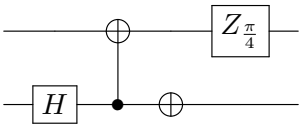
ZXCalculusForCAP – THE CATEGORY OF ZX-DIAGRAMS AS A CATEGORICAL TOWER

MOHAMED BARAKAT AND FABIAN ZICKGRAF

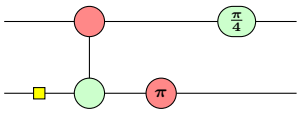
The category of ZX-diagrams is a diagrammatic realization of the PROP generated by the 2-dimensional Hilbert space (qubit space) equipped with two special Frobenius algebra structures stemming from two mutually unbiased bases, namely the computational basis (the eigenvectors of the Pauli-Z matrix) and the Hadamard basis (the eigenvectors of the Pauli-X matrix). In this extended abstract we explain how to implement (an ambient category of) the category of ZX-diagrams as a categorical tower. The outcome lies in a constructive sense in the doctrine of rigid symmetric monoidal categories (or even †-compact categories). The resulting foundational functional (quantum) programming language is the internal language of that doctrine. As such, this language allows the definition of function types, free variables, contexts, abstraction, application, and hence higher order functions.

1. ZX-DIAGRAMS

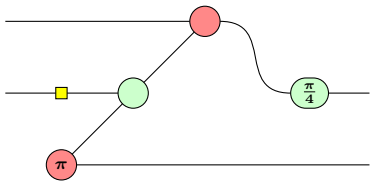
For example, the quantum circuit



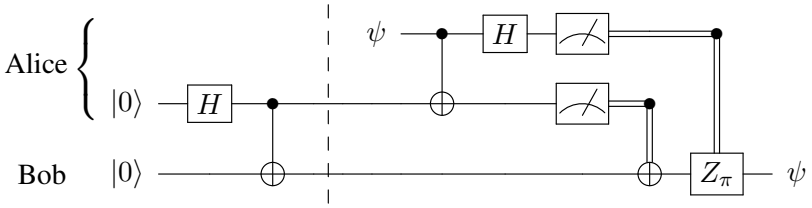
can be written as a ZX-diagram as follows:



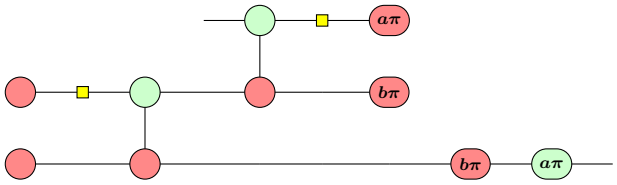
While we have retained the structure of the quantum circuit, most of this structure is not a formal part of the ZX-diagram. In particular, the "flow" from left to right and the mapping of gates to qubits is not encoded in the ZX-diagram and we could, for example, also draw the diagram as follows:



One can also encode the initialization of qubits and measurements in ZX-diagrams. As an example for this, consider the circuit describing quantum teleportation:



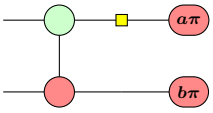
This circuit can be translated to the following parameterized ZX-diagram:



The parameters $a, b \in \{0, 1\}$ link the outcomes of the measurements to the application of the NOT gate and/or the phase shift gate. We can simplify this diagram using the rules of the ZX-calculus and get

which corresponds to an identity. This shows that the input quantum state is indeed teleported without modification from Alice to Bob.

Note that for $a = b = 0$, the subdiagram



2010 *Mathematics Subject Classification.* 68W30,

Key words and phrases. ZX-calculus, CompilerForCAP, category constructors, categorical towers,

This work is a contribution to the Qompiler Project – Supported by the German Federal Ministry for Economic Affairs and Climate Action.

simplifies to

which is known as the *Bell effect*.

In the next section we describe the concept of categorical towers which we will use in Section 3 in order to model the ambient category of ZX-diagrams.

2. Categorical towers

By a **doctrine** $\mathscr{D}$ we mean a 2-category of structured categories. Examples are the category $\mathcal{C}\mathrm{at}$ of locally small categories, functors, and natural transformations, or the category $\mathcal{R}\mathrm{ex}$ of categories with finite colimits, cocontinuous (= right exact) functors, and natural transformations. A **category constructor** is a 2-functor $\mathscr{L} : \mathcal{D} \to \mathcal{E}$ of doctrines. Typically $\mathcal{E}$ is a doctrine of categories with *extra* structure. The doctrine $\mathcal{I}\mathrm{nit}$ consisting of the empty category $\emptyset$ enables us to view a specific category $\mathbf{E}$ in a doctrine $\mathcal{E}$ as a special 2-functor $\mathcal{I}\mathrm{nit} \to \mathcal{E}, \emptyset \mapsto \mathbf{E}$. In fact, $\mathcal{I}\mathrm{nit}$ is naturally a 1-category which we view as a 2-category with identity cells. Another such degenerate example used below is the 1-category $\mathcal{Q}\mathrm{uiv}$ of quivers. We now list the category constructors we need for building the **categorical tower** modeling the ambient category of ZX-diagrams.

