general closed categories do not have boxes for cups and caps.

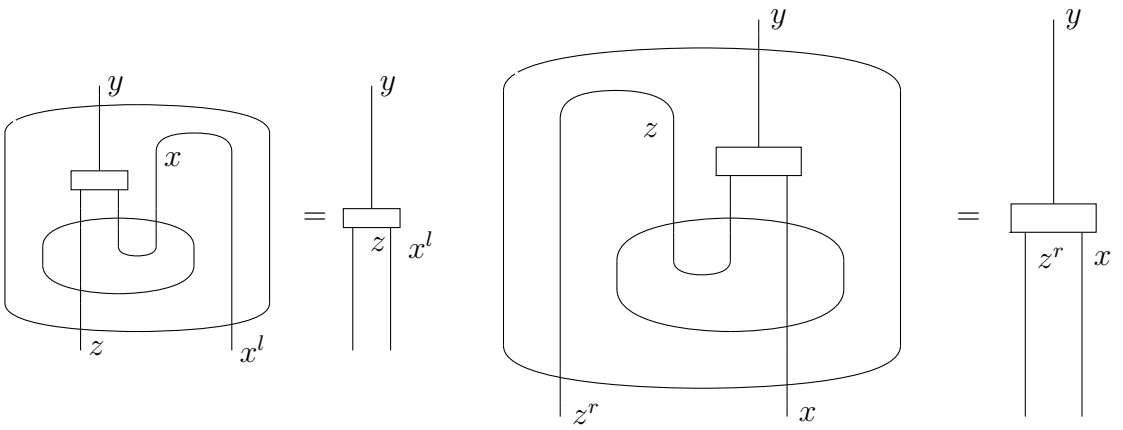
Listing 1.4.49. Implementation of free rigid categories as closed categories.

```
rigid.Ty.__lshift__ = lambda self, other: self @ other.l
rigid.Ty.__rshift__ = lambda self, other: self.r @ other
rigid.Diagram.under = staticmethod(lambda base, exponent: base << exponent)
rigid.Diagram.over = staticmethod(lambda base, exponent: exponent >> base)

@classmethod
def ev(cls, base: rigid.Ty, exponent: rigid.Ty, left=True) -> rigid.Diagram:
    return cls.cups(exponent, exponent.l) @ cls.id(base) if left\
        else cls.id(base) @ cls.cups(exponent.r, exponent)

def curry(self: rigid.Diagram, n=1, left=True) -> rigid.Diagram:
    base, exponent = (self.dom[n:], self.dom[:n]) if left\
        else (self.dom[len(self.dom) - n:], self.dom[:len(self.dom) - n])
    return self.caps(exponent.r, exponent) @ base >> exponent.r @ self if left\
        else base @ self.caps(exponent, exponent.l) >> self @ @ exponent.l

Diagram.ev, Diagram.curry = ev, curry
```

Example 1.4.50. *We can draw closed diagrams by applying a functor to a rigid category with bubbled evaluation and currying.*

```
class ClosedDrawing(rigid.Diagram):
    ev = staticmethod(
        lambda base, exponent, left=True: rigid.Diagram.ev(base, exponent, left).bubble())
    curry = lambda self, n=1, left=True: rigid.Diagram.curry(self, n, left).bubble()

Draw = Functor(lambda x: x, lambda f: f, cod=Category(rigid.Ty, ClosedDrawing))
```

```
Diagram.draw = lambda self, **params: Draw(self).draw(**params)

drawing.equation((f @ x >> Ev(z << x)).curry(left=False), f)
drawing.equation((x @ g >> Ev(z >> x)).curry(), g)
```



1.5 A premonoidal approach

In the previous section, we have seen that cartesian closed categories give us enough syntax to interpret (simply typed) lambda terms. Thus, we can execute the diagrams in a free cartesian closed category as functions by applying a functor into **Set** or **Pyth**, we can also interpret them as functors by applying a functor into **Cat** with the cartesian product as tensor. Now if we remove the cartesian assumption, the diagrams of free closed categories give us a programming language with higher-order functions where we cannot copy, discard or even swap data: the (non-commutative) *linear lambda calculus*. With a more restricted language, we get a broader range of possible interpretations. For example, there can be only one cartesian closed structure on **Cat** (because any other would be naturally isomorphic) but are there any other monoidal closed structures? Foltz, Lair and Kelly [FLK80] answer the question with the positive: **Cat** has exactly two closed structures: the usual cartesian closed structure with the exponential D^C given by the category of functors $C \to D$ and natural transformations, and a second one where the exponential $C \Rightarrow D$ is given by the category of functors $C \to D$ and *transformations*, with no naturality requirement.

The corresponding tensor product on **Cat**, i.e. the left-adjoint $C\square - \dashv C \Rightarrow -$, is called the *funny tensor product*, maybe because mathematicians thought it was funny not to require naturality. More explicitly, the funny tensor $C\square D$ can be described as the push-out of $C \times D_0 \leftarrow C_0 \times D_0 \to C_0 \times D$ where C_0, D_0 are the discrete categories of objects, or equivalently as a quotient of the coproduct

$(C \times D_0 + C_0 \times D)/R$ where the relations are given by $(0, \mathtt{id}(x), y) = (1, x, \mathtt{id}(y))$. Even more explicitly, the objects of $C \square D$ are given by the cartesian product $C_0 \times D_0$, the arrows are alternating compositions $(0, f_1, y_1) \fatsemi (1, x_2, g_2) \fatsemi \cdots \fatsemi (0, f_{n-1}, y_{n-1}) \fatsemi (1, x_n, g_n)$ of arrows in one category paired with an object of the other. When the two categories are in fact monoids, the funny tensor is called the *free product* because it sends free monoids to free monoids, i.e. $X^\star \square Y^\star = (X + Y)^\star$. This is also true for free categories, i.e. $F(\Sigma) \square F(\Sigma') = F(\Sigma \times \Sigma'_0 \cup \Sigma_0 \times \Sigma)$, so maybe $\square$ should be called free rather than funny. While a functor on a cartesian product $F : C \times D \to E$ can be seen as a functor of two arguments that is functorial in both simultaneously, i.e. $F(f \fatsemi f', g \fatsemi f') = F(f, g) \fatsemi F(f', g')$, a functor on a funny product $F : C \square D \to E$ is functorial separately in its C and D arguments, i.e. $F(0, f \fatsemi f', y) = F(0, f, y) \fatsemi F(0, f', y)$ and $F(1, x, g \fatsemi g') = F(1, x, g) \fatsemi F(1, x, g')$.

1.5.1 Premonoidal categories & state constructions

Now recall that a (strict) monoidal category C is a monoid in $(\mathbf{Cat}, \times)$, i.e. the tensor is a functor on a cartesian product $\otimes : C \times C \to C$. In a similar way, we define a (strict) *premonoidal category* as a monoid in $(\mathbf{Cat}, \square)$, i.e. a category C with an associative, unital functor on the funny product $\boxtimes : C \square C \to C$. A (strict) *premonoidal functor* is a functor that commutes with $\boxtimes$, i.e. $F(f \boxtimes g) = F(f) \boxtimes F(g)$, thus we get a category **PreMonCat**. Similarly a *premonoidal transformation* is a transformation $\alpha : F \to G$ between two premonoidal functors such that $\alpha(x \boxtimes y) = \alpha(x) \boxtimes \alpha(y)$. As for monoidal categories, we can show that every premonoidal category is premonoidally equivalent to a foo one, i.e. where the monoid of objects is free, thus we get a forgetful functor $U : \mathbf{PreMonCat} \to \mathbf{MonSig}$. The image of $\boxtimes$ on objects may be given by concatenation, its image on arrows is called *whiskering*, it is denoted by $\boxtimes(0, f, x) = f \boxtimes x$ and $\boxtimes(1, x, f) = x \boxtimes f$. As we have seen in section 1.2, from whiskering we can define a (biased) tensor product on arrows $f \boxtimes g = f \boxtimes \mathtt{dom}(g) \fatsemi \mathtt{cod}(f) \boxtimes g$ and conversely, we can define whiskering as tensoring with identity arrows.

Thus, we can take the data for a premonoidal category C to be the same as that of a foo-monoidal category and the only axioms to be those for $(C_1, \boxtimes, \mathtt{id}(1))$ being a monoid. That is, a premonoidal category is almost a monoidal category, only the interchange law does not necessarily hold. Every monoidal category (functor) is also a premonoidal category (functor), hence we have an inclusion functor $\mathbf{MonCat} \hookrightarrow \mathbf{PreMonCat}$. An arrow of a premonoidal category C is called *central* if it interchanges with every other arrow, a transformation is called

central if every component is central. Every identity is central and composition preserves centrality, thus we can define the *center* $Z(C)$ as the subcategory of central arrows and show that $Z : \mathbf{PreMonCat} \to \mathbf{MonCat}$ is in fact the right adjoint of the inclusion. A *symmetric premonoidal category* is a premonoidal category with a central natural isomorphism $S : x \boxtimes y \to y \boxtimes x$ such that the hexagon equations hold and $S(x, 1) = \mathtt{id}(x) = S(1, x)$. Again, we get an inclusion functor from symmetric monoidal categories to symmetric premonoidal, and its right adjoint given by the center.

Example 1.5.1. *A premonoidal category with one object is just a set with two monoid structures. They do not satisfy the interchange law so the Eckmann-Hilton argument does not apply, the two monoids need not coincide nor be commutative. In this case, the notion of center coincides with the usual notion of center of a monoid, i.e. the submonoid of elements that commute with everything else. Indeed, the monoidal center of a one-object premonoidal is the intersection of the centers of its two monoid structures.*

Example 1.5.2. *For any small category C, the category $C \Rightarrow C$ of endofunctors $C \to C$ with (not-necessarily-natural) transformations as arrows is premonoidal.*

Example 1.5.3. *The category of matrices $\mathbf{Mat}_{\mathbb{S}}$ with entries in a rig $\mathbb{S}$ with the Kronecker product as tensor is a premonoidal category, it is monoidal precisely when $\mathbb{S}$ is commutative.*

Example 1.5.4. *The category* **Pyth** *with* `list[type]` *as objects and* `Function` *as arrows is premonoidal with* `tuple` *as tensor. Every pure function is in the center $Z(\mathbf{Pyth})$, but the converse is not necessarily true: take the side effect $f : x \to 1$ which increments a private, internal counter every time it is called. It is impure, but not enough that we can observe it by parallel composition, i.e. although it does not commute with copy and discard, it can still be interchanged with any other function. In other words, it is in the monoidal center, but not the cartesian center (i.e. the subcategory of comonoid homomorphisms).*

Premonoidal categories were introduced by Power and Robinson [PR97] as a way to model programming languages with side effects, reformulating an earlier framework of Moggi [Mog91] which captured notions of computation as *monads*. Our last two examples can be seen as special cases of a more general pattern: they are *Kleisli categories* for a *strong monad*. Infamously, a monad is just a monoid $T : C \to C, \mu : T \fatsemi T \to T, \eta : 1 \to T$ in the category C^C of endofunctors with natural

transformations as arrows. Its Kleisli category $K(T)$ has the same objects as C and arrows given by $K(T)(x, y) = C(x, T(y))$, with the identity given by the unit $\mathtt{id}_K(x) = \eta(x)$ and composition given by post-composition with the multiplication, i.e. $f \fatsemi_K g = f \fatsemi T(g) \fatsemi \mu(z)$ for $f : x \to T(y)$ and $g : y \to T(z)$. Now if C happens to be a monoidal category, we can ask for T to be a monoidal functor, but we also want the multiplication and unit of the monad to play well with the monoidal structure. We could ask for a *monoidal monad* where μ and η are monoidal transformations, i.e. for the monad to be a monoid in the category of monoidal endofunctors and monoidal natural transformations and show that the Kleisli category $K(T)$ inherits a monoidal structure. More generally, we can ask only for a (bi)*strong monad*, equipped with two natural transformations $\sigma(a, b) : a \otimes T(b) \to T(a \otimes b)$ and $\tau(a, b) : T(a) \otimes b \to T(a \otimes b)$ subject to sufficient conditions for the Kleisli category $K(T)$ to inherit a premonoidal structure. It is monoidal precisely when the monad is commutative, i.e. the two arrows from $T(x) \otimes T(y)$ to $T(x \otimes y)$ are equal.

Example 1.5.5. *Take the category $C =$* **FinSet** *and the* distribution monad *$T(X) = \mathbb{S}^X$ for a rig $\mathbb{S}$ with the image on arrows given by* pushforward *$T(f : X \to Y)(p : X \to \mathbb{S}) : y \mapsto \sum_{x \in f^{-1}(y)} p(x)$, the multiplication and unit induced by the rig multiplication and unit. Its Kleisli category is precisely the category $\mathbf{Mat}_S$ of matrices, i.e. functions $m : Y \to \mathbb{S}^X \simeq X \times Y \to \mathbb{S}$. One can show this is a strong monad, and it is commutative precisely when the rig is commutative.*

Example 1.5.6. *Take any closed symmetric category C and the* state monad *$T(x) = s^{s \otimes x}$ for some object s, an arrow $f : x \to y$ in the Kleisli category $K(T)$ is given by an arrow $f : s \otimes x \to s \otimes y$ in C (up to uncurrying). One can show that T is strong and thus $K(T)$ is premonoidal. When $C =$* **Set** *the state monad is a non-commutative as it gets: T is commutative if and only if s is trivial. Whiskering an arrow $f : s \otimes x \to s \otimes y$ by an object z on the left is given by pre- and post-composition with swaps $z \boxtimes f = S(s, z) \otimes x \fatsemi z \otimes f \fatsemi S(z, s) \otimes y$, whiskering on the right is easier $f \boxtimes z = f \otimes z$.*

Jeffrey [Jef97] then gave the first definition of free premonoidal categories, his construction formalises the intuition that non-central arrows are to be thought as arrows with side effects. The *state construction* takes as input a symmetric monoidal category C and an object s, and builds a symmetric premonoidal category $\mathbf{St}(C, s)$ with the same objects as C, arrows given by $\mathbf{St}(C, s)(x, y) = C(s \otimes x, s \otimes y)$ and whiskering defined as in the state monad. Intuitively, an arrow in $\mathbf{St}(C, s)$ is an arrow in C which also updates a global state encoded in the object s, which we

can draw as an extra wire passing through every box of the diagram, preventing them from being interchanged. More formally, given a monoidal signature Σ we can construct the free symmetric premonoidal category $F^{SP}(\Sigma) = \mathbf{St}(F^S(\Sigma + \{s\}), s)$ as the state construction over the free symmetric category with an extra object. We can generalise this to (non-symmetric) free premonoidal categories but we still need symmetry at least for the extra object, i.e. natural isomorphisms $S(s, x) : s \otimes x \to x \otimes s$ for each object x, subject to hexagon and unit equations.

