Category Theory for Quantum Natural Language Processing

Alexis TOUMI

Wolfson College

University of Oxford

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

Doctor of Philosophy

January 27, 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 (Chapter 1). 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 (Chapter 2). 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 (Chapter 3).

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 [Bal21].

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+20b; Coe+20] 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+20a] 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; VN21], 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 nick-name) 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 [Law70b] subsumed Grothendieck's definition and emphasised the foundational concept of *adjunction* [Law69; Law70a]. "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 [VT04; 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 [Kv20; Dun+20; dBW20], compilation [CSD20; dD20] 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 [Kv19] 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 (as in free beer and in free speech) at:

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

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 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[1] give an elementary definition of string diagrams for monoidal categories. Our construction decomposes the free monoidal category construction into three basic steps: 1) a layer monad 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.

- We prove the equivalence between our elementary definition and the topological definition of Joyal and Street [JS88]. 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 describe our object-oriented implementation of monoidal category theory. 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.

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

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

[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.”

- 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 third chapter 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 parameterised functors in general.

Publications

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

[Mei+20b] 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`. arXiv: `2005.04147`.

[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+20] 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+20a] 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`.

[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.

[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`.

[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`.

[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`.

1

DisCoPy: monoidal categories in Python

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

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

This chapter contains a description of the DisCoPy package alongside an elementary list-based definition of string diagrams. The first section introduces categories and functors for the Python programmer, i.e. with no mathematical prerequisites apart from sets and monoids. The second section introduces monoidal categories for the Python programmer, defining string diagrams from first principles. The third section gives the category theoretic foundations for our definition, which we call the premonoidal approach. The fourth section defines the drawing and reading algorithms for string diagrams, which arise as the two sides of the equivalence between the premonoidal and the topological definitions. The fifth section introduces monoidal categories with extra structure (rigid, biclosed, symmetric, cartesian, hypergraph) and the inheritence mechanism which implements this hierarchy of structure. The last section discusses the relationship between our list-based premonoidal approach and the exisiting graph-based definitions of diagrams in symmetric monoidal categories.

1.1 Categories in Python

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

- C_0 and C_1 are classes of *objects* and *arrows* respectively,
- $\texttt{dom}, \texttt{cod} : C_1 \to C_0$ are functions called *domain* and *codomain*,
- $\texttt{id} : C_0 \to C_1$ 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:

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

- $\texttt{id}(x) : x \to x$ 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 `Object` and `Arrow` of listing 1.1.1, and then checking that the axioms hold via some (necessarily non-exhaustive) software tests.

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

Note that annotations with dependent types are not supported by any Python implementation yet. Since Python could not statically check that compositions are well-typed, DisCoPy has no type hints and raises an `AxiomError` *at runtime instead.*

```python
class Object: ...

class Arrow:
    dom: Object, cod: Object

    @staticmethod
    def id(x: Object) -> Arrow[x, x]: ...

    def then(self, other: Arrow[self.cod, y]) -> Arrow[self.dom, y]: ...

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

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

class Transformation:
    dom: Functor, cod: Functor

    def __call__(self, x: Object) -> Arrow[self.dom(x), self.cod(x)]: ...
```

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. *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 η : **Id** $\to$ **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 1.1.3. Implementation of the category **Pyth** with `type` as objects and `Function` as arrows.

```
@dataclass
class Function:
    inside: Callable
    dom: type
    cod: type

    @staticmethod
```

```python
    def id(x: type) -> Function:
        return Function(lambda *xs: xs, x, x)

    def then(self, other: Function) -> Function:
        return Function(lambda *xs: other(*self(*xs)), self.dom, other.cod)

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

f = Function(range, int, Iterable)
g = Function(sum, Iterable, int)
h = Function(lambda n: n * (n - 1) // 2, int, int)
assert f.then(g)(42) == h(42) == 861
```

Example 1.1.4. *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 1.1.5. *When the class of objects and arrow are finite sets, we can draw the category as a directed multigraphs 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. For example, 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 $f \fatsemi g = h$.*

$$
\begin{tikzcd}
\texttt{int} \arrow[rr, "\texttt{lambda n: n * (n - 1) // 2}"] \arrow[dr, "\texttt{range}"'] & & \texttt{int} \\
& \texttt{Iterable} \arrow[ur, "\texttt{sum}"'] &
\end{tikzcd}
$$

Thus, this commutative diagram is the equation `sum(range(n)) = n * (n - 1) // 2`*. When the finite category is bigger than a triangle, one commutative diagram can state a large number of equations, which can be read by* diagram chasing*.*

Example 1.1.6. *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): 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 (i.e. injective on arrows with the same domain and codomain)* $\mathbf{FinSet} \to \mathbf{Mat}_{\mathbb{B}}$ *which sends finite sets to their cardinality and functions to their graph.*

Listing 1.1.7. Implementation of the Boolean semiring $\mathbb{B}$ with addition and multiplication defined as disjunction and conjunction.

```
class B:
    def __init__(self, inside: bool): self.inside = bool(inside)
    __add__ = __radd__ = lambda self, other: B(self.inside or other)
    __mul__ = __rmul__ = lambda self, other: B(self.inside and other)
    __bool__ = lambda self: self.inside
    __eq__ = lambda self, other: bool(self) == bool(other)
    __repr__ = __str__ = lambda self: repr(int(self.inside))
```

Listing 1.1.8. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ with `int` as objects and `Matrix` with entries in $\mathbb{S} =$ `dtype` as arrows.

```
class Matrix:
    dtype = B

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

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

    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 Matrix(inside, self.dom, other.cod)

    __getitem__ = lambda self, key: self.inside[key]
    __repr__ = lambda self: "Matrix({}, {}, {})".format(
        self.inside, self.dom, self.cod)
```

Example 1.1.9. *The category* **Circ** *has natural numbers as objects and n-qubit quantum circuits as arrows $n \to n$. There is a functor* `eval` $: \mathbf{Circ} \to \mathbf{Mat}_{\mathbb{C}}$ *which sends n qubits to 2^n dimensions and evaluates each circuit to its unitary matrix.*

Example 1.1.10. *Just about any class of mathematical structures as objects and their homomorphisms as arrows will form a category. For example, the category* **Set** *of sets and functions, the category* **Mon** *of monoids and homomorphisms, the category* **Cat** *of small categories and functors, etc. The faithful functor $U : \mathbf{Mon} \to \mathbf{Set}$ which sends monoids to their underlying set and homomorphisms to functions is called a* forgetful functor*.*

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 1.1.2: 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.

[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 $C(-, U(y)), D(F(-), y) : C \to \mathbf{Set}$.

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 $f \mathbin{⨟} \mathtt{dom} = \mathtt{dom} \mathbin{⨟} f$ and $f \mathbin{⨟} \mathtt{cod} = \mathtt{cod} \mathbin{⨟} f$. Thus, signatures and their morphisms form a category $\mathbf{Sig}$ and there is a faithful functor $U : \mathbf{Cat} \to \mathbf{Sig}$ 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 $\mathtt{dom}(f_1) = x$, $\mathtt{cod}(f_n) = y$ and $\mathtt{cod}(f_i) = \mathtt{dom}(f_{i+1})$ 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 1.1.11.