- The 2-functor $\mathscr{L} = \texttt{SkeletalFinSets} : \mathcal{I}\mathrm{nit} \to \mathcal{C}\mathrm{at}$ takes the empty category $\emptyset$ to the skeletal category $\mathbf{SkeletalFinSets}$ of finite von Neumann cardinals $n := \{0, \ldots, n-1\}$.
- The 2-functor $\mathscr{L} = \texttt{PathCategory} : \mathcal{Q}\mathrm{uiv} \to \mathcal{C}\mathrm{at}$, which maps a quiver $\mathbf{Q}$ to its category of paths (freely generated on $\mathbf{Q}$).
- The 2-functor $\mathscr{L} = \texttt{PreSheaves} : \mathcal{F}\mathrm{in}\mathcal{C}\mathrm{at} \to \mathcal{E}\mathrm{t}$ from the category of $\texttt{SkeletalFinSets}$-enriched finite categories to the doctrine $\mathcal{E}\mathrm{t}$ of elementary toposes, which maps a finite category $\mathbf{C} \in \mathcal{F}\mathrm{in}\mathcal{C}\mathrm{at}$ to the category $\texttt{PreSheaves}(\mathbf{C}, \mathbf{SkeletalFinSets})$ of (finitely presented) presheaves on $\mathbf{C}$.
- The 2-functor $\mathscr{L} = \texttt{SliceCategory} : \mathcal{C}\mathrm{at}_0 \to \mathcal{C}\mathrm{at}$ from the doctrine of categories each with a distinguished object (and functors sending the distinguished object of their source to that of their target category), which maps a pair $(\mathbf{C}, A \in \mathbf{C})$ to the slice category $\mathbf{C}/A$. It is well-known that $\texttt{Slice}$ induces a 2-functor between $\mathcal{E}\mathrm{t}_0 \to \mathcal{E}\mathrm{t}$.
- The 2-functor $\mathscr{L} = \texttt{CategoryOfCospans} : \mathcal{C}\mathrm{ocart} \to \mathcal{R}\mathrm{ig}\mathcal{S}\mathrm{ym}\mathcal{M}\mathrm{on}\mathcal{C}\mathrm{at}$ from the doctrine of cocartesian monoidal categories to the doctrine of rigid symmetric monoidal categories, which maps a cocartesian category to its category of cospans on the same set of objects but where a morphism $S \to T$ in $\texttt{CategoryOfCospans}(\mathbf{C})$ is a cospan $S \to A \leftarrow T$ in $\mathbf{C}$.
- The 2-functor $\mathscr{L} = \texttt{Subcategory} : \mathcal{C}\mathrm{at}_{\subseteq} \to \mathcal{C}\mathrm{at}$ from the doctrine of categories each with a distinguished subcategory (and functors sending the distinguished subcategory of their source to that of their target category), which maps a pair $(\mathbf{C}, \mathbf{D} \subseteq \mathbf{C})$ to $\mathbf{D} \in \mathcal{C}\mathrm{at}$. In our algorithmic context the datum $\mathbf{D} \subseteq \mathbf{C}$ is specified by a morphism-membership function.
- Various forgetful 2-functors $\mathscr{U} : \mathcal{E} \to \mathcal{D}$ that forget the extra structure of the categories in the doctrine $\mathcal{E}$.

Reinterpretations of categorical towers allow to formally view categorical towers as primitive category constructors with possibly simplified data structures for objects and morphisms. The concept was developed in the context of `CompilerForCAP` as the central mechanism for generating efficient primitive implementations from categorical towers in CAP. For details see Appendix A.

Each category constructor used for building a categorical tower is self-contained. Hence, the organization of categorical towers introduces a high degree of modularity. This has the following advantages:

- *Reusability*: The category constructors used to build one categorical tower can be reused for building categorical towers in other contexts.
- *Separation of concerns*: Every category constructor in a categorical tower can focus on a single concept, simplifying the definitions.
- *Verifiability*: The category constructors used to build a categorical tower can be verified independently of each other. Each category constructor has a limited scope and is hence relatively simple to comprehend and verify.
- *Emergence*: Simple and natural constructions at each level of the tower can lead to highly complex structures of the tower as a whole.

On a computer, all these advantages lead to higher quality code:

- reusability of code leads to less code overall and better code coverage,
- the separation of concerns makes it possible to focus on one aspect at a time when implementing the various category constructors,
- better verifiability makes it easier to check code for correctness, and
- convoluted algorithms for the tower as a whole emerge from simple and natural algorithms at each level of the tower.