We call this definition of the free premonoidal category as a state construction over a free monoidal category with an extra swappable object the *monoidal approach* to premonoidal categories. In what we call the *premonoidal approach* to monoidal categories, definitions go the other way around with free premonoidal categories as the fundamental notion and free monoidal categories as an interesting quotient. Indeed, we have been using the arrows of free premonoidal categories all along: they are string diagrams, defined as lists of layers without quotienting by interchanger. Equivalently, they are labeled generic progressive plane graphs up to generic deformation, i.e. with at most one box node at each height. While in the monoidal approach, string diagrams are defined as non-planar graphs and the ordering of boxes is materalised by extra wires connecting the boxes in sequence, in the premonoidal approach we take this ordering as data: boxes are in a list. This comes with an immediate advantage: equality of premonoidal diagrams can be defined in terms of equality of lists, hence it is decidable in linear time whereas equality of monoidal diagrams has quadratic complexity and equality of symmetric diagrams could be as hard as graph isomorphism. Another advantage of representing string diagrams with lists rather than graphs is that the code for functor application, i.e. the interpretation of diagrams, is a simple `for` loop rather than an elaborate graph algorithm, it requires no choices. Similarly, the algorithm for drawing premonoidal diagrams requires almost no choices, the order of wires and boxes is fixed, we can only choose their shape and the spacing between them. On the other hand, drawing graphs requires complex heuristics and graphical interfaces in order to get satisfying results.

1.5.2 Hypergraph, compact, traced, symmetric

In order to compare the two approaches, we need to say a few words about how string diagrams for symmetric categories are implemented. Recall from sections 1.4.2 and 1.4.3 that equality of diagrams in symmetric and hypergraph categories reduce to graph and hypergraph isomorphisms respectively. This can

be made explicit by implementing these diagrams as graphs and hypergraphs rather than lists of layers with explicit boxes for swaps and spiders. Given a monoidal signature Σ, a *hypergraph diagram* (also called hypergraphs with *ports*) f is given by:

- its domain and codomain $\texttt{dom}(f), \texttt{cod}(f) \in \Sigma_0^\star$,
- a list of boxes $\texttt{boxes}(f) \in \Sigma_1^\star$ from which we define:
  - $\texttt{input\_ports}(f) = \texttt{cod}(f) + \coprod_i \texttt{dom}(f_i)$,
  - $\texttt{output\_ports}(f) = \texttt{dom}(f) + \coprod_i \texttt{cod}(f_i)$,
  - and $\texttt{ports}(f) = \texttt{input\_ports}(f) + \texttt{output\_ports}(f)$
- a number of *spiders* $\texttt{spiders}(f) = n \in \mathbb{N}$ together with their list of types $\texttt{spider\_types}(f) \in \Sigma_0^n$,
- a set of *wires* $\texttt{wires}(f) : \texttt{ports}(f) \to \texttt{spiders}(f)$.

The tensor of two hypergraph diagrams is given by concatenating their domain, codomain, boxes and spiders. The composition is defined in terms of *pushouts*. Given $f : x \to y$ and $g : y \to x$ we have a *span* of functions $\texttt{spiders}(f) \leftarrow y \to \texttt{spiders}(g)$ induced by the wires from the codomain of f and the domain of g, we define $\texttt{spiders}(f \fatsemi g)$ as the size of the quotient set $(\texttt{spiders}(f) + \texttt{spiders}(g))/R$ under the relation given by the codomain wires of f and the domain wires of g. Concretely, this is computed as the reflexive transitive closure of the binary relation on $\texttt{spiders}(f) + \texttt{spiders}(g)$. The identity diagram $\texttt{id}(x)$ has $\texttt{spiders}(f) = |x|$ and wires given by the two injections $|x| + |x| \to \texttt{spiders}(f)$. Now we can define a notion of interchanger which takes a hypergraph diagram f and some index $i < |\texttt{boxes}(f)|$ and returns the diagram with boxes i and $i{+}1$ interchanged, i.e. with the wires relabeled appropriately. While the interchanger of monoidal diagrams is ill-defined when the boxes are connected, that of hypergraph diagrams is always defined. The category $\mathbf{Hyp}(\Sigma)$ with equivalence classes of hypergraph diagrams is in fact isomorphic to the free hypergraph category $F^H(\Sigma)$ which we defined in section 1.4.3 in terms of special commutative Frobenius algebras [Bon+16, Theorem 3.3]. The data structure for hypergraph diagrams has swaps and spiders built-in: they are hypergraph diagrams with no boxes.

We say a hypergraph diagram is *bijective* when each spider to be connected to either zero or two ports, so that they define a bijection $\texttt{ports}(f) \to \texttt{ports}(f)$.

The resulting subcategory is isomorphic to the free compact-closed category defined in section 1.4.1 in terms of cups and caps. Spiders connected to zero ports correspond to dimension scalars, i.e. circles composed of a cap then a cup. A hypergraph diagram is *monogamous* when each spider is connected to exactly one input port and one output port, so that they define a bijection $\texttt{output\_ports}(f) \to \texttt{input\_ports}(f)$. The subcategory of monogamous hypergraph diagrams is the free *traced symmetric category*[1], i.e. the free symmetric monoidal category C with a family of functions $\texttt{trace}_x(a, b) : C(a \otimes x, b \otimes x) \to C(a, b)$ subject to axioms that formalise the intuition that we can trace a morphism $f : a \otimes x \to b \otimes x$ by connecting its input and output x-wires in a loop [JSV96]. Every compact closed category has a trace given by cups and caps, traced monoidal categories allow to express recursion and fixed points more generally in non-rigid categories. Indeed in the case of a cartesian category such as **Pyth**, the trace can equivalently be given in terms of *fixed point operators* $\texttt{fix}_x : C(a \times x, x) \to C(a, x)$ [Sel10, Proposition 6.8]. Dually, in a cocartesian category the trace can be defined in terms *iteration operators* $\texttt{iter}_x : C(x, a + x) \to C(x, a)$. When the category has biproducts, it is sufficient to define a *repetition operator* $\texttt{repeat}_x : C(x, x) \to C(x, x)$ [Sel10, Proposition 6.11]. In the category of finite sets and relations $\mathbf{Mat}_{\mathbb{B}}$ with the direct sum as tensor, this coincides with the usual notion of reflexive transitive closure [JSV96, Proposition 6.3].

Listing 1.5.7. Implementation of the syntax for free traced categories.

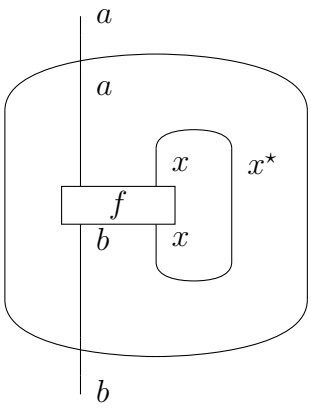
```
class Trace(Box):
    def __init__(self, inside: Diagram, n=1):
        assert inside.dom[-n:] == inside.cod[-n:]
        self.inside, name = inside, "Trace({}, {})".format(inside, n)
        super().__init__(name, inside.dom[:-n], inside.cod[:-n])

class Diagram(monoidal.Diagram):
    trace = Trace

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

    def __call__(self, other):
        if isinstance(other, Trace):
            n = len(self(other.inside.dom)) - len(self(other.dom))
```

[1] We have not been able to find a proof of this statement nor of the compact-closed case in the literature, they are a straightforward generalisation of [Bon+16, Theorem 3.3].

```
            return self.cod.ar.trace(self(other.inside), n)
        return super().__call__(other)
```

Example 1.5.8. *We can draw traced diagrams by applying a traced functor into compact-closed categories with bubbles.*

```
def compact_trace(self, n=1):
    assert self.dom[-n:] == self.cod[-n:]
    return self.dom[:-n] @ self.caps(self.dom[-n:], self.dom[-n:].r)\
        >> self @ self.dom[:n].r\
        >> self.cod[:-n] @ self.cups(self.cod[-n:], self.cod[-n:].r)

compact.Diagram.trace = compact_trace

class TracedDrawing(compact.Diagram):
    trace = lambda self, n: compact_trace(self, n).bubble()

Draw = Functor(lambda x: x, lambda f: f, cod=Category(Ty, TracedDrawing))
Diagram.draw = lambda self, **params: Draw(self).draw(**params)

a, b, x = map(Ty, "abx")
Box('f', a @ x, b @ x).draw()
```



Listing 1.5.9. Implementation of **Pyth** as a traced cartesian category.

```
class Function:
    ...
    def iter(self, n=1):
        assert self.is_additive and self.cod[-n:] == self.dom
        dom, cod = self.dom, self.cod[:-n]
        def inside(x):
            i, y = self.inside(x)
            return y if i < len(self.cod) - n else inside(y)
        return Function(inside, dom, cod)
```

```
    def fix(self, n=1):
        assert not self.is_additive and self.dom[-n:] == self.cod
        dom, cod = self.dom[:-n], self.cod
        def inside(*xs, ys=None):
            result = self.inside(*xs + (ys or []))
            return ys if result == ys else inside(*xs, ys=result)
        return Function(inside, dom, cod)

    def trace(self, n=1):
        assert self.dom[-n:] == self.cod[-n:]
        dom, cod, traced = self.dom[:-n], self.cod[:-n], self.dom[-n:]
        if self.is_additive:
            iterated = (self.unit(cod) @ traced >> self).iter()
            return dom @ self.unit(traced) >> self >> cod @ iterated >> self.merge(cod)
        fixed = (self >> self.discard(cod) @ traced).fix()
        return self.copy(dom) >> dom @ fixed >> self >> cod @ self.discard(traced)
```

Example 1.5.10. *We can compute the golden ratio as a fixed point. Note that in order to find a fixed point we need a default value to start from.*

```
phi = Function(lambda x=1: 1 + 1 / x, [int], [int]).fix()
assert phi() == (1 + sqrt(5)) / 2
```

We can define even *as a recursive function.*

```
inside = lambda n: (0, n == 0) if n < 2 else (1, n - 2)
even = Function.additive(inside, dom=[int], cod=[bool, int]).iter()
assert even(42)
```

Listing 1.5.11. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ as a traced biproduct category.

```
class Matrix:
    ...
    def repeat(self):
        assert self.dtype is bool and self.dom == self.cod
        return sum(Matrix.id(self.dom).then(*n * [self]) for n in range(self.dom + 1))

    def trace(self, n=1):
        assert self.dtype is bool
        A, B, C, D = (row >> self >> column
                      for row in [self.id(self.dom - n) @ self.unit(n),
                                  self.unit(self.dom - n) @ self.id(n)],
                      for column in [self.id(self.cod - n) @ self.discard(n),
                                     self.discard(self.cod - n) @ self.id(n)])
        return A + (B >> D.repeat() >> C)
```

A hypergraph diagram is *progressive* when it is monogamous and furthermore when the output port of box i is connected to the input port of box j we have $i < j$. An equivalent condition is that the underlying hypergraph obtained by forgetting the ports is *acyclic* and the order of boxes witnesses that acyclicity. An interchanger between boxes i and $i + 1$ is progressive when the two boxes are not connected, i.e. progressive interchangers preserve progressivity. The resulting category of progressive hypergraph diagrams up to progressive interchangers is isomorphic to the free symmetric category defined in section 1.4.2 in terms of braidings [Bon+16, Theorem 3.12]. If we remove the interchanger quotient, we get premonoidal versions of free hypergraph, compact closed, traced and symmetric categories. Traced preonoidal categories were introduced by Benton and Hyland [BH03] in order to model recursion in the presence of side-effects. To the best of our knowledge, no one has ever considered premonoidal compact closed categories: the snake equations still hold but we cannot yank the snakes away if there are obstructions, i.e. the snake removal algorithm of section 1.2 does not apply.

1.5.3 Hypergraph versus premonoidal diagrams

At every level of this symmetric-traced-compact-hypergraph hierarchy, premonoidal diagrams have the same linear-time algorithm for deciding equality, the data structure for hypergraph diagrams faithfully encode the premonoidal axioms without needing to compute any quotient: normal form becomes identity. Now suppose we want to compute the interpretation of such a hypergraph diagram in a category given only access to its methods for identity, composition, tensor, swaps and spiders. This requires to compute the isomorphism $\mathbf{Hyp}(\Sigma) \to F^H(\Sigma)$, i.e. we want to describe the given hypergraph diagram as a chosen representative in its equivalence class of premonoidal diagrams with explicit boxes for swaps and spiders. The inverse isomorphism $F^H(\Sigma) \to \mathbf{Hyp}(\Sigma)$ is computed by applying a premonoidal functor (i.e. a `for` loop on a list of layers) sending swap and spider boxes to hypergraph diagrams with no boxes.