Listing 1.1.11. Outline of the classes `Ob`, `Arrow` and `Box`.

```
@dataclass
class Ob:
```

```python
    name: str
    __str__ = lambda self: self.name

@dataclass
class Arrow:
    dom: Ob
    cod: Ob
    boxes: list[Arrow]

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

    @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
        boxes = self.boxes + sum([other.boxes for other in others], [])
        return self.upgrade(Arrow(dom, cod, boxes))

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

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

    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.boxes == [self]

    upgrade = Arrow.upgrade
    __str__ = lambda self: self.name
```

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.boxes == [f]`. In order 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 set to be 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`, otherwise we would be able to compose a diagram then a box but not a box then a diagram.

Example 1.1.12. *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(Circuit, Box):
    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, Id = Gate("X", 1), Gate("Y", 1), Gate("Z", 1), Circuit.id(Ob('1'))
assert (X >> Y) >> Z == X >> (Y >> Z) and X >> Id == X == Id >> X
assert isinstance(Id, Circuit) and isinstance(X >> Y, Circuit)
```

The `Functor` class listed in 1.1.13 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 `ob_factory` and `ar_factory` serve to define functors with arbitrary categories as codomain. At this point, their only use is for `ar_factory` to define identity arrows, otherwise the codomain of the functor is defined implicitly by the codomain of the `ob` and `ar` mappings.

Listing 1.1.13. Outline of the `Functor` class.

```python
@dataclass
class Functor:
    ob: dict[Ob, Ob]
    ar: dict[Box, Arrow]
    ob_factory, ar_factory = Ob, Arrow

    def __call__(self, other):
        if isinstance(other, Ob): return self.ob[other]
        if isinstance(other, Box):
            result = self.ar[other]  # This will allow some nice syntactic sugar.
            if not isinstance(result, self.ar_factory):
                result = self.ar_factory(result, self(other.dom), self(other.cod))
            return result
        if isinstance(other, Arrow):
            return self.ar_factory.id(self(other.dom)).then(
                *self(box) for box in other.boxes)
        raise TypeError
```

Example 1.1.14. *A typical DisCoPy script starts by defining objects and boxes:*

```python
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:

```python
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:

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

We can interpret our arrows as matrices:

```python
H = Functor(
    ob={x: 1, y: 2, z: 2},
    ar={f: [[0, 1]], g: [[0, 1], [1, 0]], h: [[1], [0]]},
    ob_factory=int, ar_factory=Matrix)
assert H(f >> g) == H(h)
```

Provided we implement `dom`, `cod`, `id` *and* `then` *for the* `Functor` *class, we can even build functors into* **Cat**, *i.e. interpret arrows as functors:*

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

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 $F = Q \,\mathbin{⨾}\, F'$. 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 $R = \{(f \mathbin{⨾} g, h) \in F(\Sigma) \times F(\Sigma) \mid f \mathbin{⨾} g = h \in C\}$. 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 1.1.15. *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:*

- $H \mathbin{⨾} X = Z \mathbin{⨾} H$ *and* $Z \mathbin{⨾} X = (-1) \mathbin{⨾} X \mathbin{⨾} Z$,
- $f \mathbin{⨾} f = \mathtt{id}(1)$ *and* $f \mathbin{⨾} (-1) = (-1) \mathbin{⨾} f$ *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* `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.*

We conclude this section by discussing three extra pieces of implementation beyond the basics of category theory: dagger, 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. $(\dagger) \mathbin{⨟} (\dagger) = \mathtt{id}_{\mathbb{C}}$. DisCoPy implements free $\dagger$-categories by adding an attribute `is_dagger: bool` to boxes and a method `Arrow.dagger`, shortened to the postfix `[::-1]`, which reverses the order of boxes and negates `is_dagger` elementwise. A $\dagger$-functor is a functor between $\dagger$-categories which commutes with the dagger, they are implemented by adding a case to the code for functor application. For example, the conjugate transpose defines a dagger on the category $\mathbf{Mat}_{\mathbb{S}}$, the adjoint defines a dagger on the category **Circ** and the evaluation $\mathbf{Circ} \to \mathbf{Mat}_{\mathbb{S}}$ is a $\dagger$-functor.

Listing 1.1.16. Implementation of free $\dagger$-categories and $\dagger$-functors.

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

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

Listing 1.1.17. Implementation of $\mathbf{Mat}_{\mathbb{S}}$ as a $\dagger$-category.

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

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

def dagger(self: Matrix) -> Matrix:
    return self.transpose().map(lambda x: x.conjugate())
```

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].boxes == f.boxes[key]` for any `key: int` and any `key: slice` with `key.step in (-1, 1, None)`.

Listing 1.1.18. Outline of indexing and slicing for arrows.

```python
def __getitem__(self: Arrow, key: int | slice) -> Arrow:
    if isinstance(key, int): return self.upgrade(self.boxes[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: raise NotImplementedError
        return self[key.stop + 1:key.start + 1].dagger()
    dom, cod = self[key.start].dom, self[key.stop].cod
    return self.upgrade(Arrow(dom, cod, self.boxes[key]))
```

Remark 1.1.19. *Although the case of negative indices (i.e. counting backwards from the end of the list) is implemented in DisCoPy, its interactions with list reversal are too complex to fit in a thesis.*

Example 1.1.20. *We can implement a simulator for* 1*-qubit circuits as a* $\dagger$*-functor.*

```python
F = Functor(
    ob={Ob('1'): 2},
    ar={Y: [[0, -1j],
            [1j, 0]]})

assert F(Y >> Y[::-1]) == F(Id) == F(Y[::-1] >> Y)
```

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. In DisCoPy, free categories with sums are implemented by `Sum`, a subclass of `Box` with an attribute `terms: list[Arrow]`. The method `then` is straightforward: the composition of a sum is the sum of the compositions of its terms. Defining equality requires some extra care however: we want an arrow to be equal to the sum of just itself, we also want two sums to be equal when their list of terms are permutations of each other. DisCoPy functors are commutative-monoid-enriched, i.e. formal sum of arrows can be interpreted as a concrete sum of matrices.

Listing 1.1.21. Implementation of free sum-enriched categories and functors.

```python
class Arrow(cat.Arrow):
    def __add__(self, other):
        self, other = map(Sum.upgrade, (self, other))
        return Sum(self.terms + other.terms, self.dom, self.cod)

    def __eq__(self, other):
        return other.terms == [self] if isinstance(other, Sum) else super().__eq__(other)

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

class Sum(Arrow, cat.Box):
    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 upgrade(old: cat.Arrow) -> Sum:
        return old if isinstance(old, Sum) else Sum([old], old.dom, old.cod)

    def then(self, other):
```

```python
        terms = [f.then(g) for f in self.terms for g in Sum.upgrade(other).terms]
        return Sum(terms, self.dom, other.cod)

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

Listing 1.1.22. Implementation of $\mathbf{Mat}_\mathbb{S}$ as a category with sums.