In the next section we describe a categorical tower modeling the ambient category of ZX-diagrams.

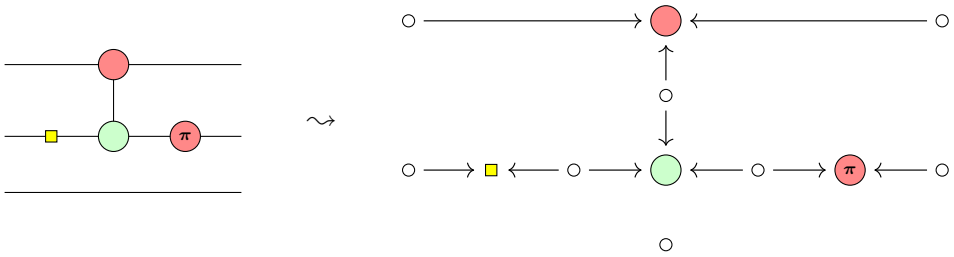
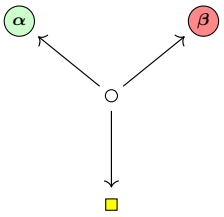


FIGURE 1. Encoding of a ZX-diagram as a quiver decorated over $S_{\mathbf{ZX}}$

3. THE CATEGORY OF ZX-DIAGRAMS

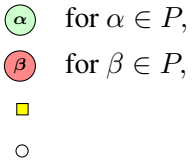
The ambient category $\mathbf{ZX} = \mathbf{ZX}_P$ of ZX-diagrams (without simplification rules) over a finite set $P \subset [0, 2\pi)$ of angles[1] can be described as a categorical tower of the category constructors in Section 2, using the finite quiver

$S_{\mathbf{ZX}} :=$ 

for the slice construction, where α and β run through all phases in P. We follow the construction in [Cic18] but refine some details which are needed for making the tower fully algorithmic, as we will see in Remark 3.3. The construction is given as follows:

level 5:	$\texttt{Subcategory}(\mathbf{O}, \mathbf{S} \subset \mathbf{O}) \simeq \mathbf{ZX}$	$\mathcal{R}\mathrm{ig}\mathcal{S}\mathrm{ym}\mathcal{M}\mathrm{on}\mathcal{C}\mathrm{at}$
level 4:	$\texttt{CategoryOfCospans}(\mathbf{D}) =: \mathbf{O}$	$\mathcal{R}\mathrm{ig}\mathcal{S}\mathrm{ym}\mathcal{M}\mathrm{on}\mathcal{C}\mathrm{at}$
level 3:	$\texttt{SliceCategory}(\mathbf{FinQuiv}, S_{\mathbf{ZX}}) =: \mathbf{D}$	$\mathcal{E}\mathrm{t} \rightsquigarrow \mathcal{C}\mathrm{ocart}$
level 2:	$\texttt{PreSheaves}(\mathbf{C}, \mathbf{SkeletalFinSets}) \simeq \mathbf{FinQuiv}$	$\mathcal{E}\mathrm{t}$
level 1:	$\texttt{PathCategory}(\mathbf{Q}) =: \mathbf{C}$ $\quad$ $\texttt{SkeletalFinSets}()$	$\mathcal{F}\mathrm{in}\mathcal{C}\mathrm{at}$, $\mathcal{E}\mathrm{t}$
level 0:	$\mathfrak{V} \rightrightarrows \mathfrak{A} =: \mathbf{Q}$	$\mathcal{Q}\mathrm{uiv}$

Remark 3.1 (Encoding ZX-diagrams as decorated quivers). Let $S_{\mathbf{ZX}}$ be given as above, that is, $S_{\mathbf{ZX}}$ is the quiver with vertices



where the so-called *neutral vertex* $\circ$ connects to all other vertices by an arrow. We choose $S_{\mathbf{ZX}}$ as the base quiver for forming the slice category $\mathbf{FinQuiv}/S_{\mathbf{ZX}}$, that is, the category (elementary topos) of finite quivers decorated by $S_{\mathbf{ZX}}$. Now, we want to encode a ZX-diagram D as a quiver Q_D decorated by $S_{\mathbf{ZX}}$. An example showing how we do this can be seen in Figure 1. Specifically, we use following rules:

- Boundary nodes of D are encoded as neutrally decorated vertices of Q_D.
- Inner nodes of D are encoded as vertices of Q_D decorated in the obvious way.
- An edge connecting a boundary node and an inner node of D is encoded as an arrow of Q_D pointing away from the corresponding neutrally decorated vertex.
- An edge connecting two inner nodes of D is encoded as two arrows of Q_D pointing away from a neutrally decorated vertex which is newly added.
- An edge connecting two boundary nodes of D is handled in a special way: The edge together with the two boundary nodes is encoded as a single neutrally decorated vertex of Q_D.