This isomorphism is implemented in the `hypergraph` module of DisCoPy which is outlined below. The composition method calls a `pushout` subroutine which takes as input the numbers of spiders on the left and right, the wires from some common boundary ports to the left and right spiders, and returns the two injections into their pushout. The three properties for bijective, monogamous and progressive diagrams implement the subcategory of compact closed, traced and symmetric diagrams respectively. The three corresponding methods take a diagram and add explicit

boxes for spiders, cups and caps so that `f.make_bijective().is_bijective` for all `f: Diagram` and similarly for monogamous and progressive. The `downgrade` method calls `make_progressive` to construct a `compact.Diagram` with explicit boxes for swaps and spiders. The `upgrade` applies a `compact.Functor` so that `upgrade(f.downgrade()) == f` on the nose for any `f: Diagram` and `upgrade(f).downgrade()` is equal to any `f: compact.Diagram` up to the special commutative Frobenius axioms. The `draw` method uses a randomised force-based layout algorithm for graphs to compute an embedding from the hypergraph diagram to the plane where wires do not cross too much. This is still an experimental feature and the results of `f.draw()` usually look much worse than the drawing of `f.downgrade().draw()` using the deterministic algorithm of section 1.3.

Listing 1.5.12. Outline of the `discopy.hypergraph` module.

```
def pushout(left: int, right: int, left_wires: list[int], right_wires: list[int]
    ) -> tuple[dict[int, int], dict[int, int]]: ...

@dataclass
class Diagram(Composable, Tensorable):
    dom: Ty
    cod: Ty
    boxes: list[Diagram]
    wires: list[int]
    spider_types: list[Ty]

    @staticmethod
    def id(x: Ty) -> Diagram: ...
    def then(self, *others: Diagram) -> Diagram: ...
    def tensor(self, *others: Diagram) -> Diagram: ...
    def interchange(self, i: int) -> Diagram: ...

    swap: Callable[[Ty, Ty], Diagram] = staticmethod(...)
    spiders: Callable[[int, int, Ty], Diagram] = staticmethod(...)

    is_bijective: bool = property(...)
    is_monogamous: bool = property(...)
    is_progressive: bool = property(...)

    def make_bijective(self) -> Diagram: ...
    def make_monogamous(self) -> Diagram: ...
    def make_progressive(self) -> Diagram: ...

    def downgrade(self) -> compact.Diagram: ...
```

```
    upgrade = staticmethod(compact.Functor(
        ob=lambda x: Ty(x[0]),
        ar=lambda box: Box(box.name, box.dom, box.cod),
        cod=Category(Ty, Diagram)))

    def draw(self, **params): ...

class Box(Diagram):
    def __init__(self, name: str, dom: Ty, cod: Ty):
        boxes, spider_types, wires = [self], list(map(Ty, dom @ cod)), ...
        self.name = name; super().__init__(dom, cod, boxes, wires, spider_types)

    __eq__ = lambda self, other: cat.Box.__eq__(self, other)\
            if isinstance(other, Box) else super().__eq__(other)
```

In some cases however, we can compute the interpretation of hypergraph diagrams without having to downgrade them back to premonoidal diagrams. This is the case for *tensor networks*, i.e. hypergraph diagrams interpreted in the category $\mathbf{Tensor}_{\mathbb{S}}$. Indeed, rather than applying premonoidal functors into our naive `Matrix` class, DisCoPy can translate hypergraph diagrams as input to *tensor contraction* algorithms such as the Einstein summation of NumPy [vdWCV11] combined with the just-in-time compilation of JAX [Bra+20], or the specialised TensorNetwork library [Rob+19]. Another example is that of quantum circuits: they are inherently symmetric diagrams. Indeed, the data structure for circuits in a quantum compiler such as t|ket⟩ [Siv+20] is secretly some premonoidal symmetric category: objects are lists of qubit (and bit) identifiers, arrows (i.e. circuits) are lists of operations. When circuits are encoded as premonoidal diagrams as we have done in example 1.2.26, qubits are forced into a line because their wires are ordered from left to right, thus applying gates to non-adjacent qubits is encoded in terms of swap boxes. When we apply a premonoidal functor to the category of t|ket⟩ circuits, those swap boxes are not interpreted as the physical operation of applying three CNOT gates, but as the logical operation of relabeling our qubit identifiers. It is then the job of the compiler to map these symmetric diagrams (where every qubit can talk to every other) onto the architecture of the machine and potentially introduce physical swaps when a logical gate applies to physical qubits that are not adjacent.

The main advantage of representing diagrams as hypergraphs rather than lists is that we can use graph rewriting algorithms to implement quotient categories. Indeed, the *double push-out* (DPO) rewriting of Ehrig et al. [EPS73] can be extended from graphs to hypergraph diagrams so that we can match the left-hand side of an

axiom in a hypergraph diagram and then compute the substitution with the right-hand side [Bon+20]. Abstractly, DPO rewriting takes two hypegraph diagrams `self` and `pattern` and iterates through all possible `match` (i.e. pairs of diagrams for top and bottom and pairs of types for left and right as defined in section 1.2.4) such that `match.subs(pattern)` is equal to `self` up to interchanger. This can be extended to the case of symmetric diagrams by implementing a Boolean property `match.is_convex` that makes sure that pattern matching does not introduce spiders [Bon+16]. DPO rewriting has been the basis of tools such as Quantomatic and its successor PyZX [KvdW19] [KZ15] for automated diagrammatic reasoning, it is also at the core of circuit optimisation in the t|ket⟩ compiler. DisCoPy implements back and forth translations from diagrams to both PyZX and t|ket⟩, thus we can use their rewriting engines to implement quotient categories, i.e. to define normal forms.

However, the hypergraph approach breaks down in the case of non-symmetric monoidal categories. Indeed, the monoidal functor from the free monoidal category to the free symmetric category is not faithful, for example it sends two nested circles and two circles side by side to the same hypergraph diagram. We conjecture that the category of planar[1] progressive hypergraph diagrams is the free *spacial category*, i.e. one with $s \otimes x = x \otimes s$ for all scalars $s : 1 \to 1$ and objects x. Translated in terms of the topological definition of string diagrams, this would correspond to taking labeled progressive plane graphs up to deformation of three-dimensional space rather than up to deformation of the plane [Sel10, Conjecture 3.4]. When presented as quotients of free monoidal categories, spacial categories require an infinite family of axioms indexed by all possible scalar diagrams: in the absence of symmetry we cannot decompose the equation $(f \fatsemi g) \otimes x = x \otimes (f \fatsemi g)$ for a state $f : 1 \to y$ followed by an effect $g : y \to 1$ in terms of two smaller equations about f and g passing through the wire x. That every braided monoidal category is spacial follows from the naturality of the braiding, we do not know of any natural example of non-free non-braided spacial category. Moreover, it is an open question whether we can extend DPO rewriting to the case of spacial monoidal categories, i.e. whether there is an efficiently checkable condition that ensures that pattern matching does not introduce swaps.

Thus, DisCoPy's planar premonoidal approach to string diagrams allows to define diagrams in non-symmetric categories that cannot be defined as hypergraphs. Although the concrete examples of categories we have discussed so far (functions,

[1] A hypergraph diagram is planar when we can embed it in the plane, or equivalently it is the image of a swap-free premonoidal diagram.

matrices, circuits) are all symmetric, planarity is essential if we are to model grammatical structure in terms of string diagrams as we will in section 2.1. Indeed, the left to right order of wires in a planar diagram encode the chronological order of words in a sentence, allowing arbitrary swaps would make grammaticality permutation-invariant: if a sentence is grammatical, then so would be any random shuffling of it. More impotantly, planarity in grammar has been given a cognitive explanation. In order to minimise the computational resources needed by the brain, human languages tend to minimise the distance between words that are syntactically connected [FMG15] and the minimisation of swaps comes as a side-effect [Can06]. It can also be given a complexity-theoretic explanation: planar grammatical structures such as Chomsky's syntax trees, Lambek's pregroup diagrams or Gaifman's dependency trees (which we introduce in section 2.1) are all *context-free*, they have the same expressive power as *push-down automaton*. In languages such as Dutch and Swiss German, *cross-serial dependencies* are counter examples where grammatical wires are allowed to cross, albeit in a restricted way that makes them *mildly context-sensitive* [Wei88]. Yeung and Kartsaklis [YK21] showed that up to word reordering, the diagrams for these cross-serial dependencies can always be rewritten in a planar way using naturality. They then used DisCoPy to encode every sentence of Alice in Wonderland as a diagram, ready to be translated into a circuit and sent to a quantum computer.

1.5.4 Towards higher-dimensional diagrams

The premonoidal approach is also well-suited to be generalised from two-dimensional diagrams to arbitrary dimensions. The first step would be to add some colours to our diagrams, generalising them from monoidal categories to (strict) *2-categories*, or equivalently from premonoidal categories to *sesquicategories*. The data for a sesquicategory C is given by:

- the data for a category $(C_0, C_1, \mathtt{dom}_0, \mathtt{cod}_0, \fatsemi_0, \mathtt{id}_0)$ where the objects C_0 and arrows C_1 are called the class of *0- and 1-cells*,
- a class C_2 of *2-cells*,
- domain and codomain $\mathtt{dom}_1, \mathtt{cod}_1 : C_2 \to C_1$,
- an identity $\mathtt{id}_1 : C_1 \to C_2$ and a partial composition $(\fatsemi_1) : C_2 \times C_2 \to C_1$.

such that the following holds

- $C(x, y) = \{f \in C_2 \,|\, f : x \to y\}$ is a category for every pair of 1-cells $x, y \in C_1$,
- composition of 1-cells is a functor $(\fatsemi)_0 : C(x, y) \square C(y, z) \to C(x, z)$ for $\square$ the funny tensor product on **Cat**.

The axioms for 2-categories are the same but now composition is a bifunctor $\fatsemi_1 : C(x, y) \times C(y, z) \to C(x, z)$ on a cartesian product. Every bifunctor is also functorial in its two arguments separately thus every 2-category is also a sesquicategory. The canonical example of a (2-category) sesquicategory is **Cat** with categories as 0-cells, functors as 1-cells and (natural) transformations as 2-cells. Every monoidal (premonoidal) category is a 2-category (sesquicategory) with one 0-cell. A 2-functor $F : C \to D$ between two 2-categories is given by three functions $\{F_i : C_i \to D_i\}_{0 \leq i \leq 2}$ such that $(F_0, F_1) : (C_0, C_1) \to (D_0, D_1)$ and $(F_1, F_2) : C(x, y) \to D(F_1(x), F_1(y))$ are functors for all $x, y \in C_1$.

Free sesquicategories are defined in the same way as free premonoidal categories (i.e. as lists of layers) except that now every type comes itself with a domain and codomain, represented as the background colours on the left and right of the wire. Thus we need a *2-signature* $\Sigma = (\Sigma_0, \Sigma_1, \Sigma_2, \texttt{dom}, \texttt{cod})$ where

- Σ_0 is a set of colours,
- Σ_1 is a set of objects with colours as domain and codomain,
- Σ_2 is a set of boxes with domain and codomain in the free category $F(\Sigma_0, \Sigma_1)$, i.e. lists of generating objects with composable colours.

We also need to require the *globular conditions* $\texttt{dom}(\texttt{dom}(f)) = \texttt{dom}(\texttt{cod}(f))$ and $\texttt{cod}(\texttt{dom}(f)) = \texttt{cod}(\texttt{cod}(f))$ that ensure that the top-left (top-right) colour is the same as the bottom-left (bottom-right, respectively). Intuitively, the only changes in background colour happen at the wires, labeled by a generating object with the appropriate domain and codomain. The free 2-category $F^{2C}(\Sigma)$ can then be described by its set of *0-cells* Σ_0 (the colours), its category of *1-cells* $F(\Sigma_0, \Sigma_1)$ (the types) and a category for every pair of 1-cells: the category of coloured diagrams up to interchanger. The implementation is straightforward: we just need to make `Ty` a subclass of both `monoidal.Ty` (so that it can be used as domain and codomain for diagrams) and `Arrow` (so that it can have a domain and codomain itself).