```python
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 Matrix(inside, self.dom, self.cod)

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

By a *bubble* we mean an operator which takes an arrow and puts it into a box. More formally, a bubble in a category C is a pair of functions $b_{\texttt{dom}}, b_{\texttt{cod}} : C_0 \to C_0$ between objects and a unary operator between homsets $b : C(x, y) \to C(b_{\texttt{dom}}(x), b_{\texttt{cod}}(y))$ for each pair of objects $x, y \in C_0$. DisCoPy implements the free category with bubbles via a `Bubble` class initialised by `dom`, `cod` and an attribute `inside: Arrow`. DisCoPy functors automatically respect bubbles, i.e. we have that $F(b(f)) = b(F(f))$ for all arrows f. Thus, we can interpret arrows with bubbles as arbitrary operations on the codomain of our interpretation functors. For example, we can define a negation bubble on the category $\mathbf{Mat}_\mathbb{B}$ of Boolean matrices: it is the identity on objects and sends each matrix f to its entrywise negation $\bar{f}$. The resulting syntax with bubbles is strictly more expressive than that of free categories alone: negation cannot be expressed as a composition, there is no matrix n in $\mathbf{Mat}_\mathbb{B}$ such that $f \fatsemi n = \bar{f}$ for all matrices f. This is the case for the element-wise application of any non-linear function, such as the rectified linear units (ReLU) used in neural networks. As we will discuss in Chapter 3, differentiation of parameterised matrices cannot be expressed as a composition either, but it is a unary operator between homsets, i.e. a bubble.

Listing 1.1.23. Implementation of free categories with bubbles and their functors.

```python
class Bubble(Box):
    def __init__(self, inside: Arrow, dom: Optional[Ob], cod: Optional[Ob]):
```

```
        dom, cod = dom or self.dom, cod or self.cod
        self.inside, name = inside, "Bubble({}, {}, {})".format(inside, dom, cod)
        super().__init__(name, dom, cod)

Arrow.bubble = Bubble

class Functor(cat.Functor):
    def __call__(self, other):
        if isinstance(other, Bubble):
            return self.ar_factory.bubble(self(other.dom), self(other.cod))
        return super().__call__(other)
```

Example 1.1.24. *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.dtype, Matrix.bubble = float, 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()

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]]})

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

Example 1.1.25. *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.dtype, Matrix.bubble = B, lambda self: self.map(lambda x: not x)

p, q = Box('p', x, x), Box('q', x, x)
```

```
p_implies_q = q.bubble().then(p).bubble()
not_p_or_q = p.bubble() + q

for a, b in itertools.product([0, 1], [0, 1]):
    F = Functor(ob={x: 1}, ar={p: [[a]], q: [[b]]}, ar_factory=Matrix)
    ev = lambda f: F(f)[0][0]  # Take the scalar out of a 1x1 matrix.
    assert ev(p_implies_q) == not (not ev(q) and ev(p))\
        == ev(not_p_or_q) == not ev(p) or ev(q)
```

1.2 String diagrams in Python

In the previous section, we introduced the idea of arrows in free categories as formal 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 $(C_1, \otimes, \mathtt{id}(1))$ 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 arrow $f : 1 \to x$ from the unit is called a *state* of the object 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*. The interchange law implies that the scalars form a commutative monoid, by the following Eckmann-Hilton argument:

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

[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. *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*[2] *to a strict one. In the case of* **Pyth***, there is an equivalent monoidal structure with flattened tuples instead:* `(t1 @ t2) @ t3 = t1 @ (t2 @ t3) = tuple[t1, t2, t3]`*.*

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 1.4,* **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 1.2.2. Implementation of **Pyth** as a monoidal category.

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

Example 1.2.3. *With some effort, we can also make* **Pyth** *monoidal with the* tagged union *as tensor on objects and* `typing.NoReturn` *as unit. Given two types*

[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.

[2]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.

`t1, t2`, *their tagged union* `t1 + t2` *is the union of the types* `tuple[True, t1]` *and* `tuple[False, t2]`[1], *i.e. a term* `(b, x): t1 + t2` *is a pair of a Boolean* `b: bool` *and a term* `x: t1` *if* `b` *else* `x: t2`. *Given two functions* `f, g` *we can define their tensor* `f + g = lambda b, x: (b, f(x) if b else g(x))`.

Example 1.2.4. *Every monoid M can also be seen as a discrete monoidal category, i.e. with only identity arrows.*

Example 1.2.5. *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 $[n] = \{0, 1, \dots, n-1\}$. The states can be identified with elements and discarding is the only effect.* **FinSet** *is also monoidal with the empty set 0 as unit and disjoint union as tensor.*

Example 1.2.6. *The category* $\mathbf{Mat}_\mathbb{S}$ *is monoidal with addition of natural numbers as tensor on objects and the* direct sum $f \oplus g = \left(\begin{smallmatrix} f & 0 \\ 0 & g \end{smallmatrix}\right)$ *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 1.2.7. Implementation of $\mathbf{Mat}_\mathbb{S}$ as a monoidal category with Kronecker product as tensor.

```
def tensor(self: Matrix, 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 Matrix(inside, dom, cod)
```

Example 1.2.8. *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* $\texttt{eval} : \mathbf{Circ} \to \mathbf{Mat}_\mathbb{C}$ *is monoidal: it sends the parallel composition of circuits to the Kronecker product of their unitary matrices.*

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`

[1]What we really mean is `tuple[Literal[True], t1] | tuple[Literal[False], t2]`.

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 $F : \mathbf{MonSig} \to \mathbf{MonCat}$, the left adjoint to the forgetful functor $U : \mathbf{MonCat} \to \mathbf{MonSig}$ 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 $f \mathbin{⨟} \texttt{dom} = \texttt{dom} \mathbin{⨟} f^\star$ and $f \mathbin{⨟} \texttt{cod} = \texttt{cod} \mathbin{⨟} f^\star$. Thus, we have defined the category $\mathbf{MonSig}$ of monoidal signatures and their morphisms. In order to define the forgetful functor $U : \mathbf{MonCat} \to \mathbf{MonSig}$, we will need the following technical lemma.

Definition 1.2.9. *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 1.2.10. *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 1.2.11. *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 $x_1 \times \dots \times x_n = y_1 \times \dots \times y_m$ in M.*

Example 1.2.12. *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 called* coloured PROs, *but a coloured PRO is precisely a foo monoidal category.*

Example 1.2.13. *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 $(x_1 \times \dots \times x_n) \times (y_1 \times \dots \times y_m)$ 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 1.2.14. Implementation of $\mathbf{Tensor}_\mathbb{S} \simeq \mathbf{Mat}_\mathbb{S}$.