[1] Any quantum circuit only involves a finite set of phases $e^{i\alpha}$. Hence, one only needs the finite set P of angles α appearing in the quantum circuits of the intended application.

Finally, we also associate a decorated quiver Q_n to natural numbers n, whose role will become clear in the next construction: We define Q_n as a quiver with n neutrally decorated vertices, which can be visualized as follows:

$$\begin{array}{c}\circ\\ \vdots n\\ \circ\end{array}$$

With this, we can model the ambient category of ZX-diagrams as a category of cospans:

Remark 3.2 (The ambient category of ZX-diagrams as a categorical tower). We use the notation of Remark 3.1. We define an embedding $\mathcal{R}$ of $\mathbf{ZX}$ into $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$ as follows:

$$\begin{aligned}\mathcal{R} : \mathbf{ZX} &\hookrightarrow \texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}}),\\ n &\mapsto Q_n,\\ (m \xrightarrow{D} n) &\mapsto (Q_m \to Q_D \leftarrow Q_n),\end{aligned}$$

where the morphisms of the cospan map the i-th vertex of Q_m (respectively Q_n) to the vertex of Q_D coming from the i-th input (respectively output) of D. In particular, the ZX-diagram D is now realized by the *morphism* $Q_m \to Q_D \leftarrow Q_n$.

With this, $\mathcal{R}$ defines an isomorphism to a subcategory $\mathbf{S}$ of $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$. Hence, we can model $\mathbf{ZX}$ as a reinterpretation of the subcategory $\mathbf{S}$ of the categorical tower $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$ via $\mathcal{R}$. The subcategory $\mathbf{S}$ can be characterized as follows: Its only objects are Q_n for natural numbers n. Its morphisms are only those cospans where the central object is given by a decorated quiver whose vertices fulfill the degree restrictions in the definition of ZX-diagrams, that is:

- a vertex decorated as "Hadamard" must have degree 2,
- the vertices representing inputs or outputs must have degree 1, with the degenerate case that a vertex might represent two boundary nodes at the same time and has degree 0.

At first, modeling the category of ZX-diagrams as a categorical tower seems to be much more cumbersome than just working with the category of ZX-diagrams as in the original definition. This is true as long as one stays on a purely visual level. However, as soon as one wants to implement the category on the computer, one quickly notices that defining a suitable data structure for morphisms and formalizing the notion of "glueing" ZX-diagrams is difficult. When working with the categorical tower, though, the data structures and formalizations arise quite naturally. This is a perfect example of *emergence*.

More precisely, the complicated part of a direct hand-written implementation of the category of ZX-diagrams would be the composition of morphisms, i.e., the glueing of two ZX-diagrams with equal number of outputs and inputs. This task is delegated by the categorical tower through composition in the category of cospans all the way down to the computation of pushouts and hence the computation of coequalizers in $\mathbf{SkeletalFinSets}$. The latter is nothing but the computation of reflexive transitive closures of binary relations on finite sets.

Moreover, the relevant fragment of the theory of presheaf categories, which we use to model quivers, and slice categories is well developed, so we can actually build on existing theory for the mathematics and on computer implementations which pre-existed in our software framework for completely different applications. We will see this in Remark 3.5, where we construct a rigid symmetric monoidal structure on the category of ZX-diagrams using the tower $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$. This is a nice example of *reusability*.

Remark 3.3. We deviate in various points from [Cic18], which can be seen by comparing the decorating quiver used there to our decorating quiver:

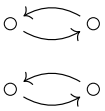
We look at the differences:

- In the decorating quiver on the left, we see an additional kind of node: the so-called *diamond node*. This is due to [Cic18] using a presentation of the ZX-calculus accounting for so-called *scalar factors*, see [DKPvdW20, Remark 2.2]. We could easily include additional kinds of nodes if needed by simply adding them to our decorating quiver.
- For every arrow in the decorating quiver on the left there also exists a reversed arrow. Probably the intention in [Cic18] is to model undirected graphs as directed graphs by encoding every undirected edge by two directed arrows covering both possible directions. However, this is not actually done in the paper and it seems like the problem is not properly addressed. We avoid this problem by simply normalizing the direction of arrows to always point away from neutrally decorated vertices.
- In the decorating quiver on the left, the neutral vertex has a self-loop. To see the significance of this loop, recall the following special case in the construction of the directed quiver Q_D corresponding to a ZX-diagram D above:

An edge connecting two boundary nodes of D is handled in a special way: The edge together with the two boundary nodes is encoded as a single neutrally decorated vertex of Q_D.