Listing 1.5.13. Implementation of the free sesquicategory with `Colour` as 0-cells, `Ty` as 1-cells and `Diagram` as 2-cells.

```
class Colour(cat.Ob):
    def __init__(self, name="white"):
```

```
        super().__init__(name)

class Ty(cat.Arrow, monoidal.Ty):
    def __init__(self, inside: cat.Arrow | str, dom=Colour(), cod=Colour()):
        inside = inside if isinstance(inside, Arrow) else cat.Box(inside, dom, cod)
        Arrow.__init__(self, inside.inside, inside.dom, inside.cod)

    @classmethod
    def upgrade(cls, old: Arrow | monoidal.Ty) -> Ty:
        return old if isinstance(old, cls) else cls(
            old.objects if isinstance(old, monoidal.Ty) else old)

    tensor = Arrow.then

class Layer(monoidal.Layer): pass

class Diagram(monoidal.Diagram): pass

class Box(monoidal.Box, Diagram):
    def __init__(self, name: str, dom: Ty, cod: Ty):
        assert (dom.dom, dom.cod) == (cod.dom, cod.cod)
        monoidal.Box.__init__(self, name, dom, cod)

@dataclass
class TwoCategory(Category):
    colours: type

@dataclass
class TwoFunctor(monoidal.Functor):
    colours: dict[Ob, Ob]

    dom = cod = TwoCategory(colours=Colour, ob=Ty, ar=Diagram)

    def __call__(self, other):
        if isinstance(other, (Diagram, Layer)): return super().__call__(other)
        dom, cod = (Category(C.colours, C.ob) for C in [self.dom, self.cod])
        return cat.Functor(ob=self.colours, ar=self.ob, dom=dom, cod=cod)(other)
```

Listing 1.5.14. Implementation of **Cat** as a sesquicategory with transformations as 2-cells.

```
class Transformation(Composable, Tensorable):
    def __init__(self, inside: Callable, dom: Functor, cod: Functor):
        assert (dom.dom, dom.cod) == (cod.dom, cod.cod)
```

```
        self.inside, self.dom, self.cod = inside, dom, cod

    @staticmethod
    def id(F: Functor):
        return Transformation(lambda x: F.cod.ar.id(x), dom=F, cod=F)

    def then(self, other: Transformation) -> Transformation:
        assert self.cod == other.dom
        return Transformation(lambda x: other(self(x)), self.dom, other.cod)

    def tensor(self, other: Transformation | Functor) -> Transformation:
        if isinstance(other, Transformation): return self @ other.dom >> self.cod @ other
        return Transformation(
            lambda x: other(self(x)), self.dom >> other, self.cod >> other)

    def __rmatmul__(self, other: Functor) -> Transformation:
        return Transformation(
            lambda x: self(other(x)), other >> self.dom, other >> self.cod)

    def __call__(self, other: Ty):
        inside, dom, cod = self.inside(other), self.dom(other), self.cod(other)
        return inside if isinstance(inside, self.cod.ar)\
            else self.cod.ar(inside, dom, cod)

Cat = TwoCategory(colours=Category, ob=Functor, ar=Transformation)
```

Example 1.5.15. *We can interpret colours as categories, types as functors and diagrams as transformations.*

```
a = Colour('a')
x = Ty('x', dom=a, cod=a)
f, g = Box('f', Ty.id(a), x), Box('g', x @ x, x)

Pyth = Category(list[type], Function)
List = Functor(
    ob=lambda xs: [list[xs]], ar=lambda f: lambda xs: map(f, xs), dom=Pyth, cod=Pyth)
Unit = Transformation(lambda _: lambda x: [x], dom=Functor.id(Pyth), cod=List)
Mult = Transformation(lambda _: lambda xs: sum(xs, []), dom=List >> List, cod=List)

F = TwoFunctor(colours={a: Pyth}, ob={x: List}, ar: {f: Unit, g: Mult}, cod=Cat)
assert F(f)(int)(42) == [42] and F(g)(int)([[1], [2], [3]]) == [1, 2, 3]
assert F(f @ x >> g)(int)([1, 2, 3]) == [1, 2, 3]
```

We have already discussed another way to construct a 2-categories: taking types as 0-cells, diagrams as 1-cells and rewrites as 2-cells, in fact this gives a *monoidal 2-category*. A monoidal 2-signature Σ is a 2-signature where the objects in Σ_1 have lists of colours $\Sigma_0^\star$ as domain and codomain and the boxes in Σ_2 have domain and codomain in the free premonoidal category $F^P(\Sigma_0, \Sigma_1)$. Thus, a box $r : f \to g$ in a monoidal 2-signature may be seen as a rewrite rule with parallel diagrams $f : x \to y$ and $g : x \to y$ as domain and codomain. It generates a free monoidal 2-category $F^{M2C}(\Sigma)$ with types $\Sigma_0^\star$ as 0-cells, diagrams $F^P(\Sigma_0, \Sigma_1)$ as 1-cells and rewrites as 2-cells. We can construct $F^{M2C}(\Sigma)$ explicitly by generalising layers to *slices* with not only types on the left and right but also diagrams on the top and bottom, i.e. a rewrite rule together with a match. Monoidal 2-categories have too much data and axioms for us to list here, we will only outline their implementation. `Rule` is a subclass of `Box` with diagrams as domain and codomain, `Slice` is a box made of a rule inside a match with methods for left and right whiskering as well as pre- and post-composition. `Rewrite` is a subclass of `Diagram` with `Slice` as layers, it inherits its vertical composition (i.e. two rewrites applied in sequence) from the diagram class as well as its tensor product (i.e. two rewrites applied in parallel on the tensor of two diagrams). The horizontal composition (i.e. two rewrites applied in parallel on the composition of two diagrams) is given by an optional argument to `then` which temporarily replaces left and right whiskering by pre- and post-composition before calling `Diagram.tensor`.

Listing 1.5.16. Outline of the implementation of the free monoidal 2-category.

```
class Rule(Box):
    def __init__(self, name: str, dom: Diagram, cod: diagram):
        super().__init__(name, dom, cod)

class Slice(Box):
    def __init__(self, rule: Rule, match: Match):
        dom, cod = match.subs(rule.dom), matchs.subs(rule.cod)
        super().__init__("Slice({}, {})".format(rule, match), dom, cod)

    __matmul__ = lambda self, other: Slice(self.rule, self.match @ other)
    __rmatmul__ = lambda self, other: Slice(self.rule, other @ self.match)
    __lshift__ = __rrshift__ = lambda self, other: Slice(self.rule, self.match >> other)
    __rshift__ = __rlshift__ = lambda self, other: Slice(self.rule, other >> self.match)

Match.__matmul__ = lambda self, other: Match(
    self.top @ other.dom, self.bottom @ other, self.left, self.right @ other.dom)
Match.__rmatmul__ = lambda self, other: Match(
```

```
    other @ self.top, other.cod @ self.bottom, other.cod @ self.left, self.right)
Match.__lshift__ = Match.__rrshift__ = lambda self, other: Match(
    self.top, self.bottom >> other, self.left, self.right)
Match.__rshift__ = Match.__rlshift__ = lambda self, other: Match(
    other >> self.top, self.bottom, self.left, self.right)

class Rewrite(Diagram):
    def __init__(self, slices: list[Slice], dom: Ty | Diagram, cod: Ty | Diagram):
        dom, cod = (x if isinstance(x, Diagram) else Diagram.id(x) for x in (dom, cod))
        super().__init__(slices, dom, cod)

    def then(self, other, horizontal=False):
        if not horizontal: return super().then(other)
        tmp = Slice.__matmul__, Slice.__rmatmul__
        Slice.__matmul__, Slice.__rmatmul__ = Slice.__lshift__, Slice.__rshift__
        result = super().tensor(other)
        Slice.__matmul__, Slice.__rmatmul__ = tmp
        return result
```

Note that when the monoidal 2-signature Σ is in fact a simple 2-signature, i.e. every box $f \in \Sigma_1$ has domain and codomain of length one, the definition of a rewrite coincides with the definition of coloured diagram. Indeed we can relabel everything one level down: the types are colours, the boxes are types and the rules are boxes: a coloured diagram can be seen as a rewrite of one dimensional diagrams, i.e. lists of types with composable colours. Symmetrically, the rewriting of an n-dimensional diagram can itself be seen as an $n+1$ diagram. If we compose the two constructions together, we get a monoidal 3-category of coloured diagram rewriting, which in turn can be seen as a 4-category with a single 0-cell. What could be the use of a four-dimensional diagram? For example, the free 4-category with a single 0-, 1- and 2-cell is the same as the free symmetric category (once we relabel everything three levels down). Indeed, the swaps are given by the interchange law and the 4d space in which the diagrams live allows wires to cross and every knot to be untied: every diagram interpreted in **Pyth** or $\mathbf{Mat}_{\mathbb{S}}$ is secretly four-dimensional. The proof assistant Globular [BKV18] allows to construct such 4d diagrams using a graphical interface for drawing slices and projections in two dimensions. In fact, the drawing algorithm presented in section 1.3 was also reverse engineered from that of Globular.

Its successor homotopy.io [RV19] went from four to arbitrary dimensions based on a data structure for diagrams in free n-sesquicategories [BV17]. Interfacing DisCoPy with homotopy.io is in the backlog of features yet to be implemented, so that the user can define diagrams by drag-and-dropping boxes then interpret

them in arbitrary n-categories. One of the new feature of homotopy.io compared to its predecessor is the possibility of drawing non-generic diagrams, i.e. with more than one box on the same layer. This amounts to taking the free category over $L^+(\Sigma) = (\Sigma_0 + \Sigma_1)^\star \simeq \Sigma_0^\star \square \Sigma_1^\star$ rather than $L(\Sigma) = \Sigma_0^\star \times \Sigma_1 \times \Sigma_0^\star$. This is also in DisCoPy's backlog, implementing the syntax is straightforward but then it requires to extend the algorithms for functors, drawing, normal forms, etc. Layers with arbitrarily many boxes also allow to define the *depth* of an arrow in any quotient of a free premonoidal category as the minimum number of layers in its equivalence class of diagrams. As we mentioned in section 1.2.4, premonoidal diagrams also have a well-defined notion of *width* (the maximum number of parallel wires) which we can extend in the same way to define the width of any quotient. This makes diagrams a foundational data structure for computational complexity theory: a signature can be seen as both a machine and a language, a diagram as both code and data. In the other direction, this also allows to borrow results from complexity theory to characterise the computational resources required in solving problems about diagrams. This will be needed in the next chapter when we will look at NLP problems through the lens of diagrams.

1.6 Summary & future work

This chapter gave a comprehensive overview of DisCoPy and the mathematics behind its design principles: we take the definitions of category theory (as strictly and freely as possible) and translate them into a Pythonic syntax. Figure 1.3 summarises the different modules and their inheritance hierarchy, implementing a subset of the hierarchy of graphical languages surveyed by Selinger [Sel10]. We hope it may be useful both as an introduction to monoidal categories for the Python programmer, and an introduction to Python programming for the applied category theorist.

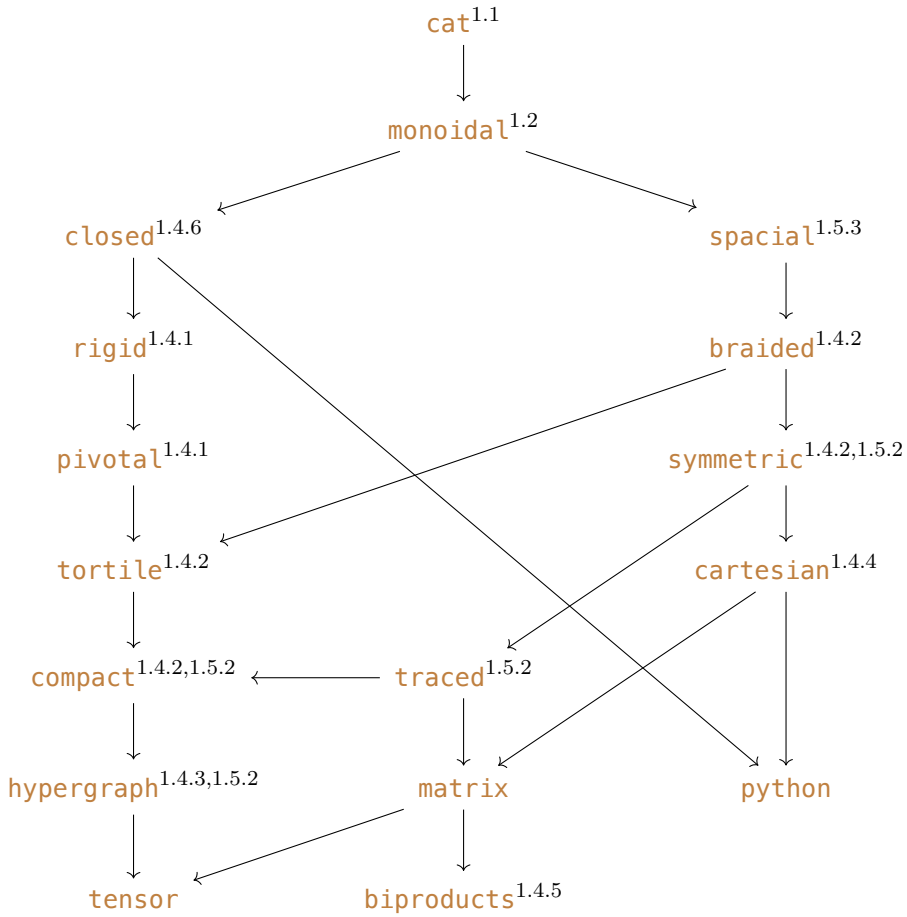


Figure 1.3: DisCoPy's modules and the sections where they are discussed.

We list but a few of the many potential directions for further developments.

- DisCoPy was implemented only with correctness in mind, thus there is much room for improving performance. For now, this has not been quite necessary since the diagrams we manipulate are exponentially smaller than the computation they represent. If we want to implement any serious rewriting efficiently, we will need to port the core algorithms to a lower-level language such as Rust [KN19] and wrap them with Python bindings. This strategy has improved the time performance of PyZX by over four thousand on a small benchmark consisting of the fusion of one million spiders[1].

[1] `https://github.com/quantomatic/quizx`

2

Quantum natural language processing

References

[Aar15] Scott Aaronson. "Read the Fine Print". In: *Nature Physics* 11.4 (4 Apr. 2015), pp. 291–293. DOI: 10.1038/nphys3272.

[Abb+21] Mina Abbaszade, Vahid Salari, Seyed Shahin Mousavi, Mariam Zomorodi, and Xujuan Zhou. "Application of Quantum Natural Language Processing for Language Translation". In: *IEEE Access* 9 (2021), pp. 130434–130448. DOI: 10.1109/ACCESS.2021.3108768.

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

[AC04] Samson Abramsky and Bob Coecke. "A Categorical Semantics of Quantum Protocols". In: *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 2004, pp. 415–425. DOI: 10.1109/LICS.2004.1319636.