```python
def product(x: list[int]) -> int: return reduce(lambda a, b: a * b, x, 1)

@dataclass
class Tensor:
    inside: list[list[Number]]
    dom: list[int]
    cod: list[int]

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

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

    def then(self, other: Tensor) -> Tensor:
        inside = self.downgrade().then(other.downgrade()).inside
        return Tensor(inside, self.dom, other.cod)

    def tensor(self, other: Tensor) -> Tensor:
```

```
        inside = self.downgrade().tensor(other.downgrade()).inside
        return Tensor(inside, self.dom + other.dom, self.cod + other.cod)
```

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 1.2.15. *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 1.2.16. *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:*

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

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

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 $f \otimes g = (f \mathbin{;} \mathtt{id}(y)) \otimes (\mathtt{id}(z) \mathbin{;} g) = f \otimes \mathtt{id}(z) \mathbin{;} \mathtt{id}(y) \otimes g$. □

We now 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*:

$$(axb,\ g,\ c) \ \mathbin{;} \ (a,\ f,\ bwc) \quad \to_R \quad (a,\ f,\ bzc) \ \mathbin{;} \ (ayb,\ g,\ c)$$

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 $(1, f, 1) \fatsemi (1, g, 1) \leftarrow_R (x, g, 1) \fatsemi (1, f, y)$. Delpeuch and Vicary [DV18] give a quadratic solution to the word problem for monoidal categories, i.e. deciding when two diagrams are equal.

Theorem 1.2.17 ([DV18])**.** *The equality of diagrams is decidable in linear time in the connected case, and quadratic in the general case. For connected diagrams, i.e. when the Eckmann-Hilton argument does not apply, the right interchanger is confluent and it reaches a normal form in a cubic number of steps.*

Now that 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. Now 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 related by the interchanger.

We can check that we have indeed defined a monoidal category $F(\Sigma)$. 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.

Listing 1.2.18. Outline of the class `monoidal.Ty`.

```python
class Ty(Ob):
    def __init__(self, *objects: Ob | str):
        self.objects = [x if isinstance(x, Ob) else Ob(x) for x in objects]
        super().__init__(name="Ty({})".format(
            ', '.join([x.name for x in self.objects])))

    @classmethod
    def upgrade(cls, old: Ob) -> Ty:
```

```python
        if isinstance(old, cls): return old
        return cls(*old.objects) if isinstance(old, Ty) else cls(old)

    def tensor(self, *others: Ty) -> Ty:
        if any(not isinstance(other, Ty) for other in others):
            return NotImplemented  # This allows whiskering on the left.
        return self.upgrade(Ty(*self.objects + sum(
            [other.objects for other in others], [])))

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

The implementation of the class `Ty` for types (i.e. lists of objects) is straightforward, it is sketched in listing 1.2.18. 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 1.2.19. *We can define a* `Qubits` *subclass and be sure that the tensor of qubits is still an instance of* `Qubits`*, not merely* `Ty`*.*

```python
class Qubits(Ty):
    def __init__(self, n: int):
        super().__init__(self, n * [Ob("qubit")])

    def upgrade(old):
        return Qubits(len(old.boxes))

qubit = Qubits(1)
assert qubit @ qubit == Qubits(2) and isinstance(qubit @ qubit, Qubits)
```

The implementation of `Layer` as a subclass of `cat.Box` is sketched in listing 1.2.20. 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.

Listing 1.2.20. Outline of the class `monoidal.Layer`.

```python
class Layer(cat.Box):
    def __init__(self, left: Ty, box: cat.Box, right: Ty):
        self.left, self.box, self.right = left, box, right
        name = "Layer({}, {}, {})".format(left, box, right)
        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)

    @staticmethod
    def upgrade(old: cat.Box) -> Layer:
        return old if isinstance(old, Layer) else Layer(Ty(), old, Ty())
```

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. The `interchange` method takes an integer `i < len(self.layers)` and returns the diagram with layers `i` and `i + 1` interchanged, or raises an `AxiomError` if their boxes are connected. It also takes an optional argument `left: bool` which allows to choose between left and right interchangers. 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 1.2.21. Outline of the class `monoidal.Diagram`.

```
class Diagram(cat.Arrow):
    def __init__(self, dom: Ty, cod: Ty, layers: list[Layer]):
        self.layers = layers; super().__init__(dom, cod, boxes=layers)

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

    __matmul__ = tensor

    def interchange(self, i: int, left=False) -> Diagram: ...

    def normal_form(self, left=False) -> Diagram: ...

    def draw(self, **params): ...

    @classmethod
    def upgrade(cls, old: cat.Arrow) -> Diagram:
        if isinstance(old, cls): return old
```

```
        layers = list(map(Layer.update, old.boxes))
        dom, cod = map(Ty.upgrade, (old.dom, old.cod))
        return cls(dom, cod, layers)
```

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.

Example 1.2.22. *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`*.*

```
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, bit: bool):
        Box.__init__(self, "Ket({})".format(bit), Qubits(0), Qubits(1))

Gate.upgrade = Ket.upgrade = Circuit.upgrade

H, CX = Gate("H", n_qubits=1), Gate("CX", n_qubits=2)
Id, sqrt2 = Circuit.id(Qubits(1)), Gate("sqrt(2)", n_qubits=0)
assert isinstance(sqrt2 @ Ket(0) @ Ket(0) >> H @ Id >> CX, Circuit)
```

The implementation of `monoidal.Box` as a subclass of `Diagram` and `cat.Box` 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 1.2.23. Outline of the class `monoidal.Box`.

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

    def __init__(self, name: str, dom: Ty, cod: Ty):
        cat.Box.__init__(self, name, dom, cod)
        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.layers == [Layer.upgrade(self)]
```

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 1.2.24. Outline of the class `monoidal.Functor`.

```
class Functor(cat.Functor):
    def __call__(self, other):
        if isinstance(other, Ty):
            return self.ob_factory.tensor(
                *[self(x) for x in other.objects])
        if isinstance(other, Layer):
            left, box, right = other
            return self(left) @ self(box) @ self(right)
        return super().__call__(other)
```

Example 1.2.25. *We can simulate quantum circuits by applying a functor from* `Circuit` *to* `Tensor`*.*

```
Circuit.eval = Functor(
    ob={Ob("qubit"): [2]},
    ar={H: [[1 / sqrt(2), 1 / sqrt(2)],
            [1 / sqrt(2), -1 / sqrt(2)]],
        CX: [[1, 0, 0, 0],
             [0, 1, 0, 0],
             [0, 0, 0, 1],
             [0, 0, 1, 0]],
        Ket(0): [[0, 1]], Ket(1): [[1, 0]], sqrt2: [[sqrt(2)]]},
    ar_factory=Tensor)

circuit = sqrt2 @ Ket(0) @ Ket(0) >> H @ Id >> CX
superposition =  Ket(0) @ Ket(0) + Ket(1) @ Ket(1)
assert circuit.eval() == superposition.eval()
```

Remark 1.2.26. *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.*

```
def encode(diagram: Diagram) -> tuple[Ty, list[tuple[Box, int]]]:
    return diagram.dom, [(box, len(left)) for left, box, _ in diagram.layers]
```

```python
def decode(dom: Ty, boxes_and_offsets: list[tuple[Box, int]]) -> Diagram:
    result = Diagram.id(dom)
    for box, offset in boxes_and_offsets:
        left, right = result.cod[:offset], result.cod[offset + len(box.dom):]
        result >>= Diagram.id(left) @ box @ Diagram.id(right)
    return result

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

As in the previous section, we introduce three extra pieces of implementation: dagger, sums and bubbles. A †-monoidal category is a monoidal category with a † (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.

Listing 1.2.27. Implementation of free †-monoidal categories.

```python
def dagger(self: Layer) -> 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 method `Sum.tensor`.

Listing 1.2.28. Implementation of free monoidal categories with sums.