If we would endow the neutral vertex with a self-loop, we could avoid this special case and instead encode an edge connecting two boundary nodes in D by an actual arrow in Q_D. For example, we could encode the ZX-diagram D given as

as a decorated quiver Q'_D as follows:



However, with this, $\mathcal{R}$ would not map identities of $\mathbf{ZX}$ to identities of $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$ anymore: The diagram D above is just the diagram defining id_2 in $\mathbf{ZX}$. However, Q'_D is not isomorphic to

which is the central object of the cospan defining the identity on $\mathcal{R}(2)$ in $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$:

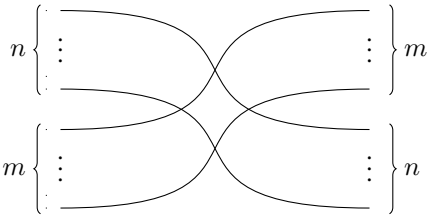
$$Q_2 \xrightarrow{\mathrm{id}_{Q_2}} Q_2 \xleftarrow{\mathrm{id}_{Q_2}} Q_2.$$

To solve this problem, [Cic18] passes to a quotient of $\texttt{CategoryOfCospans}(\mathbf{FinQuiv}/S_{\mathbf{ZX}})$ where the images of the identity morphisms of $\mathbf{ZX}$ are identified with the corresponding identity morphisms of the tower. However, working with such a quotient in a computer implementation is cumbersome. Hence, we prefer to solve the problem right in the encoding process.

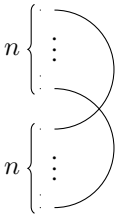
Remark 3.4. The category $\mathbb{ZX}$ of ZX-diagrams can be constructed by adding one more category constructor to the above tower, namely the quotient category of $\mathbf{ZX}$ modulo the simplification rules of the ZX calculus. Among these rules are the spider fusion rules which state each of the two Frobenius algebra structures are special in the sense that the composition of their comultiplication with their multiplication is the identity morphism. We know of no software tool to *efficiently*[2] decide the equivalence of two ZX non-Clifford diagrams modulo the simplification rules of the ZX calculus. One could of course use a tool like `pyzx` [KvdW20] (or its graphical interface `zxlive`) to manually explore transformations of one ZX-diagram (morphism) into another.

Remark 3.5. The skeletal category $\mathbf{ZX}$ lies in the doctrine $\mathcal{R}\mathrm{ig}\mathcal{S}\mathrm{ym}\mathcal{M}\mathrm{on}\mathcal{C}\mathrm{at}$, which is bi-equivalent to the doctrine of compact closed categories. Furthermore, each object in $\mathbf{ZX}$ is self-dual, i.e., $n^* = n$, and the duality on morphisms interchanges inputs and outputs. The tower implies the following structure for the braiding, evaluation, coevaluation, and the coherence conditions of dualizable objects:

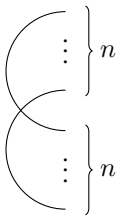
Braiding:



Evaluation:

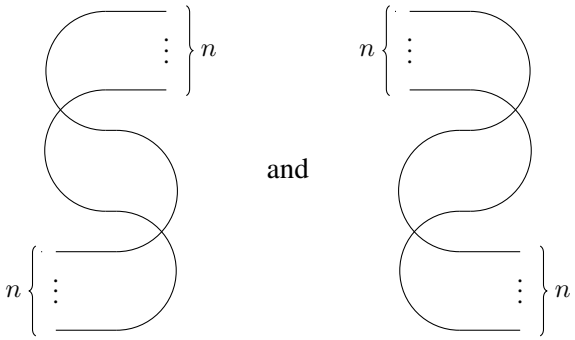


Coevaluation:



Yanking (the following diagrams yield the identity morphism on n):

[2] Constructing and comparing the corresponding matrices is not an efficient method.



The internal language of a category in $\mathcal{R}\mathrm{ig}\mathcal{S}\mathrm{ym}\mathcal{M}\mathrm{on}\mathcal{C}\mathrm{at}$ yields a typed (quantum) lambda calculus with free variables, contexts, abstraction, applications, and function types. In particular, $\mathbf{ZX}$ can be used to model a foundational **functional quantum programming language**.

Remark 3.6. In order for $\mathbf{ZX}$ to be a †-compact category, the set P of angles must be closed under negation $P = -P$. The negation induces an involution on the quiver $S_{\mathbf{ZX}}$ which we denote by $\overline{\cdot}$, as it coincides with complex conjugation of corresponding phases $e^{i\alpha} \mapsto \overline{e^{i\alpha}} = e^{-i\alpha}$. This conjugation induces a covariant involution on the slice category $\mathbf{D}$ of decorated quivers, on the cospan category $\mathbf{O}$ of open graphs, and finally on the subcategory $\mathbf{S} \simeq \mathbf{ZX}$ of ZX-diagrams. The † can now be defined as the composition of the duality (contravariant) and the conjugation (covariant) endofunctors, turning $\mathbf{ZX}$ into a †-compact category. Since each object in $\mathbf{ZX}$ is self-dualizing, the † structure allows the ZX-diagram of every quantum circuit to be inverted. This is relevant for "partial uncomputations".