[AC08] Samson Abramsky and Bob Coecke. *Categorical Quantum Mechanics*. Aug. 7, 2008. arXiv: 0808.1023 [quant-ph].

[AT10] Samson Abramsky and Nikos Tzevelekos. *Introduction to Categories and Categorical Logic*. 2010.

[AB08] Dorit Aharonov and Michael Ben-Or. "Fault-Tolerant Quantum Computation with Constant Error Rate". In: *SIAM Journal on Computing* 38.4 (Jan. 1, 2008), pp. 1207–1282. DOI: 10.1137/S0097539799359385.

[Ajd35] Kazimierz Ajdukiewicz. "Die Syntaktische Konnexität". In: *Studia Philosophica* (1935).

[Amb04] A. Ambainis. "Quantum Search Algorithms". In: *ACM SIGACT News* 35.2 (June 1, 2004), pp. 22–35. DOI: 10.1145/992287.992296.

[Ari66] Aristotle. *Categories, and De Interpretatione*. In collab. with Internet Archive. Trans. by J. L. Ackrill. Oxford [England] : Clarendon Press ; New York : Oxford University Press, 1966. 178 pp. URL: http://archive.org/details/categoriesdeinte00aris_0 (visited on 12/07/2021).

[Aru+15] Srinivasan Arunachalam, Vlad Gheorghiu, Tomas Jochym-O'Connor, Michele Mosca, and Priyaa Varshinee Srinivasan. "On the Robustness of Bucket Brigade Quantum RAM". In: *New Journal of Physics* 17.12 (Dec. 7, 2015), p. 123010. DOI: 10.1088/1367-2630/17/12/123010.

[Aru+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. "Quantum Supremacy Using a Programmable Superconducting Processor". In: *Nature* 574.7779 (7779 Oct. 2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.

[Ati88] Michael F Atiyah. "Topological Quantum Field Theory". In: *Publications Mathématiques de l'IHÉS* 68 (1988), pp. 175–186.

[Bac+20] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. *There and Back Again: A Circuit Extraction Tale*. Mar. 4, 2020. arXiv: 2003.01664 [quant-ph].

[Bae06] John Baez. "Quantum Quandaries: A Category-Theoretic Perspective". In: *The Structural Foundations of Quantum Gravity*. Oxford: Oxford University Press, 2006. DOI: 10.1093/acprof:oso/9780199269693.003.0008.

[BE14] John C. Baez and Jason Erbele. *Categories in Control*. May 27, 2014. arXiv: 1405.6881 [quant-ph].

[BF15] John C. Baez and Brendan Fong. "A Compositional Framework for Passive Linear Networks". In: (2015). eprint: arXiv:1504.05625.

[BP17] John C. Baez and Blake S. Pollard. "A Compositional Framework for Reaction Networks". In: *Reviews in Mathematical Physics* 29.09 (Oct. 2017), p. 1750028. DOI: 10.1142/S0129055X17500283.

[BS09] John C. Baez and Mike Stay. *Physics, Topology, Logic and Computation: A Rosetta Stone*. Mar. 2, 2009. arXiv: 0903.0340 [quant-ph].

[BD95] John C Baez and James Dolan. "Higher-Dimensional Algebra and Topological Quantum Field Theory". In: *Journal of mathematical physics* 36.11 (1995), pp. 6073–6105.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. May 19, 2016. arXiv: 1409.0473 [cs, stat].

[Bal] Philip Ball. *Major Quantum Computing Strategy Suffers Serious Setbacks*. Quanta Magazine. URL: https://www.quantamagazine.org/major-quantum-computing-strategy-suffers-serious-setbacks-20210929/ (visited on 10/26/2021).

[Ban+08] Jeongho Bang, James Lim, M. S. Kim, and Jinhyoung Lee. *Quantum Learning Machine*. Mar. 31, 2008. arXiv: 0803.2976 [quant-ph].

[Bar53] Yehoshua Bar-Hillel. "A Quasi-Arithmetical Notation for Syntactic Description". In: *Language* 29.1 (Jan. 1953), p. 47. DOI: 10.2307/410452.

[Bar54] Yehoshua Bar-Hillel. "Logical Syntax and Semantics". In: *Language* 30.2 (Apr. 1954), p. 230. DOI: 10.2307/410265.

[BKV18] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. "Globular: An Online Proof Assistant for Higher-Dimensional Rewriting". In: *Logical Methods in Computer Science* 14.1 (2018). DOI: 10.23638/LMCS-14(1:8)2018.

[BV17] Krzysztof Bar and Jamie Vicary. "Data Structures for Quasistrict Higher Categories". In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). June 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005147.

[BLS19] Marcello Benedetti, Erika Lloyd, and Stefan Sack. *Parameterized Quantum Circuits as Machine Learning Models*. June 18, 2019. arXiv: 1906.07682 [quant-ph].

[Ben80] Paul Benioff. "The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines". In: *Journal of Statistical Physics* 22.5 (May 1, 1980), pp. 563–591. DOI: 10.1007/BF01011339.

[Ben+97] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. "Strengths and Weaknesses of Quantum Computing". In: *SIAM Journal on Computing* 26.5 (Oct. 1, 1997), pp. 1510–1523. DOI: 10.1137/S0097539796300933.

[Ben70] David B. Benson. "Syntax and Semantics: A Categorical View". In: *Information and Control* 17.2 (Sept. 1970), pp. 145–160. DOI: 10.1016/S0019-9958(70)90517-6.

[BH03] Nick Benton and Martin Hyland. "Traced Premonoidal Categories". In: *RAIRO - Theoretical Informatics and Applications* 37.4 (Oct. 2003), pp. 273–299. DOI: 10.1051/ita:2003020.

[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*. Feb. 13, 2020. arXiv: 1811.04968 [physics, physics:quant-ph].

[BBD09] Daniel J Bernstein, Johannes Buchmann, and Erik Dahmén. *Post-Quantum Cryptography*. Berlin: Springer, 2009.

[BKL13] William Blacoe, Elham Kashefi, and Mirella Lapata. "A Quantum-Theoretic Approach to Distributional Semantics". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL-HLT 2013. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 847–857. URL: https://aclanthology.org/N13-1105 (visited on 12/01/2021).

[Bol+17] Joe Bolt, Bob Coecke, Fabrizio Genovese, Martha Lewis, Dan Marsden, and Robin Piedeleu. "Interacting Conceptual Spaces I : Grammatical Composition of Concepts". In: *CoRR* abs/1703.08314 (2017). arXiv: 1703.08314.

[Bon+16] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. "Rewriting modulo Symmetric Monoidal Structure". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '16* (2016), pp. 710–719. DOI: 10.1145/2933575.2935316.

[Bon+20] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. *String Diagram Rewrite Theory I: Rewriting with Frobenius Structure*. Dec. 3, 2020. arXiv: 2012.01847 [cs, math].

[BSS18] Filippo Bonchi, Jens Seeber, and Pawel Sobocinski. *Graphical Conjunctive Queries*. Apr. 20, 2018. arXiv: 1804.07626 [cs].

[BSZ14] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. "A Categorical Semantics of Signal Flow Graphs". In: *CONCUR 2014 – Concurrency Theory*. Ed. by Paolo Baldan and Daniele Gorla. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 435–450. DOI: 10.1007/978-3-662-44584-6_30.

[Boo54] George Boole. *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*. In collab. with University of California Libraries. London : Walton and Maberly, 1854. 450 pp. URL: http://archive.org/details/investigationofl00boolrich (visited on 09/07/2019).

[Bor+19] Emanuela Boros, Alexis Toumi, Erwan Rouchet, Bastien Abadie, Dominique Stutzmann, and Christopher Kermorvant. "Automatic Page Classification in a Large Collection of Manuscripts Based on the International Image Interoperability Framework". In: *International Conference on Document Analysis and Recognition*. 2019. DOI: 10.1109/ICDAR.2019.00126.

[Bra+20] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. "JAX: Composable Transformations of Python+ NumPy Programs, 2018". In: *URL http://github. com/google/jax* 4 (2020), p. 16.

[BT00] Geraldine Brady and Todd Trimble. "A Categorical Interpretation of C.S. Peirce's Propositional Logic Alpha". In: *Journal of Pure and Applied Algebra* 149 (June 2000), pp. 213–239. DOI: 10.1016/S0022-4049(98)00179-0.

[BT98] Geraldine Brady and Todd H. Trimble. "A String Diagram Calculus for Prediate Logic and C. S. Peirce's System Beta". 1998. URL: http://people.cs.uchicago.edu/~brady/beta98.ps.

[Bra+02] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. *Quantum Amplitude Amplification and Estimation*. 2002.

[21] "Cambridge Quantum Releases World's First Quantum Natural Language Processing Toolkit and Library". In: *HPC wire* (Oct. 13, 2021). URL: https://www.hpcwire.com/off-the-wire/cambridge-quantum-releases-worlds-first-quantum-natural-language-processing-toolkit/ (visited on 02/24/2022).

[Can06] R. Ferrer i Cancho. "Why Do Syntactic Links Not Cross?" In: *Europhysics Letters (EPL)* 76.6 (Dec. 2006), pp. 1228–1235. DOI: 10.1209/epl/i2006-10406-0.

[Car37] Rudolf Carnap. *Logical Syntax of Language*. London: Kegan Paul and Co., Ltd, 1937.

[Car47] Rudolf Carnap. *Meaning and Necessity: A Study in Semantics and Modal Logic*. University of Chicago Press, 1947.

[Cha+18] Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren, and Dominic Horsman. *Graphical Structures for Design and Verification of Quantum Error Correction*. Jan. 12, 2018. arXiv: 1611.08012 [quant-ph].

[Che02] Joseph CH Chen. "Quantum Computation and Natural Language Processing". Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky, 2002.

[CJ19] Kenta Cho and Bart Jacobs. "Disintegration and Bayesian Inversion via String Diagrams". In: *Mathematical Structures in Computer Science* 29.7 (Aug. 2019), pp. 938–971. DOI: 10.1017/S0960129518000488.

[Cho56] Noam Chomsky. "Three Models for the Description of Language". In: *IRE Transactions on Information Theory* 2.3 (Sept. 1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

[Cho57] Noam Chomsky. *Syntactic Structures*. The Hague: Mouton and Co., 1957.

[CGK98] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. "Experimental Implementation of Fast Quantum Searching". In: *Physical Review Letters* 80.15 (Apr. 13, 1998), pp. 3408–3411. DOI: 10.1103/PhysRevLett.80.3408.

[CP07a] Cindy Chung and James Pennebaker. "The Psychological Functions of Function Words". In: *Social communication* (Jan. 1, 2007).

[CJS13] B. D. Clader, B. C. Jacobs, and C. R. Sprouse. "Preconditioned Quantum Linear System Algorithm". In: *Physical Review Letters* 110.25 (June 18, 2013), p. 250504. DOI: 10.1103/PhysRevLett.110.250504.

[CCS08] Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. "A Compositional Distributional Model of Meaning". In: *Proceedings of the Second Symposium on Quantum Interaction (QI-2008)*. 2008, pp. 133–140.

[CCS10] Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. "Mathematical Foundations for a Compositional Distributional Model of Meaning". In: *A Festschrift for Jim Lambek*. Ed. by J. van Benthem, M. Moortgat, and W. Buszkowski. Vol. 36. Linguistic Analysis. 2010, pp. 345–384. arXiv: 1003.4394.

[CP07b] Stephen Clark and Stephen Pulman. "Combining Symbolic and Distributional Models of Meaning". In: *Quantum Interaction, Papers from the 2007 AAAI Spring Symposium, Technical Report SS-07-08, Stanford, California, USA, March 26-28, 2007*. AAAI, 2007, pp. 52–55. URL: http://www.aaai.org/Library/Symposia/Spring/2007/ss07-08-008.php.

[CGS13] B. Coecke, E. Grefenstette, and M. Sadrzadeh. "Lambek vs. Lambek: Functorial Vector Space Semantics and String Diagrams for Lambek Calculus". In: *ArXiv e-prints* (2013). arXiv: 1302.0393.

[Coe13] Bob Coecke. *An Alternative Gospel of Structure: Order, Composition, Processes*. July 15, 2013. arXiv: 1307.4038 [quant-ph].

[Coe19] Bob Coecke. *The Mathematics of Text Structure*. Apr. 6, 2019. arXiv: 1904.03478.

[CD08] Bob Coecke and Ross Duncan. "Interacting Quantum Observables". In: *Automata, Languages and Programming*. Ed. by Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 298–310. DOI: 10.1016/0022-4049(80)90101-2.

[CD11] Bob Coecke and Ross Duncan. "Interacting Quantum Observables: Categorical Algebra and Diagrammatics". In: *New Journal of Physics* 13 (2011), p. 043016. arXiv: quant-ph/09064725.

[Coe+20a] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "Foundations for Near-Term Quantum Natural Language Processing". In: *CoRR* abs/2012.03755 (2020). arXiv: 2012.03755.

[Coe+20b] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. *Quantum Natural Language Processing*. Apr. 7, 2020. URL: https://medium.com/cambridge-quantum-computing/quantum-natural-language-processing-748d6f27b31d (visited on 02/24/2022).

[Coe+21] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "How to Make Qubits Speak". In: *CoRR* abs/2107.06776 (2021). arXiv: 2107.06776.

[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.

[CS12] Bob Coecke and Robert W. Spekkens. "Picturing Classical and Quantum Bayesian Inference". In: *Synthese* 186.3 (June 2012), pp. 651–696. DOI: 10.1007/s11229-011-9917-5.