```python
def tensor(self: Sum, other: Sum) -> Sum:
    dom, cod = self.dom @ other.dom, self.cod @ other.cod
    return Sum([f @ g for f in self.terms for g in self.terms], dom, cod)
```

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.

Example 1.2.29. *We can implement the formulae of first-order logic using Peirce's* existential graphs *which happen to be the first examples of string diagrams [BT98; BT00; MZ16; HS20] as well as the first definition of first-order logic. 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* $\mathbf{Mat}_{\mathbb{B}}$.

```
men, mortal = Box("men", x, y), Box("mortal", y, x)
all_men_are_mortal = men.bubble().then(mortal).bubble()
for a, b, c, d in itertools.product(*4 * [[0, 1]]):
    ev = Functor(
        ob={x: 1, y: 2},
        ar={men: [[a, b]], mortal: [[c], [d]]}, ar_factory=Matrix)
    assert ev(all_men_are_mortal) == all(
        not ev(men)[0][i] or ev(mortal)[i][0] for i in range(ev(y)))
```

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 syntax for Python functions instead of the point-free syntax. Given `dom: Ty`, `cod: Ty` and `boxes: 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 1.2.30. *We can define quantum circuits as Python functions on qubits.*

```
@diagramize(dom=Qubits(2), cod=Qubits(2), boxes=[sqrt2, Ket, H, CX])
def circuit():
    sqrt2(); qubit0, qubit1 = Ket(0)(), Ket(0)()
    return CX(H(qubit0), qubit1)

assert circuit == sqrt2 @ Ket(0) @ Ket(0) >> H @ Id >> 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 a domain node of a box (or a codomain node of the whole diagram) whenever the corresponding boxes 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 1.2.31. Translation from `Diagram` to `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.layers):
        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. We also need to assume that every codomain node is connected to exactly one domain node. In sections 1.5 and 1.6 we will discuss the case of diagrams induced by non-planar graphs, with potentially multiple edges between domain and codomain nodes. Listing 1.2.31 shows the implementation of the inverse translation `diagram2graph`. This 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 and 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.

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 *edges* respectively, such that:

- G_0 is discrete and $G - G_0 = \bigcup G_1$,
- each edge $e \in G_1$ 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 edge $e \in G_1$, i.e. edges 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 edge $e \in G_1$, i.e. edges 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 edges 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 edges 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 edges 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 edges 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 : G₁ → Ty, λ : boxes(G) → Box ) -> Diagram:
    dom = [ λ(e) for x ∈ dom(G) for e ∈ G₁ if x ∈ ∂e ]
    boxes = [ λ(x) for x ∈ boxes(G) ]
    offsets = [len( G₁ ∩ {p₀(x)} × ℝ ) for x ∈ boxes(G) ]
    return decode(dom, zip(boxes, offsets))
```

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 edge 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 edge a straight line and as vertical as possible. We first initialise the lgpp graph of the identity with a constant spacing between each edge, then for each layer we update the embedding so that there is enough space for the output edges of the box before we add it to the graph.

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

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

```python
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.layers):
        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
```

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 edges of the box node: they are hidden by the rectangle. The only exceptions are *spider boxes* where we draw the box node (the head) and its outgoing edges (the legs of the spider) as well as *cup and cap boxes* where we do not draw the box node at all, only its two outgoing edges which are drawn as Bézier curves to look like cups and caps respectively. Spiders, cups and caps will be discussed, and drawn, in section 1.5.

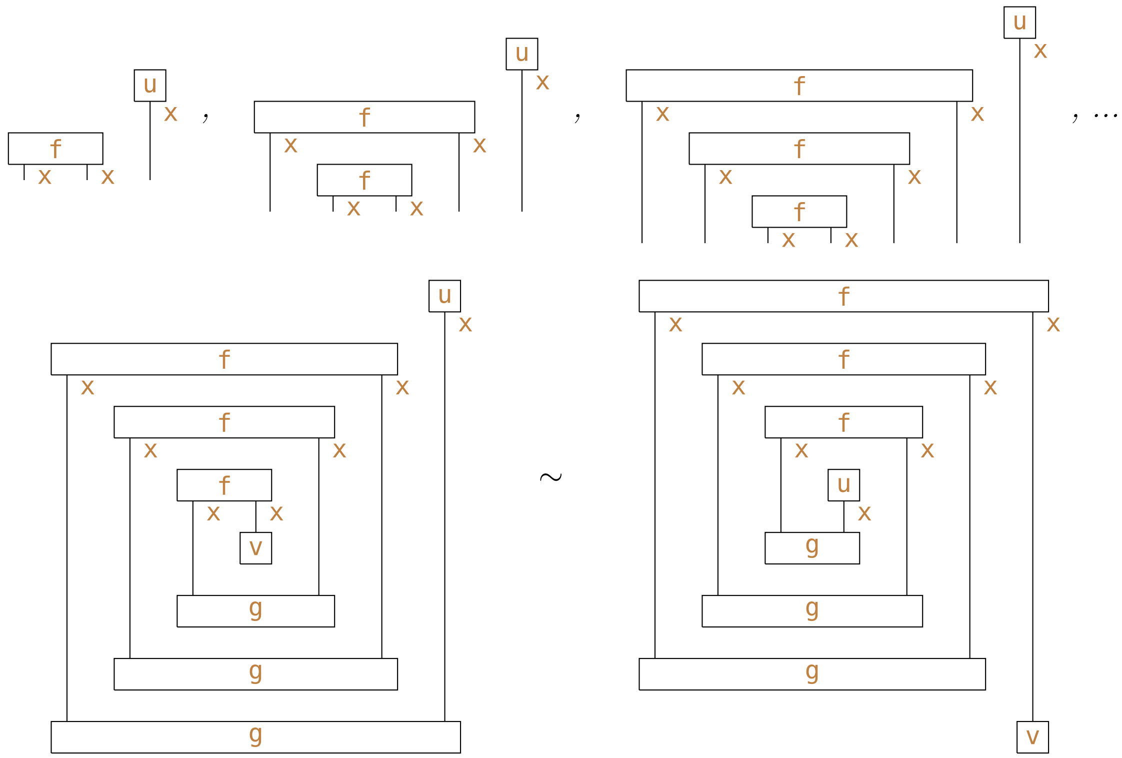
Example 1.3.3. *The following spiral diagram is the cubic worst-case for interchanger normal form. It is also the quadratic worst-case for drawing, at each layer of the first half we need to update the position of every preceding layer in order to make space for the output edges.*

```python
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

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



Next, we define the inverse translation `graph2diagram`.

Listing 1.3.4. 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 1.3.5. `graph2diagram(self.draw()) == self` *for all* `self: Diagram`*.*

Proof. By induction on `n = len(self.layers)`. If `not len(self.layers)` we get that `dom == self.dom` and `boxes == offsets == []`. If the theorem holds for `self`, it holds for `self >> Layer(left, box, right)`. Indeed, we have:

- `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)]`. □

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 1.3.6. *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 edge 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 edges in the middle. We can apply the case of the identity to the left and right slices, for the middle slice we make the edges 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 edges straight, we need to make space for the edges going out of the box.