4. ZXCalculusForCAP

The category of ZX-diagrams is in particular a strict closed monoidal category, which defines a so-called *typed generalized lambda calculus*, yielding a foundational functional programming language. By unfolding the definitions, one can check that in cases where the ZX-diagrams correspond to quantum circuits, we get the following semantics:

the typed generalized lambda calculus defined by ZX	**interpretation as a quantum programming language**
a type T	T qubits
a lambda term of type T in a context $(x : T)$	a quantum function on T qubits
application of lambda terms	application of quantum functions

Hence, in cases where the ZX-diagrams correspond to quantum circuits, we indeed get the semantics expected for a quantum programming language. Existing quantum circuits can be imported as so-called *library functions* into the calculus. For more details see [Zic24b, Sections 2.11, 5.4, 5.5].

Remark 4.1 (A universal functional quantum programming language). Summing up, we have actually created a functional quantum programming language:

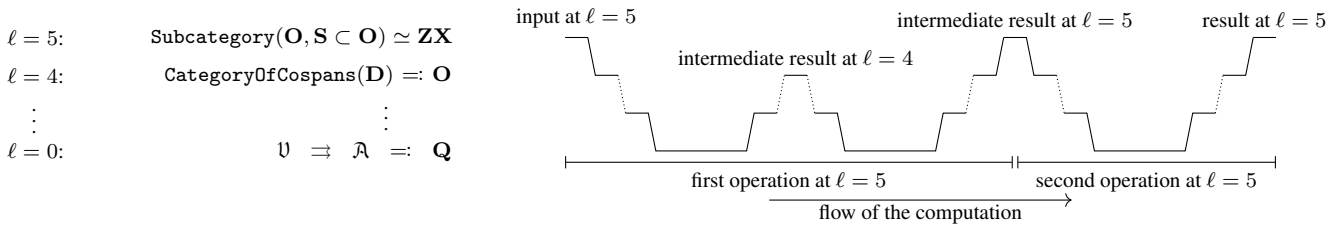
We can import existing quantum circuits as ZX-diagrams and view them as constants, which we call *library functions*. We can then write typed programs using these library functions, (free) variables, as well as abstraction and application of (higher-order) functions.

Such a program is represented by a ZX-diagram. If we can extract a quantum circuit from the diagram, we can hence execute the program on a quantum computer. Moreover, the semantics of lambda terms and application of lambda terms in our programs actually corresponds to quantum functions and application of quantum functions. Hence, the quantum circuit extracted from the ZX-diagram representing our program will actually compute what we formulated abstractly in our program. In particular, we actually get a semantics of higher-order quantum functions, as one would expect in a functional quantum programming language.

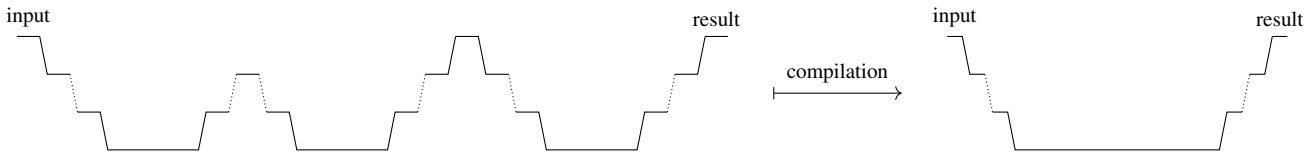
Assuming that we include the ZX-diagrams corresponding to a universal set of quantum gates, we can create every quantum circuit in the programming language. Hence, every quantum function can be modeled in the programming language, that is, the programming language is a *universal* functional quantum programming language.

5. CompilerForCAP: Compiling the tower and computational performance

Structuring the software as a tower of category constructors naturally comes with a measurable performance overhead. We start by describing three sources of overhead which to some extent could be solved by a generic compiler, for example Julia's JIT compiler. The first source of overhead is the excessive amount of superfluous boxing and unboxing of data structures occurring during the computations, especially in an interpreted language like GAP. For example, a simple computation consisting of two operations at the topmost level of the tower in Section 2, where the first operation itself consists of two operations at level $\ell = 4$, can be visualized as follows:



Passing one level down or up during the computation means unboxing or boxing a corresponding data structure. The picture illustrates that data which is boxed to form an intermediate result will again be immediately unboxed by the next operation at the same level. A generic compiler can rewrite code so that the given input objects and morphisms only have to be unboxed once at the beginning of a computation and boxed again once the computation is finished:



The second source of overhead is that a high-level computation will often trigger repeated evaluation of low-level algorithms with the same input, that is, will compute the same thing multiple times. To avoid this, one typically uses caching, which is, however, detrimental to parallelization. A generic compiler can use common subexpression elimination and hoisting to solve this problem without caching.