[CDH20] Cole Comfort, Antonin Delpeuch, and Jules Hedges. *Sheet Diagrams for Bimonoidal Categories*. Dec. 19, 2020. arXiv: 2010.13361 [math].

[CSD20] Alexander Cowtan, Will Simmons, and Ross Duncan. *A Generic Compilation Strategy for the Unitary Coupled Cluster Ansatz*. Aug. 27, 2020. arXiv: 2007.10515 [quant-ph].

[20a] "CQC Researchers Make Major Quantum NLP Advance in Steps Toward 'Meaning Aware' Computers". In: *The Quantum Insider* (Dec. 10, 2020). URL: https://thequantuminsider.com/2020/12/10/meaning-aware-computers-cqc-researchers-make-major-nlp-advance-in-using-quantum-computers-to-understand-language-and-towards-achieving-meaningful-quantum-advantage/ (visited on 02/24/2022).

[Cro18] Andrew Cross. "The IBM Q Experience and QISKit Open-Source Quantum Computing Software". In: 2018 (Jan. 1, 2018), p. L58.003. URL: https://ui.adsabs.harvard.edu/abs/2018APS..MARL58003C (visited on 01/11/2022).

[dBBW20] Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. *Fast and Effective Techniques for T-count Reduction via Spider Nest Identities*. Apr. 14, 2020. arXiv: 2004.05164 [quant-ph].

[dGD20] Arianne Meijer-van de Griend and Ross Duncan. *Architecture-Aware Synthesis of Phase Polynomials for NISQ Devices*. Apr. 13, 2020. arXiv: 2004.06052 [quant-ph].

[Del20a] Antonin Delpeuch. "A Complete Language for Faceted Dataflow Programs". In: *Electronic Proceedings in Theoretical Computer Science* 323 (Sept. 15, 2020), pp. 1–14. DOI: 10.4204/EPTCS.323.1.

[Del20b] Antonin Delpeuch. *The Word Problem for Double Categories*. Jan. 2, 2020. arXiv: 1907.09927 [cs, math].

[DV18] Antonin Delpeuch and Jamie Vicary. *Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm*. Apr. 20, 2018. arXiv: 1804.07832 [cs].

[DV21] Antonin Delpeuch and Jamie Vicary. *The Word Problem for Braided Monoidal Categories Is Unknot-Hard*. May 10, 2021. arXiv: 2105.04237 [math].

[Deu85] David Deutsch. "Quantum Theory, the Church–Turing Principle and the Universal Quantum Computer". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818 (1985), pp. 97–117.

[DJ92] David Deutsch and Richard Jozsa. "Rapid Solution of Problems by Quantum Computation". In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (Dec. 8, 1992), pp. 553–558. DOI: 10.1098/rspa.1992.0167.

[Dev+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. May 24, 2019. arXiv: 1810.04805 [cs].

[Dun+20] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. "Graph-Theoretic Simplification of Quantum Circuits with the ZX-calculus". In: *Quantum* 4 (June 4, 2020), p. 279. DOI: 10.22331/q-2020-06-04-279.

[DV19] Lawrence Dunn and Jamie Vicary. *Coherence for Frobenius Pseudomonoids and the Geometry of Linear Proofs*. 2019.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. "Graph-Grammars: An Algebraic Approach". In: *14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. 14th Annual Symposium on Switching and Automata Theory (Swat 1973). Oct. 1973, pp. 167–180. DOI: 10.1109/SWAT.1973.11.

[EM42a] Samuel Eilenberg and Saunders MacLane. "Group Extensions and Homology". In: *Annals of Mathematics* 43.4 (1942), pp. 757–831. DOI: 10.2307/1968966.

[EM42b] Samuel Eilenberg and Saunders MacLane. “Natural Isomorphisms in Group Theory”. In: *Proceedings of the National Academy of Sciences of the United States of America* 28.12 (1942), p. 537.

[EM45] Samuel Eilenberg and Saunders MacLane. “General Theory of Natural Equivalences”. In: *Transactions of the American Mathematical Society* 58 (1945), pp. 231–294. DOI: 10.1090/S0002-9947-1945-0013131-6.

[FGG14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. Nov. 14, 2014. arXiv: 1411.4028 [quant-ph].

[FN18] Edward Farhi and Hartmut Neven. *Classification with Quantum Neural Networks on Near Term Processors*. Version 2. Aug. 30, 2018. arXiv: 1802.06002 [quant-ph].

[Fel+20] Giovanni de Felice, Elena Di Lavore, Mario Román, and Alexis Toumi. “Functorial Language Games for Question Answering”. In: *Proceedings of the 3rd Annual International Applied Category Theory Conference 2020, ACT 2020, Cambridge, USA, 6-10th July 2020*. Ed. by David I. Spivak and Jamie Vicary. Vol. 333. EPTCS. 2020, pp. 311–321. DOI: 10.4204/EPTCS.333.21.

[FMT19] Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. “Functorial Question Answering”. In: *Proceedings Applied Category Theory 2019, ACT 2019, University of Oxford, UK*. Vol. 323. EPTCS. 2019. DOI: 10.4204/EPTCS.323.6.

[FTC20] Giovanni de Felice, Alexis Toumi, and Bob Coecke. “DisCoPy: Monoidal Categories in Python”. In: *Proceedings of the 3rd Annual International Applied Category Theory Conference, ACT*. Vol. 333. EPTCS, 2020. DOI: 10.4204/EPTCS.333.13.

[Fey85] Richard P Feynman. “Quantum Mechanical Computers”. In: *Optics news* 11.2 (1985), pp. 11–20.

[Fey82] Richard P. Feynman. “Simulating Physics with Computers”. In: *International Journal of Theoretical Physics* 21.6 (June 1, 1982), pp. 467–488. DOI: 10.1007/BF02650179.

[Fir57] John R Firth. “A Synopsis of Linguistic Theory, 1930-1955”. In: *Studies in linguistic analysis* (1957).

[FLK80] François Foltz, Christian Lair, and GM Kelly. “Algebraic Categories with Few Monoidal Biclosed Structures or None”. In: *Journal of Pure and Applied Algebra* 17.2 (1980), pp. 171–177.

[FST17] Brendan Fong, David I. Spivak, and Rémy Tuyéras. "Backprop as Functor: A Compositional Perspective on Supervised Learning". In: (2017). eprint: arXiv:1711.10455.

[Fre+03] Michael Freedman, Alexei Kitaev, Michael Larsen, and Zhenghan Wang. "Topological Quantum Computation". In: *Bulletin of the American Mathematical Society* 40.1 (2003), pp. 31–38.

[Fre84] Gottlob Frege. "The Foundations of Arithmetic". In: *JL Austin. New York: Philosophical Library* (1884).

[FMG15] Richard Futrell, Kyle Mahowald, and Edward Gibson. "Large-Scale Evidence of Dependency Length Minimization in 37 Languages". In: *Proceedings of the National Academy of Sciences* 112.33 (Aug. 18, 2015), pp. 10336–10341. DOI: 10.1073/pnas.1502134112.

[Fut21] Futurati Podcast. *Ep. 52: Bob Coecke and Konstantinos Meichanetzidis on Quantum Natural Language Processing*. Sept. 21, 2021. URL: https://www.youtube.com/watch?v=5YZG96t8SLQ (visited on 02/24/2022).

[Gha+18] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. *Compositional Game Theory*. Feb. 15, 2018. arXiv: 1603.04641 [cs].

[GF19] Craig Gidney and Austin G. Fowler. *Flexible Layout of Surface Code Computations Using AutoCCZ States*. May 21, 2019. arXiv: 1905.08916 [quant-ph].

[GLM08] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum Random Access Memory". In: *Physical Review Letters* 100.16 (Apr. 21, 2008), p. 160501. DOI: 10.1103/PhysRevLett.100.160501.

[GPT20] GPT-3. "A Robot Wrote This Entire Article. Are You Scared yet, Human?" In: *The Guardian. Opinion* (Sept. 8, 2020). URL: https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3 (visited on 11/19/2021).

[Gra+19] Edward Grant, Leonard Wossnig, Mateusz Ostaszewski, and Marcello Benedetti. "An Initialization Strategy for Addressing Barren Plateaus in Parametrized Quantum Circuits". In: *Quantum* 3 (Dec. 9, 2019), p. 214. DOI: 10.22331/q-2019-12-09-214.

[Gra44] Hermann Grassmann. *Die Lineale Ausdehnungslehre Ein Neuer Zweig Der Mathematik: Dargestellt Und Durch Anwendungen Auf Die Übrigen Zweige Der Mathematik, Wie Auch Auf Die Statik, Mechanik, Die Lehre Vom Magnetismus Und Die Krystallonomie Erläutert*. Vol. 1. O. Wigand, 1844.

[GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. Ieee. 2013, pp. 6645–6649.

[GS11] Edward Grefenstette and Mehrnoosh Sadrzadeh. *Experimental Support for a Categorical Compositional Distributional Model of Meaning*. 2011. arXiv: 1106.4058 [cs, math].

[Gre+10] Edward Grefenstette, Mehrnoosh Sadrzadeh, Stephen Clark, Bob Coecke, and Stephen Pulman. *Concrete Sentence Spaces for Compositional Distributional Models of Meaning*. Dec. 31, 2010. arXiv: 1101.0309 [cs].

[GD60] Alexandre Grothendieck and Jean Dieudonné. "Eléments de Géométrie Algébrique". In: *Publications Mathématiques de l'Institut des Hautes Études Scientifiques* 4.1 (1960), pp. 5–214.

[Gro97] Lov K. Grover. "Quantum Mechanics Helps in Searching for a Needle in a Haystack". In: *Physical Review Letters* 79.2 (July 14, 1997), pp. 325–328. DOI: 10.1103/PhysRevLett.79.325.

[HNW18] Amar Hadzihasanovic, Kang Feng Ng, and Quanlong Wang. "Two Complete Axiomatisations of Pure-state Qubit Quantum Computing". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom). LICS '18. New York, NY, USA: ACM, 2018, pp. 502–511. DOI: 10.1145/3209108.3209128.

[HSS08] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring Network Structure, Dynamics, and Function Using Networkx*. LA-UR-08-05495; LA-UR-08-5495. Los Alamos National Lab. (LANL), Los Alamos, NM (United States), Jan. 1, 2008. URL: https://www.osti.gov/biblio/960616 (visited on 01/27/2022).

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

[Har54] Zellig S. Harris. "Distributional Structure". In: *WORD* 10.2-3 (Aug. 1, 1954), pp. 146–162. DOI: 10.1080/00437956.1954.11659520.

[HHL09] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum Algorithm for Linear Systems of Equations". In: *Physical Review Letters* 103.15 (Oct. 7, 2009), p. 150502. DOI: 10.1103/PhysRevLett.103.150502.

[Hau89] John Haugeland. *Artificial Intelligence: The Very Idea*. MIT press, 1989.

[Hav+19] Vojtech Havlicek, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. "Supervised Learning with Quantum Enhanced Feature Spaces". In: *Nature* 567.7747 (Mar. 2019), pp. 209–212. DOI: 10.1038/s41586-019-0980-2.

[HS20] Nathan Haydon and Pawel Sobocinski. "Compositional Diagrammatic First-Order Logic". In: (2020), p. 16.

[Heb49] Donald Olding Hebb. *The Organisation of Behaviour: A Neuropsychological Theory*. Science Editions New York, 1949.

[HL18] Jules Hedges and Martha Lewis. *Towards Functorial Language-Games*. July 20, 2018. arXiv: 1807.07828 [cs].

[Heg12] Georg Wilhelm Friedrich Hegel. *Wissenschaft Der Logik*. F. Frommann, 1812.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Oxford Graduate Texts in Mathematics. Oxford: Oxford University Press, 2019. 336 pp. DOI: 10.1093/oso/9780198739623.001.0001.

[Hil97] Melanie Hilario. "An Overview of Strategies for Neurosymbolic Integration". In: *Connectionist-Symbolic Integration: From Unified to Hybrid Approaches* (1997), pp. 13–36.

[Hoc98] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06.02 (Apr. 1998), pp. 107–116. DOI: 10.1142/S0218488598000094.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1, 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[Hof21] Thomas Hoffmann. "Quantum Models for Word- Sense Disambiguation". In: (2021). URL: https://odr.chalmers.se/handle/20.500.12380/302687 (visited on 12/02/2021).

[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.

[Hot65] Günter Hotz. “Eine Algebraisierung Des Syntheseproblems von Schaltkreisen I”. Trans. by Johannes Drever. In: *Elektronische Informationsverarbeitung und Kybernetik* 1 (1965), pp. 185–205. URL: https://github.com/drever/hotz-translation.

[Hot66] Günter Hotz. “Eindeutigkeit Und Mehrdeutigkeit Formaler Sprachen”. In: *J. Inf. Process. Cybern.* (1966). DOI: 10.5604/16431243.1040101.

[Hun07] John D Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in science & engineering* 9.03 (2007), pp. 90–95.

[Hut04] W John Hutchins. “The Georgetown-Ibm Experiment Demonstrated in January 1954”. In: *Conference of the Association for Machine Translation in the Americas*. Springer. 2004, pp. 102–114.

[JPV18] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. “A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom). LICS ’18. New York, NY, USA: ACM, 2018, pp. 559–568. DOI: 10.1145/3209108.3209131.

[Jef97] Alan Jeffrey. “Premonoidal Categories and a Graphical View of Programs”. In: *Preprint, Dec* (1997).

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

[JS91] André Joyal and Ross Street. “The Geometry of Tensor Calculus, I”. In: *Advances in Mathematics* 88.1 (July 1, 1991), pp. 55–112. DOI: 10.1016/0001-8708(91)90003-P.