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

Theorem 1.3.7. *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. □

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 $\mathtt{id}(x_1 \dots x_n)$ is the graph with edges $(i, a) \to (i, b)$ 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 edges $s^+(\mathtt{cod}(G)_i) \to s^-(\mathtt{dom}(G')_i) \in E$ for each $i \leq \mathtt{len}(\mathtt{cod}(G)) = \mathtt{len}(\mathtt{dom}(G'))$.

The deformations for the unitality axioms are straightforward: there is a deformation $G \fatsemi \mathtt{id}(\mathtt{cod}(G)) \sim G \sim \mathtt{id}(\mathtt{dom}(G)) \fatsemi G$ which contracts the edges 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 following hand-drawn diagrams from Joyal and Street.

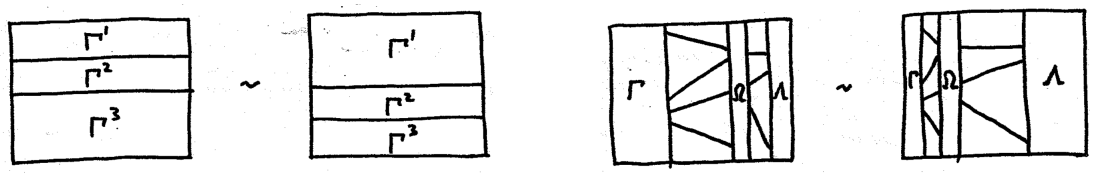


Figure 1.1: Deformations for the associativity of tensor and composition.

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

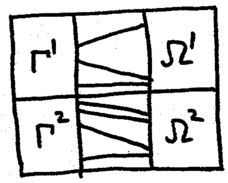


Figure 1.2: The graph of the interchange law.

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 $f ⨟ \lambda$, its image on arrows is given in listing 1.3.8. 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 1.3.8. Implementation of the functor $G : \mathbf{Monsig} \to \mathbf{MonCat}$ on arrows.