The third source of overhead comes from computing intermediate lower-level results that end up not affecting the final high-level result. This can be addressed using lazy data structures, whereas a generic compiler can more elegantly just eliminate dead code.

However, to simplify the categorical code and improve the performance even further we also need support for

(1) the reinterpretation (cf. Section 2), i.e., changing the data structures of objects and morphisms at the topmost level to completely avoid the repeated (un)boxing, even at the beginning and the end, so that the compiled code contains no references to the tower anymore;
(2) full control over the compilation process, for example to provide doctrine-specific logical rewriting rules at any chosen level of the categorical tower;
(3) the flexibility to compile against any chosen level ℓ of the categorical tower generating readable categorical code, for example to detect missing advantageous logical rewriting rules;
(4) transpiling the compiled code generated in (3) for level $\ell = 1$ into native code (at level $\ell = 0$) in various programming languages or computer algebra systems (some of which support parallel computing), which can be further compiled to (parallelized) machine code, e.g., by JULIA's JIT compiler.

`CompilerForCAP` supports advanced compiler optimizations like common subexpression elimination, loop-invariant code motion, constant folding and propagation, dead-code elimination, and generalized loop fusion that help improve the quality of the compiled code significantly. Experiments have shown that `CompilerForCAP` can apply these optimizations more widely and effectively than other general purpose compilers could do, even more than those for functional programming languages like Haskell's compiler GHC. This is due to the fact that modern CAP code, as code implementing category-theoretic constructions, follows stricter rules and uses interfaces with stricter mathematical specifications than general code. So the combination of the domain-specific `CompilerForCAP` and JULIA's JIT-compiler produces far better compiled code than the latter alone could be able to produce.

Finally, using peephole optimization via so-called logic templates one can further improve the quality of the compiled code by applying mathematical knowledge.

The above could only be achieved by a *category-theory-aware compiler* taking advantage of the strict hierarchy of data types provided by category theory. Such a compiler is implemented in our package `CompilerForCAP` [Zic24a]. Additionally, we are using `CompilerForCAP` equipped with its logical rewriting rules for code verification or, "dually", as a yet simple-minded proof assistant. On the one hand, we can regard a low-level translation of a high-level algorithm as adequately verified once we succeed in producing it using `CompilerForCAP`. On the other hand, `CompilerForCAP` helps us find such translations if not yet known. In any case, the compiled low-level code typically reaches a level of complexity that would make a direct implementation, let alone a manual verification, error-prone and practically impossible.

For example, when compiling the above categorical tower, the tensor product of objects m, n in $\mathbf{ZX}$ compiles to the sum $m + n$ of the underlying integers. However the compiled code for the composition of two morphisms, which corresponds to the glueing of two ZX-diagrams with equal number of input and output qubits would have been hard to write by hand.

APPENDIX A. REINTERPRETATIONS OF CATEGORICAL TOWERS

Definition A.1 (Reinterpretations of categorical towers)**.** Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors with the same input specification, where $\mathcal{T}$ is a categorical tower and $\mathcal{C}$ is a primitive category constructor. If there exists an isomorphism[3] $\mathcal{R}_I : \mathcal{T}(I) \to \mathcal{C}(I)$ for all inputs I, we call

- $\mathcal{C}$ a *reinterpretation* of the categorical tower $\mathcal{T}$ via the functor $\mathcal{R}$, and
- $\mathcal{T}$ a *model* of $\mathcal{C}$.

We usually write $\mathcal{M}_I : \mathcal{C}(I) \to \mathcal{T}(I)$ for the inverse of $\mathcal{R}_I$. In contexts with additional structures like (pre)additive or monoidal structures, we assume that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the additional structures.

To take full advantage of categorical towers, we would actually like to *define* primitive category constructors as reinterpretations of categorical towers. For this, we note that in the context of a reinterpretation, the categorical structure of the primitive category is already uniquely determined by the categorical structure of the categorical tower:

Remark A.2. Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors as in Definition A.1 and let I be some input. Set $\mathbf{T} := \mathcal{T}(I)$, $\mathbf{C} := \mathcal{C}(I)$, $\mathcal{R} := \mathcal{R}_I$, and $\mathcal{M} := \mathcal{M}_I$. For morphisms f and g in $\mathbf{C}$, we have

$$f \cdot g = \mathcal{R}(\mathcal{M}(f \cdot g)) = \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)). \tag{1}$$

Moreover, for an object A in $\mathbf{C}$, we have

$$\mathrm{id}_A = \mathcal{R}(\mathcal{M}(\mathrm{id}_A)) = \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)}). \tag{2}$$

This shows that the composition and the identity morphisms of $\mathbf{C}$ are uniquely determined by the composition and the identity morphisms of $\mathbf{T}$ together with the values of $\mathcal{R}$ and $\mathcal{M}$ on objects and morphisms.