[JS95] André Joyal and Ross Street. “The Geometry of Tensor Calculus II”. In: *Unpublished draft, available from Ross Street’s website* 312 (1995), p. 313.

[JSV96] André Joyal, Ross Street, and Dominic Verity. “Traced Monoidal Categories”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 119.3 (Apr. 1996), pp. 447–468. DOI: 10.1017/S0305004100074338.

[Kan81] Immanuel Kant. *Critique of Pure Reason*. Trans. by Norman Kemp Smith. Read Books Ltd. (2011), 1781.

[Kar+21] Dimitri Kartsaklis, Ian Fan, Richie Yeung, Anna Pearson, Robin Lorenz, Alexis Toumi, Giovanni de Felice, Konstantinos Meichanetzidis, Stephen Clark, and Bob Coecke. “Lambeq: An Efficient High-Level Python Library for Quantum NLP”. In: *CoRR* abs/2110.04236 (2021). arXiv: 2110.04236.

[KSP13] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, and Stephen Pulman. "Separating Disambiguation from Composition in Distributional Semantics". In: (2013), p. 10.

[KP16] Iordanis Kerenidis and Anupam Prakash. *Quantum Recommendation Systems*. Mar. 29, 2016. arXiv: 1603.08675 [quant-ph].

[KU19] Aleks Kissinger and Sander Uijlen. *A Categorical Semantics for Causal Structure*. 2019.

[KvdW19] Aleks Kissinger and John van de Wetering. *PyZX: Large Scale Automated Diagrammatic Reasoning*. Apr. 9, 2019. arXiv: 1904.04735 [quant-ph].

[KvdW20] Aleks Kissinger and John van de Wetering. "Reducing T-count with the ZX-calculus". In: *Physical Review A* 102.2 (Aug. 11, 2020), p. 022406. DOI: 10.1103/PhysRevA.102.022406.

[KZ15] Aleks Kissinger and Vladimir Zamdzhiev. "Quantomatic: A Proof Assistant for Diagrammatic Reasoning". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 326–336. arXiv: 1503.01034.

[Kit95] A. Yu Kitaev. *Quantum Measurements and the Abelian Stabilizer Problem*. Nov. 20, 1995. arXiv: quant-ph/9511026.

[Kit03] A. Yu. Kitaev. "Fault-Tolerant Quantum Computation by Anyons". In: *Annals of Physics* 303.1 (Jan. 1, 2003), pp. 2–30. DOI: 10.1016/S0003-4916(02)00018-0.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[Knu68] Donald E. Knuth. "Semantics of Context-Free Languages". In: *In Mathematical Systems Theory*. 1968, pp. 127–145.

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

[LF11] Adam Lally and Paul Fodor. "Natural Language Processing with Prolog in the IBM Watson System". In: *The Association for Logic Programming (ALP) Newsletter* 9 (2011).

[Lam88] J. Lambek. "Categorial and Categorical Grammars". In: *Categorial Grammars and Natural Language Structures*. Ed. by Richard T. Oehrle, Emmon Bach, and Deirdre Wheeler. Studies in Linguistics and Philosophy. Dordrecht: Springer Netherlands, 1988, pp. 297–317. DOI: 10.1007/978-94-015-6878-4_11.

[Lam10] J. Lambek. “Compact Monoidal Categories from Linguistics to Physics”. In: *New Structures for Physics*. Ed. by Bob Coecke. Vol. 813. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 467–487. DOI: 10.1007/978-3-642-12821-9_8.

[Lam58] Joachim Lambek. “The Mathematics of Sentence Structure”. In: *The American Mathematical Monthly* 65.3 (Mar. 1, 1958), pp. 154–170. DOI: 10.1080/00029890.1958.11989160.

[Lam59] Joachim Lambek. “Contributions to a Mathematical Analysis of the English Verb-phrase”. In: *Canadian Journal of Linguistics/Revue canadienne de linguistique* 5.2 (1959), pp. 83–89. DOI: 10.1017/S0008413100018715.

[Lam61] Joachim Lambek. “On the Calculus of Syntactic Types”. In: *Structure of language and its mathematical aspects* 12 (1961), pp. 166–178.

[Lam68] Joachim Lambek. “Deductive Systems and Categories”. In: *Mathematical Systems Theory* 2.4 (1968), pp. 287–318.

[Lam69] Joachim Lambek. “Deductive Systems and Categories II. Standard Constructions and Closed Categories”. In: *Category Theory, Homology Theory and Their Applications I*. Springer, 1969, pp. 76–122.

[Lam72] Joachim Lambek. “Deductive Systems and Categories III. Cartesian Closed Categories, Intuitionist Propositional Calculus, and Combinatory Logic”. In: *Toposes, Algebraic Geometry and Logic*. Springer, 1972, pp. 57–82.

[Lam99a] Joachim Lambek. “Deductive Systems and Categories in Linguistics”. In: *Logic, Language and Reasoning*. Ed. by Hans Jürgen Ohlbach and Uwe Reyle. Red. by Ryszard Wójcicki, Petr Hájek, David Makinson, Daniele Mundici, Krister Segerberg, and Alasdair Urquhart. Vol. 5. Trends in Logic. Dordrecht: Springer Netherlands, 1999, pp. 279–294. DOI: 10.1007/978-94-011-4574-9_12.

[Lam99b] 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.

[Lam01] Joachim Lambek. “Type Grammars as Pregroups”. In: *Grammars* 4 (2001), pp. 21–39. DOI: 10.1023/A:1011444711686.

[Lam08] Joachim Lambek. *From Word to Sentence: A Computational Algebraic Approach to Grammar*. Open Access Publications. Polimetrica, 2008.

[Lau05] Aaron D. Lauda. *Frobenius Algebras and Planar Open String Topological Field Theories*. Aug. 18, 2005. arXiv: `math/0508349`.

[Law64] F William Lawvere. "An Elementary Theory of the Category of Sets". In: *Proceedings of the National academy of Sciences of the United States of America* 52.6 (1964), p. 1506.

[Law70a] F William Lawvere. "Quantifiers and Sheaves". In: *Actes Du Congres International Des Mathematiciens, Nice*. Vol. 1. 1970, pp. 329–334.

[Law79] F William Lawvere. "Categorical Dynamics". In: *Topos theoretic methods in geometry* 30 (1979), pp. 1–28.

[Law89] F William Lawvere. "Display of Graphics and Their Applications, as Exemplified by 2-Categories and the Hegelian "Taco"". In: *Proceedings of the First International Conference on Algebraic Methodology and Software Technology, University of Iowa*. 1989, pp. 51–74.

[LS86] F William Lawvere and Stephen H Schanuel. *Categories in Continuum Physics: Lectures given at a Workshop Held at SUNY, Buffalo 1982*. Lecture Notes in Mathematics 1174. Springer, 1986.

[Law63] F. William Lawvere. "Functorial Semantics of Algebraic Theories". In: *Proceedings of the National Academy of Sciences of the United States of America* 50.5 (1963), pp. 869–872. JSTOR: `71935`.

[Law66] F. William Lawvere. "The Category of Categories as a Foundation for Mathematics". In: *Proceedings of the Conference on Categorical Algebra*. Ed. by S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl. Springer Berlin Heidelberg, 1966, pp. 1–20.

[Law69] F. William Lawvere. "Adjointness in Foundations". In: *Dialectica* 23.34 (1969), pp. 281–296. DOI: `10.1111/j.1746-8361.1969.tb01194.x`.

[Law70b] F. William Lawvere. "Equality in Hyperdoctrines and the Comprehension Schema as an Ad-Joint Functor". In: 1970.

[Law91] F. William Lawvere. "Some Thoughts on the Future of Category Theory". In: *Category Theory*. Ed. by Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini. Vol. 1488. Lecture Notes in Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 1–13. URL: `http://link.springer.com/10.1007/BFb0084208` (visited on 12/15/2021).

[Law92] F. William Lawvere. "Categories of Space and of Quantity". In: *The Space of Mathematics*. Ed. by Javier Echeverria, Andoni Ibarra, and Thomas Mormann. Berlin, Boston: DE GRUYTER, Jan. 31, 1992. DOI: `10.1515/9783110870299.14`.

[Law96] F. William Lawvere. “Unity and Identity of Opposites in Calculus and Physics”. In: *Applied Categorical Structures* 4.2-3 (1996), pp. 167–174. DOI: 10.1007/BF00122250.

[LMR13] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. *Quantum Algorithms for Supervised and Unsupervised Machine Learning*. Nov. 4, 2013. arXiv: 1307.0411 [quant-ph].

[LMR14] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. “Quantum Principal Component Analysis”. In: *Nature Physics* 10.9 (Sept. 2014), pp. 631–633. DOI: 10.1038/nphys3029.

[LB02] Edward Loper and Steven Bird. *NLTK: The Natural Language Toolkit*. May 17, 2002. arXiv: cs/0205028.

[Lor+21] Robin Lorenz, Anna Pearson, Konstantinos Meichanetzidis, Dimitri Kartsaklis, and Bob Coecke. *QNLP in Practice: Running Compositional Models of Meaning on a Quantum Computer*. Feb. 25, 2021. arXiv: 2102.12846 [quant-ph].

[Mac21] Machine Learning Street Talk. *#53 Quantum Natural Language Processing - Prof Bob Coecke*. 2021. URL: https://www.youtube.com/watch?v=X9uSV1YcOy4 (visited on 02/24/2022).

[Mac38] Saunders MacLane. “Carnap on Logical Syntax”. In: *Bulletin of the American Mathematical Society* 44.3 (1938), pp. 171–176.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1971. URL: https://books.google.fr/books?id=eBvhyc4z8HQC.

[Man80] Yuri Manin. “Computable and Uncomputable”. In: *Sovetskoye Radio, Moscow* 128 (1980).

[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.

[Mar47] A Markov. “On Certain Insoluble Problems Concerning Matrices”. In: *Doklady Akad. Nauk SSSR*. Vol. 57. 6. 1947, pp. 539–542.

[McC+18] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, Ryan Babbush, and Hartmut Neven. “Barren Plateaus in Quantum Neural Network Training Landscapes”. In: *Nature Communications* 9.1 (Dec. 2018), p. 4812. DOI: 10.1038/s41467-018-07090-4.

[McC+16] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. "The Theory of Variational Hybrid Quantum-Classical Algorithms". In: *New Journal of Physics* 18.2 (Feb. 4, 2016), p. 023023. DOI: 10.1088/1367-2630/18/2/023023.

[MP43] Warren S McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[McP+21] Lachlan McPheat, Gijs Wijnholds, Mehrnoosh Sadrzadeh, Adriana Correia, and Alexis Toumi. "Anaphora and Ellipsis in Lambek Calculus with a Relevant Modality: Syntax and Semantics". In: *CoRR* abs/2110.10641 (2021). arXiv: 2110.10641.

[Mei+20a] Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni de Felice, Nicolò Chiappori, Alexis Toumi, and Bob Coecke. "Quantum Natural Language Processing on Near-Term Quantum Computers". In: *Proceedings 17th International Conference on Quantum Physics and Logic, QPL 2020, Paris, France, June 2 - 6, 2020*. Ed. by Benoît Valiron, Shane Mansfield, Pablo Arrighi, and Prakash Panangaden. Vol. 340. EPTCS. 2020, pp. 213–229. DOI: 10.4204/EPTCS.340.11.

[Mei+20b] Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. "Grammar-Aware Question-Answering on Quantum Computers". In: *ArXiv e-prints* (2020). arXiv: 2012.03756.

[Mel06] Paul-André Melliès. "Functorial Boxes in String Diagrams". In: *Computer Science Logic*. Ed. by Zoltán Ésik. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 1–30. DOI: 10.1023/A:1024247613677.

[MZ16] Paul-André Melliès and Noam Zeilberger. *A Bifibrational Reconstruction of Lawvere's Presheaf Hyperdoctrine*. Aug. 12, 2016. arXiv: 1601.06098 [cs, math].

[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.

[Mir+21] Eduardo Reck Miranda, Richie Yeung, Anna Pearson, Konstantinos Meichanetzidis, and Bob Coecke. *A Quantum Natural Language Processing Approach to Musical Intelligence*. Nov. 10, 2021. arXiv: 2111.06741 [quant-ph].

[Mog91] Eugenio Moggi. "Notions of Computation and Monads". In: *Information and computation* 93.1 (1991), pp. 55–92.

[Mol21] Paula Garcia Molina. *QNLP Qiskit Hackathon*. Oct. 22, 2021. URL: https://github.com/PaulaGarciaMolina/QNLP_Qiskit_Hackathon (visited on 02/24/2022).

[Mon70a] Richard Montague. "English as a Formal Language". In: *Linguaggi Nella Societa e Nella Tecnica*. Ed. by Bruno Visentini. Edizioni di Communita, 1970, pp. 188–221.

[Mon70b] Richard Montague. "Universal Grammar". In: *Theoria* 36.3 (1970), pp. 373–398. DOI: 10.1111/j.1755-2567.1970.tb00434.x.

[Mon73] Richard Montague. "The Proper Treatment of Quantification in Ordinary English". In: *Approaches to Natural Language* (1973). Ed. by K. J. J. Hintikka, J. Moravcsic, and P. Suppes, pp. 221–242.

[nLa] nLab. *Concept with an Attitude in nLab*. URL: https://ncatlab.org/nlab/show/concept+with+an+attitude (visited on 12/22/2021).

[Pat17] Evan Patterson. *Knowledge Representation in Bicategories of Relations*. June 1, 2017. arXiv: 1706.00526 [cs, math].

[PSV21] Evan Patterson, David I. Spivak, and Dmitry Vagner. *Wiring Diagrams as Normal Forms for Computing in Symmetric Monoidal Categories*. Jan. 25, 2021.