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

def G(f: SigMorph) -> Callable[[Graph], Graph]:
    def G_of_f(graph: Graph) -> Graph:
        def relabel(node):
            if node.kind == 'box':
                return Node('box', f[1][node.label], node.i, node.j)
            return Node(node.kind, f[0][node.label], node.i, node.j)
        return Graph(map(relabel, graph.edges))
    return G_of_f
```

Theorem 1.3.9. *There is a natural isomorphism $F \simeq G$.*

Proof. From theorems 1.3.6 and 1.3.7, we have an isomorphism between `Diagram` and `PlaneGraph` given by `d2g = Diagram.draw` and `g2d = graph2diagram`. Now fix `F = lambda f: Functor(ob=f[0], ar=f[1])`. Given a morphism of monoidal signatures `f: SigMorph` we have two naturality squares:

$$\begin{tikzcd} \texttt{Diagram} \arrow[r, "\texttt{d2g}"] \arrow[d, "\texttt{F(f)}"'] & \texttt{PlaneGraph} \arrow[d, "\texttt{G(f)}"] \\ \texttt{Diagram} \arrow[r, "\texttt{d2g}"] & \texttt{PlaneGraph} \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} \texttt{PlaneGraph} \arrow[r, "\texttt{g2d}"] \arrow[d, "\texttt{G(f)}"'] & \texttt{Diagram} \arrow[d, "\texttt{F(f)}"] \\ \texttt{PlaneGraph} \arrow[r, "\texttt{g2d}"] & \texttt{Diagram} \end{tikzcd}$$

□

1.4 The premonoidal approach

1.5 Adding extra structure

1.6 Related & future work

2
Quantum natural language processing

3
Diagrammatic differentiation

References

[Aar15] Scott Aaronson. "Read the Fine Print". In: *Nature Phys* 11.4 (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.

[AC04] Samson Abramsky and Bob Coecke. "A Categorical Semantics of Quantum Protocols". In: *CoRR* quant-ph/0402130 (2004).

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

[AB08] Dorit Aharonov and Michael Ben-Or. "Fault-Tolerant Quantum Computation with Constant Error Rate". In: *SIAM J. Comput.* 38.4 (Jan. 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: *SIGACT News* 35.2 (June 2004), pp. 22–35. DOI: 10.1145/992287.992296.

[Ari66] Aristotle. *Categories, and De Interpretatione.* Trans. by J. L. Ackrill. Oxford [England] : Clarendon Press ; New York : Oxford University Press, 1966.

[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 J. Phys.* 17.12 (Dec. 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 (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". In: *arXiv:2003.01664 [quant-ph]* (Mar. 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`.

[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.

[BE14] John C. Baez and Jason Erbele. "Categories in Control". In: *arXiv:1405.6881 [quant-ph]* (May 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: *Rev. Math. Phys.* 29.09 (Oct. 2017), p. 1750028. DOI: `10.1142/S0129055X17500283`. arXiv: `1704.02051`.

[BS09] John C. Baez and Mike Stay. “Physics, Topology, Logic and Computation: A Rosetta Stone”. In: *arXiv:0903.0340 [quant-ph]* (Mar. 2009). arXiv: `0903.0340 [quant-ph]`.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *arXiv:1409.0473 [cs, stat]* (May 2016). arXiv: `1409.0473 [cs, stat]`.

[Bal21] Philip Ball. *Major Quantum Computing Strategy Suffers Serious Setbacks.* https://www.quantamagazine.org/major-quantum-computing-strategy-suffers-serious-setbacks-20210929/. Sept. 2021.

[Ban+08] Jeongho Bang, James Lim, M. S. Kim, and Jinhyoung Lee. “Quantum Learning Machine”. In: *arXiv:0803.2976 [quant-ph]* (Mar. 2008). arXiv: `0803.2976 [quant-ph]`.

[BKV18] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. “Globular: An Online Proof Assistant for Higher-Dimensional Rewriting”. In: *Log. Methods Comput. Sci.* 14.1 (2018). DOI: `10.23638/LMCS-14(1:8)2018`.

[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`.

[BLS19] Marcello Benedetti, Erika Lloyd, and Stefan Sack. “Parameterized Quantum Circuits as Machine Learning Models”. In: *arXiv:1906.07682 [quant-ph]* (June 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: *J Stat Phys* 22.5 (May 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 J. Comput.* 26.5 (Oct. 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`.

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

[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*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 847–857.

[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`.

[BSS18] Filippo Bonchi, Jens Seeber, and Pawel Sobocinski. "Graphical Conjunctive Queries". In: *arXiv:1804.07626 [cs]* (Apr. 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*. London : Walton and Maberly, 1854.

[BT00] Geraldine Brady and Todd Trimble. "A Categorical Interpretation of C.S. Peirce's Propositional Logic Alpha". In: *Journal of Pure and Applied Algebra - J PURE APPL ALG* 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". In: (1998).

[Bra+02] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. “Quantum Amplitude Amplification and Estimation”. In: *arXiv:quant-ph/0005055* 305 (2002), pp. 53–74. DOI: `10.1090/conm/305/05215`. arXiv: `quant-ph/0005055`.

[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”. In: *arXiv:1611.08012 [quant-ph]* (Jan. 2018). arXiv: `1611.08012 [quant-ph]`.

[Che02] Joseph CH Chen. “Quantum Computation and Natural Language Processing”. PhD thesis. Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky, 2002.

[CJ19] Kenta Cho and Bart Jacobs. “Disintegration and Bayesian Inversion via String Diagrams”. In: *Math. Struct. Comp. Sci.* 29.7 (Aug. 2019), pp. 938–971. DOI: `10.1017/S0960129518000488`. arXiv: `1709.00322`.

[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: *Phys. Rev. Lett.* 80.15 (Apr. 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. 2007).

[CJS13] B. D. Clader, B. C. Jacobs, and C. R. Sprouse. “Preconditioned Quantum Linear System Algorithm”. In: *Phys. Rev. Lett.* 110.25 (June 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.

[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.

[Coe19] Bob Coecke. “The Mathematics of Text Structure”. In: (Apr. 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.

[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. arXiv: 1102.2368.

[Coe+20] 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+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.

[CSD20] Alexander Cowtan, Will Simmons, and Ross Duncan. "A Generic Compilation Strategy for the Unitary Coupled Cluster Ansatz". In: *arXiv:2007.10515 [quant-ph]* (Aug. 2020). arXiv: `2007.10515 [quant-ph]`.

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

[dBW20] Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. "Fast and Effective Techniques for T-count Reduction via Spider Nest Identities". In: *arXiv:2004.05164 [quant-ph]* (Apr. 2020). arXiv: `2004.05164 [quant-ph]`.

[dD20] Arianne Meijer-van de Griend and Ross Duncan. "Architecture-Aware Synthesis of Phase Polynomials for NISQ Devices". In: *arXiv:2004.06052 [quant-ph]* (Apr. 2020). arXiv: `2004.06052 [quant-ph]`.

[DV18] Antonin Delpeuch and Jamie Vicary. "Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm". In: *arXiv:1804.07832 [cs]* (Apr. 2018). arXiv: `1804.07832 [cs]`.

[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. 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". In: *arXiv:1810.04805 [cs]* (May 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 2020), p. 279. DOI: `10.22331/q-2020-06-04-279`. arXiv: `1902.03178`.

[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: *Trans. Amer. Math. Soc.* 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". In: *arXiv:1411.4028 [quant-ph]* (Nov. 2014). arXiv: 1411.4028 [quant-ph].

[FN18] Edward Farhi and Hartmut Neven. "Classification with Quantum Neural Networks on Near Term Processors". In: *arXiv:1802.06002 [quant-ph]* (Aug. 2018). arXiv: 1802.06002 [quant-ph].

[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.

[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.

[Fey82] Richard P. Feynman. "Simulating Physics with Computers". In: *Int J Theor Phys* 21.6 (June 1982), pp. 467–488. DOI: 10.1007/BF02650179.

[Fey85] Richard P Feynman. "Quantum Mechanical Computers". In: *Optics news* 11.2 (1985), pp. 11–20.

[Fir57] John R Firth. "A Synopsis of Linguistic Theory, 1930-1955". In: *Studies in linguistic analysis* (1957).

[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).

[Gha+18] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. "Compositional Game Theory". In: *arXiv:1603.04641 [cs]* (Feb. 2018). arXiv: `1603.04641 [cs]`.

[GF19] Craig Gidney and Austin G. Fowler. "Flexible Layout of Surface Code Computations Using AutoCCZ States". In: *arXiv:1905.08916 [quant-ph]* (May 2019). arXiv: `1905.08916 [quant-ph]`.

[GLM08] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum Random Access Memory". In: *Phys. Rev. Lett.* 100.16 (Apr. 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* (Sept. 2020).

[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. 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". In: *arXiv:1106.4058 [cs, math]* (2011), pp. 1394–1404. 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". In: *arXiv:1101.0309 [cs]* (Dec. 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: *Phys. Rev. Lett.* 79.2 (July 1997), pp. 325–328. DOI: 10.1103/PhysRevLett.79.325. arXiv: quant-ph/9706033.

[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*. LICS '18. New York, NY, USA: ACM, 2018, pp. 502–511. DOI: 10.1145/3209108.3209128.

[Har54] Zellig S. Harris. "Distributional Structure". In: *WORD* 10.2-3 (Aug. 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: *Phys. Rev. Lett.* 103.15 (Oct. 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. arXiv: 1804.11326.

[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". In: *arXiv:1807.07828 [cs]* (July 2018). arXiv: 1807.07828 [cs].

[Heg12] Georg Wilhelm Friedrich Hegel. *Wissenschaft Der Logik*. F. Frommann, 1812.