In contexts with additional structures like (pre)additive or monoidal structures, we have assumed that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the additional structures. Hence, also the additional structures are uniquely determined by the structures of $\mathbf{T}$ together with the values of $\mathcal{R}$ and $\mathcal{M}$ on objects and morphisms.

This motivates the following structure-transport construction:

Remark A.3 (Defining categories via reinterpretations). Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors as in Definition A.1 and let I be some input. Set $\mathbf{T} := \mathcal{T}(I)$, $\mathbf{C} := \mathcal{C}(I)$, $\mathcal{R} := \mathcal{R}_I$, and $\mathcal{M} := \mathcal{M}_I$. We ignore the existing categorical structure of $\mathbf{C}$, that is, composition and identity morphisms, as well as the fact that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the composition and the identities. Instead, we define a categorical structure on $\mathbf{C}$ via

$$f \cdot g := \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \qquad \text{and} \qquad \mathrm{id}_A := \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)}).$$

With this definition, $\mathcal{R}$ and $\mathcal{M}$ are automatically compatible with the composition and the identities, and thus isomorphisms of categories again. Hence, $\mathbf{C}$ is again a reinterpretation of $\mathbf{T}$ via $\mathcal{R}$ and $\mathcal{M}$. Additional structures like (pre)additive or monoidal structures are defined in a similar way via $\mathcal{R}$, $\mathcal{M}$, and the structure of $\mathbf{T}$. Again, $\mathcal{R}$ and $\mathcal{M}$ are automatically compatible with the additional structures.

This structure-transport allows us to *define* primitive category constructors as reinterpretations of categorical towers. Remark A.2 ensures that this construction is compatible with any existing categorical structure.

Remark A.4. Remark A.3 would not work if $\mathcal{R}$ and $\mathcal{M}$ would not be isomorphisms but merely equivalences of categories: In this case, setting

$$f \cdot g := \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \qquad \text{and} \qquad \mathrm{id}_A := \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)})$$

would not define morphisms with the required source and targets because in general

$$\mathcal{R}(\mathcal{M}(A)) \not\equiv A$$

if $\mathcal{R}$ and $\mathcal{M}$ are mere equivalences. In an attempt to account for the natural isomorphisms coming with equivalences one would let $\eta : \mathcal{M} \cdot \mathcal{R} \Rightarrow \mathrm{Id}_{\mathbf{C}}$ be the natural isomorphism in $\mathbf{C}$. We consider an equation analogous to equation Equation (1): For morphisms $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$, we have

$$f \cdot g = \eta_A^{-1} \cdot \mathcal{R}(\mathcal{M}(f \cdot g)) \cdot \eta_C = \eta_A^{-1} \cdot \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \cdot \eta_C.$$

Note that the right-hand side of this equations still uses the composition in $\mathbf{C}$. Hence, this equation cannot be used for defining a composition in $\mathbf{C}$. Thus, we cannot proceed if $\mathcal{R}$ and $\mathcal{M}$ are mere equivalences.

REFERENCES

[Cic18] Daniel Cicala, *Categorifying the zx-calculus*, Electronic Proceedings in Theoretical Computer Science **266** (2018), 294–314. 3, 4, 5

[DKPvdW20] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering, *Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus*, Quantum **4** (2020), 279. 4

[KvdW20] Aleks Kissinger and John van de Wetering, *PyZX: large scale automated diagrammatic reasoning*, Proceedings 16th International Conference on Quantum Physics and Logic, Electron. Proc. Theor. Comput. Sci. (EPTCS), vol. 318, EPTCS, [place of publication not identified], 2020, pp. 229–241. MR 4146714 5

[Zic24a] Fabian Zickgraf, `CompilerForCAP` – *Speed up and verify categorical algorithms*, 2020–2024, (`https://homalg-project.github.io/pkg/CompilerForCAP`). 7

[3] We will see in Remark A.4 why a mere equivalence does not suffice for our applications.

[Zic24b] Fabian Zickgraf, *Compilerforcap – building and compiling categorical towers in algorithmic category theory*, Dissertation, University of Siegen, 2024, (http://dx.doi.org/10.25819/ubsi/10541). 6

Department of Mathematics, University of Siegen, 57068 Siegen, Germany
Email address: mohamed.barakat@uni-siegen.de

Department of Mathematics, University of Siegen, 57068 Siegen, Germany
Email address: fabian.zickgraf@uni-siegen.de