[Ped+19] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. *Leveraging Secondary Storage to Simulate Deep 54-Qubit Sycamore Circuits*. Oct. 22, 2019. arXiv: 1910.09534 [quant-ph].

[Pei06] Charles Santiago Sanders Peirce. "Prolegomena to an Apology of Pragmaticism". In: *The Monist* 16.4 (1906), pp. 492–546. JSTOR: 27899680.

[Pel01] Francis Jeffry Pelletier. "Did Frege Believe Frege's Principle?" In: *Journal of Logic, Language and information* 10.1 (2001), pp. 87–114.

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

[PR84] Roger Penrose and Wolfgang Rindler. *Spinors and Space-Time: Volume 1: Two-Spinor Calculus and Relativistic Fields*. Vol. 1. Cambridge Monographs on Mathematical Physics. Cambridge: Cambridge University Press, 1984. DOI: 10.1017/CBO9780511564048.

[Per+14] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. "A Variational Eigenvalue Solver on a Photonic Quantum Processor". In: *Nature Communications* 5.1 (1 July 23, 2014), p. 4213. DOI: 10.1038/ncomms5213.

[Pes+21] Arthur Pesah, M. Cerezo, Samson Wang, Tyler Volkoff, Andrew T. Sornborger, and Patrick J. Coles. "Absence of Barren Plateaus in Quantum Convolutional Neural Networks". In: *Physical Review X* 11.4 (Oct. 15, 2021), p. 041011. DOI: 10.1103/PhysRevX.11.041011.

[Poi95] Henri Poincaré. *Analysis Situs*. Gauthier-Villars Paris, France, 1895.

[Pos47] Emil L. Post. "Recursive Unsolvability of a Problem of Thue". In: *Journal of Symbolic Logic* 12.1 (Mar. 1947), pp. 1–11. DOI: 10.2307/2267170.

[PR97] John Power and Edmund Robinson. "Premonoidal Categories and Notions of Computation". In: *Mathematical Structures in Computer Science* 7.5 (Oct. 1997), pp. 453–468. DOI: 10.1017/S0960129597002375.

[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.

[Pre18] John Preskill. "Quantum Computing in the NISQ Era and Beyond". In: *Quantum* 2 (Aug. 6, 2018), p. 79. DOI: 10.22331/q-2018-08-06-79.

[20b] *PyData Berlin*. Sept. 21, 2020. URL: https://www.youtube.com/watch?v=5jK8qEQvR-o (visited on 02/24/2022).

[RML14] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. "Quantum Support Vector Machine for Big Data Classification". In: *Physical Review Letters* 113.13 (Sept. 25, 2014), p. 130503. DOI: 10.1103/PhysRevLett.113.130503.

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

[RV19] David Reutter and Jamie Vicary. *High-Level Methods for Homotopy Construction in Associative n-Categories*. Feb. 11, 2019. arXiv: 1902.03831 [math].

[RV16] Emily Riehl and Dominic Verity. *Infinity Category Theory from Scratch*. 2016. arXiv: 1608.05314.

[Ril18] Mitchell Riley. *Categories of Optics*. Sept. 3, 2018. arXiv: 1809.00738 [math].

[Rob+19] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. *TensorNetwork: A Library for Physics and Machine Learning*. May 3, 2019. arXiv: 1905.01330 [cond-mat, physics:hep-th, physics:physics, stat].

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (6088 Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0.

[Rus03] Bertrand Russell. *The Principles of Mathematics*. Routledge, 1903.

[Ryl37] G. Ryle. "Categories". In: *Proceedings of the Aristotelian Society* 38 (1937), pp. 189–206. JSTOR: 4544305.

[SWY75] G. Salton, A. Wong, and C. S. Yang. "A Vector Space Model for Automatic Indexing". In: *Commun. ACM* 18.11 (1975), pp. 613–620. DOI: 10.1145/361219.361220.

[SCn21] Urs Schreiber, David Corfield, and nLab. *Science of Logic*. 2021. URL: https://ncatlab.org/nlab/show/Science+of+Logic (visited on 12/15/2021).

[Sch21] Maria Schuld. *Quantum Machine Learning Models Are Kernel Methods*. Jan. 26, 2021. arXiv: 2101.11020 [quant-ph, stat].

[SK19] Maria Schuld and Nathan Killoran. "Quantum Machine Learning in Feature Hilbert Spaces". In: *Physical Review Letters* 122.4 (Feb. 1, 2019), p. 040504. DOI: 10.1103/PhysRevLett.122.040504.

[SP97] M. Schuster and K.K. Paliwal. "Bidirectional Recurrent Neural Networks". In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997), pp. 2673–2681. DOI: 10.1109/78.650093.

[Sco00] Phill J Scott. "Some Aspects of Categories in Computer Science". In: *Handbook of Algebra*. Vol. 2. Elsevier, 2000, pp. 3–77.

[Sel10] P. Selinger. "A Survey of Graphical Languages for Monoidal Categories". In: *New Structures for Physics* (2010), pp. 289–355. DOI: 10.1007/978-3-642-12821-9_4.

[Sel04] Peter Selinger. "Towards a Quantum Programming Language". In: *Mathematical Structures in Computer Science* 14.4 (2004), pp. 527–586.

[Sel07] Peter Selinger. "Dagger Compact Closed Categories and Completely Positive Maps: (Extended Abstract)". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005) 170 (Mar. 6, 2007), pp. 139–163. DOI: 10.1016/j.entcs.2006.12.018.

[SV06] Peter Selinger and Benoit Valiron. "A Lambda Calculus for Quantum Computation with Classical Control". In: *Mathematical Structures in Computer Science* 16.3 (June 2006), pp. 527–552. DOI: 10.1017/S0960129506005238.

[SV+09] Peter Selinger, Benoıt Valiron, et al. "Quantum Lambda Calculus". In: *Semantic techniques in quantum computation* (2009), pp. 135–172.

[SB09] Dan Shepherd and Michael J. Bremner. "Temporally Unstructured Quantum Computation". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 465.2105 (May 8, 2009), pp. 1413–1439. DOI: 10.1098/rspa.2008.0443.

[STS20] Dan Shiebler, Alexis Toumi, and Mehrnoosh Sadrzadeh. "Incremental Monoidal Grammars". In: *CoRR* abs/2001.02296 (2020). arXiv: 2001.02296.

[Sho94] P.W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 35th Annual Symposium on Foundations of Computer Science. Santa Fe, NM, USA: IEEE Comput. Soc. Press, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[Sho96] Peter W Shor. "Fault-Tolerant Quantum Computation". In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 56–65.

[Sim94] D. Simon. "On the Power of Quantum Computation". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 1994, pp. 116–123. DOI: 10.1109/SFCS.1994.365701.

[Siv+20] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. *Tket : A Retargetable Compiler for NISQ Devices*. Mar. 25, 2020. arXiv: 2003.10611 [quant-ph].

[Smi21] Paul Smith-Goodson. “Cambridge Quantum Makes Quantum Natural Language Processing A Reality”. In: *Forbes* (Oct. 13, 2021). URL: https://www.forbes.com/sites/moorinsights/2021/10/13/cambridge-quantum-makes-quantum-natural-language-processing-a-reality/ (visited on 02/24/2022).

[Smo87] P. Smolensky. “Connectionist AI, Symbolic AI, and the Brain”. In: *Artificial Intelligence Review* 1.2 (1987), pp. 95–109. DOI: 10.1007/BF00130011.

[Smo88] Paul Smolensky. “On the Proper Treatment of Connectionism”. In: *Behavioral and Brain Sciences* 11.1 (Mar. 1988), pp. 1–23. DOI: 10.1017/S0140525X00052432.

[Smo90] Paul Smolensky. “Tensor Product Variable Binding and the Representation of Symbolic Structures in Connectionist Systems”. In: *Artificial Intelligence* 46.1 (Nov. 1, 1990), pp. 159–216. DOI: 10.1016/0004-3702(90)90007-M.

[SWZ19] Paweł Sobociński, Paul W. Wilson, and Fabio Zanasi. “CARTOGRAPHER: A Tool for String Diagrammatic Reasoning”. In: *CALCO 2019*. Vol. 139. 2019, 20:1–20:7. DOI: 10.4230/LIPIcs.CALCO.2019.20.

[Sta79] Richard Statman. “The Typed -Calculus Is Not Elementary Recursive”. In: *Theoretical Computer Science* 9.1 (1979), pp. 73–81.

[SMH11] Ilya Sutskever, James Martens, and Geoffrey E Hinton. “Generating Text with Recurrent Neural Networks”. In: *ICML*. 2011.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html (visited on 11/22/2021).

[Tai67] William W Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *The journal of symbolic logic* 32.2 (1967), pp. 198–212.

[Tan13] Till Tantau. “Graph Drawing in TikZ”. In: *Graph Drawing*. Ed. by Walter Didimo and Maurizio Patrignani. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 517–528. DOI: 10.1007/978-3-642-36763-2_46.

[Thu14] Axel Thue. *Probleme Über Veränderungen von Zeichenreihen Nach Gegebenen Regeln.* na, 1914.

[TK21] Alexis Toumi and Alex Koziell-Pipe. "Functorial Language Models". In: *CoRR* abs/2103.14411 (2021). arXiv: 2103.14411.

[TYF21] Alexis Toumi, Richie Yeung, and Giovanni de Felice. "Diagrammatic Differentiation for Quantum Machine Learning". In: *Proceedings 18th International Conference on Quantum Physics and Logic, QPL 2021, Gdansk, Poland, and Online, 7-11 June 2021*. Ed. by Chris Heunen and Miriam Backens. Vol. 343. EPTCS. 2021, pp. 132–144. DOI: 10.4204/EPTCS.343.7.

[Tur50] A. M. Turing. "Computing Machinery and Intelligence". In: *Mind* LIX.236 (Oct. 1, 1950), pp. 433–460. DOI: 10.1093/mind/LIX.236.433.

[TP10] P. D. Turney and P. Pantel. "From Frequency to Meaning: Vector Space Models of Semantics". In: *Journal of Artificial Intelligence Research* 37 (Feb. 27, 2010), pp. 141–188. DOI: 10.1613/jair.2934.

[vdWCV11] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[Van04] André Van Tonder. "A Lambda Calculus for Quantum Computation". In: *SIAM Journal on Computing* 33.5 (2004), pp. 1109–1135.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. Dec. 5, 2017. arXiv: 1706.03762 [cs].

[Vic21] Irene Vicente Nieto. *Towards Machine Translation with Quantum Computers*. 2021. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-196602 (visited on 12/02/2021).

[Vor77] Rodiani Voreadou. *Coherence and Non-Commutative Diagrams in Closed Categories*. Vol. 9. Memoirs of the American Mathematical Society 182. American Mathematical Society, 1977. DOI: 10.1090/memo/0182.

[Wal+14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. "Scikit-Image: Image Processing in Python". In: *PeerJ* 2 (June 19, 2014), e453. DOI: 10.7717/peerj.453.

[Wea55] Warren Weaver. “Translation”. In: *Machine translation of languages* 14.15-23 (1955), p. 10.

[Wei88] David Jeremy Weir. “Characterizing Mildly Context-Sensitive Grammar Formalisms”. Philadelphia, PA, USA: University of Pennsylvania, 1988.

[Wie+19] Nathan Wiebe, Alex Bocharov, Paul Smolensky, Matthias Troyer, and Krysta M. Svore. *Quantum Language Processing*. Feb. 13, 2019. arXiv: 1902.05162 [quant-ph].

[WBL12] Nathan Wiebe, Daniel Braun, and Seth Lloyd. “Quantum Algorithm for Data Fitting”. In: *Physical Review Letters* 109.5 (Aug. 2, 2012), p. 050505. DOI: 10.1103/PhysRevLett.109.050505.

[Wit53] Ludwig Wittgenstein. *Philosophical Investigations*. Oxford: Basil Blackwell, 1953.

[Wu+21] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Qingling Zhu, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. “Strong Quantum Computational Advantage Using a Superconducting Quantum Processor”. In: *Physical Review Letters* 127.18 (Oct. 25, 2021), p. 180501. DOI: 10.1103/PhysRevLett.127.180501.

[YK21] Richie Yeung and Dimitri Kartsaklis. *A CCG-Based Version of the DisCoCat Framework*. May 24, 2021. arXiv: 2105.07720 [cs, math].

[Yon+21] Yong, Liu, Xin, Liu, Fang, Li, Haohuan Fu, Yuling Yang, Jiawei Song, Pengpeng Zhao, Zhen Wang, Dajia Peng, Huarong Chen, Chu Guo, Heliang Huang, Wenzhao Wu, and Dexun Chen. “Closing the ”Quantum Supremacy” Gap: Achieving Real-Time Simulation of a Random Quantum Circuit Using a New Sunway Supercomputer”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 14, 2021), pp. 1–12. DOI: 10.1145/3458817.3487399.

[ZC16] William Zeng and Bob Coecke. "Quantum Algorithms for Compositional Natural Language Processing". In: *Electronic Proceedings in Theoretical Computer Science* 221 (Aug. 2, 2016), pp. 67–75. DOI: 10.4204/EPTCS.221.8.

[Zho+20] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, and Jian-Wei Pan. "Quantum Computational Advantage Using Photons". In: *Science* 370.6523 (Dec. 18, 2020), pp. 1460–1463. DOI: 10.1126/science.abe8770.

[Zhu+21] Qingling Zhu, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yulin Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. "Quantum Computational Advantage via 60-Qubit 24-Cycle Random Circuit Sampling". In: *Science Bulletin* (Oct. 25, 2021). DOI: 10.1016/j.scib.2021.10.017.