[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: *Int. J. Unc. Fuzz. Knowl. Based Syst.* 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. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[Hof21] Thomas Hoffmann. “Quantum Models for Word- Sense Disambiguation”. In: (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.

[Hot66] Günter Hotz. “Eindeutigkeit Und Mehrdeutigkeit Formaler Sprachen”. In: *J. Inf. Process. Cybern.* (1966). DOI: 10.5604/16431243.1040101.

[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*. LICS ’18. New York, NY, USA: ACM, 2018, pp. 559–568. DOI: 10.1145/3209108.3209131.

[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 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.

[Kan81] Immanuel Kant. *Critique of Pure Reason*. Trans. by Norman Kemp Smith. Read Books Ltd. (2011), 1781.

[KSP13] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, and Stephen Pulman. “Separating Disambiguation from Composition in Distributional Semantics”. In: (2013), p. 10.

[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`.

[KP16] Iordanis Kerenidis and Anupam Prakash. "Quantum Recommendation Systems". In: *arXiv:1603.08675 [quant-ph]* (Mar. 2016). arXiv: `1603.08675 [quant-ph]`.

[KU19] Aleks Kissinger and Sander Uijlen. "A Categorical Semantics for Causal Structure". In: *arXiv:1701.04732 [math-ph, physics:quant-ph]* (2019). DOI: `10.23638/LMCS-15(3:15)2019`. arXiv: `1701.04732 [math-ph, physics:quant-ph]`.

[Kv19] Aleks Kissinger and John van de Wetering. "PyZX: Large Scale Automated Diagrammatic Reasoning". In: *arXiv:1904.04735 [quant-ph]* (Apr. 2019). arXiv: `1904.04735 [quant-ph]`.

[Kv20] Aleks Kissinger and John van de Wetering. "Reducing T-count with the ZX-calculus". In: *Phys. Rev. A* 102.2 (Aug. 2020), p. 022406. DOI: `10.1103/PhysRevA.102.022406`. arXiv: `1903.10477`.

[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". In: *arXiv:quant-ph/9511026* (Nov. 1995). arXiv: `quant-ph/9511026`.

[Kit03] A. Yu. Kitaev. "Fault-Tolerant Quantum Computation by Anyons". In: *Annals of Physics* 303.1 (Jan. 2003), pp. 2–30. DOI: `10.1016/S0003-4916(02)00018-0`.

[Knu68] Donald E. Knuth. "Semantics of Context-Free Languages". In: *In Mathematical Systems Theory*. 1968, pp. 127–145.

[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. 1958), pp. 154–170. DOI: 10.1080/00029890.1958.11989160.

[Lam59] Joachim Lambek. “Contributions to a Mathematical Analysis of the English Verb-phrase”. In: *Can. J. Linguist.* 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 Ryszard Wójcicki, Petr Hájek, David Makinson, Daniele Mundici, Krister Segerberg, Alasdair Urquhart, Hans Jürgen Ohlbach, and Uwe Reyle. Vol. 5. 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.

[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.

[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.

[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.

[Law70a] F. William Lawvere. "Equality in Hyperdoctrines and the Comprehension Schema as an Ad-Joint Functor". In: 1970.

[Law70b] 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.

[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. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 1–13.

[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. 1992. DOI: 10.1515/9783110870299.14.

[Law96] F. William Lawvere. "Unity and Identity of Opposites in Calculus and Physics". In: *Appl Categor Struct* 4.2-3 (1996), pp. 167–174. DOI: 10.1007/BF00122250.

[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.

[LMR13] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. “Quantum Algorithms for Supervised and Unsupervised Machine Learning”. In: *arXiv:1307.0411 [quant-ph]* (Nov. 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`. arXiv: `1307.0401`.

[LB02] Edward Loper and Steven Bird. “NLTK: The Natural Language Toolkit”. In: *arXiv:cs/0205028* (May 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”. In: *arXiv:2102.12846 [quant-ph]* (Feb. 2021). arXiv: `2102.12846 [quant-ph]`.

[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.

[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+16] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. “The Theory of Variational Hybrid Quantum-Classical Algorithms”. In: *New J. Phys.* 18.2 (Feb. 2016), p. 023023. DOI: `10.1088/1367-2630/18/2/023023`.

[McC+18] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, Ryan Babbush, and Hartmut Neven. “Barren Plateaus in Quantum Neural Network Training Landscapes”. In: *Nat Commun* 9.1 (Dec. 2018), p. 4812. DOI: `10.1038/s41467-018-07090-4`. arXiv: `1803.11173`.

[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, Alexis Toumi, Giovanni de Felice, and Bob Coecke. "Grammar-Aware Question-Answering on Quantum Computers". In: *ArXiv e-prints* (2020). arXiv: `2012.03756`.

[Mei+20b] 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`. arXiv: `2005.04147`.

[MZ16] Paul-André Melliès and Noam Zeilberger. "A Bifibrational Reconstruction of Lawvere's Presheaf Hyperdoctrine". In: *arXiv:1601.06098 [cs, math]* (Aug. 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". In: *arXiv:2111.06741 [quant-ph]* (Nov. 2021). arXiv: `2111.06741 [quant-ph]`.

[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.* https://ncatlab.org/nlab/show/concept+with+an+attitude.

[Pat17] Evan Patterson. "Knowledge Representation in Bicategories of Relations". In: *arXiv:1706.00526 [cs, math]* (June 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". In: *arXiv:2101.12046 [cs]* (Jan. 2021). DOI: 10.4204/EPTCS.333.4. arXiv: 2101.12046 [cs].

[Ped+19] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. "Leveraging Secondary Storage to Simulate Deep 54-Qubit Sycamore Circuits". In: *arXiv:1910.09534 [quant-ph]* (Oct. 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.

[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: *Nat Commun* 5.1 (July 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: *Phys. Rev. X* 11.4 (Oct. 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.

[Pre18] John Preskill. “Quantum Computing in the NISQ Era and Beyond”. In: *Quantum* 2 (Aug. 2018), p. 79. DOI: 10.22331/q-2018-08-06-79.

[RML14] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. “Quantum Support Vector Machine for Big Data Classification”. In: *Phys. Rev. Lett.* 113.13 (Sept. 2014), p. 130503. DOI: 10.1103/PhysRevLett.113.130503.

[RV19] David Reutter and Jamie Vicary. “High-Level Methods for Homotopy Construction in Associative \$n\$-Categories”. In: *arXiv:1902.03831 [math]* (Feb. 2019). arXiv: 1902.03831 [math].

[RV16] Emily Riehl and Dominic Verity. “Infinity Category Theory from Scratch”. In: *arXiv preprint arXiv:1608.05314* (2016). arXiv: 1608.05314.

[Ril18] Mitchell Riley. “Categories of Optics”. In: *arXiv:1809.00738 [math]* (Sept. 2018). arXiv: 1809.00738 [math].

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (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.

[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.

[Sch21] Maria Schuld. “Quantum Machine Learning Models Are Kernel Methods”. In: *arXiv:2101.11020 [quant-ph, stat]* (Jan. 2021). arXiv: 2101.11020 [quant-ph, stat].

[SK19] Maria Schuld and Nathan Killoran. “Quantum Machine Learning in Feature Hilbert Spaces”. In: *Phys. Rev. Lett.* 122.4 (Feb. 2019), p. 040504. DOI: 10.1103/PhysRevLett.122.040504. arXiv: 1803.07128.

[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.

[SV06] Peter Selinger and Benoit Valiron. "A Lambda Calculus for Quantum Computation with Classical Control". In: *Math. Struct. Comp. Sci.* 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 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.

[Sho96] Peter W Shor. "Fault-Tolerant Quantum Computation". In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 56–65.

[Sho94] P.W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 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.

[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". In: *arXiv:2003.10611 [quant-ph]* (Mar. 2020). arXiv: 2003.10611 [quant-ph].

[Smo87] P. Smolensky. "Connectionist AI, Symbolic AI, and the Brain". In: *Artif Intell Rev* 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. 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.

[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.

[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. 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. 2010), pp. 141–188. DOI: 10.1613/jair.2934.

[VT04] 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". In: *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762 [cs].

[VN21] Irene Vicente Nieto. *Towards Machine Translation with Quantum Computers*. 2021.

[Wea55] Warren Weaver. “Translation”. In: *Machine translation of languages* 14.15-23 (1955), p. 10.

[WBL12] Nathan Wiebe, Daniel Braun, and Seth Lloyd. “Quantum Algorithm for Data Fitting”. In: *Phys. Rev. Lett.* 109.5 (Aug. 2012), p. 050505. DOI: 10.1103/PhysRevLett.109.050505.

[Wie+19] Nathan Wiebe, Alex Bocharov, Paul Smolensky, Matthias Troyer, and Krysta M. Svore. “Quantum Language Processing”. In: *arXiv:1902.05162 [quant-ph]* (Feb. 2019). arXiv: 1902.05162 [quant-ph].

[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: *Phys. Rev. Lett.* 127.18 (Oct. 2021), p. 180501. DOI: 10.1103/PhysRevLett.127.180501.

[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. 2021), pp. 1–12. DOI: 10.1145/3458817.3487399. arXiv: 2110.14502.

[ZC16] William Zeng and Bob Coecke. “Quantum Algorithms for Compositional Natural Language Processing”. In: *Electronic Proceedings in Theoretical Computer Science* 221 (Aug. 2016), pp. 67–75. DOI: 10.4204/EPTCS.221.8. arXiv: 1608.01406.

[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. 2020), pp. 1460–1463. DOI: 10.1126/science.abe8770. arXiv: 2012.01625.

[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. 2021). DOI: 10.1016/j.scib.2021.10.017.