Typed Compositional Quantum Computation with Lenses

Jacques Garrigue
Nagoya University, Japan

Takafumi Saikawa
Nagoya University, Japan

Abstract

We propose a type-theoretic framework for describing and proving properties of quantum computations, in particular those presented as quantum circuits. Our proposal is based on an observation that, in the polymorphic type system of Coq, currying on quantum states allows one to apply quantum gates directly inside a complex circuit. By introducing a discrete notion of lens to control this currying, we are further able to separate the combinatorics of the circuit structure from the computational content of gates. We apply our development to define quantum circuits recursively from the bottom up, and prove their correctness compositionally.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Quantum computation theory

Keywords and phrases quantum programming, semantics, lens, currying, Coq, MathComp

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.15

Supplementary Material *Software*: `https://github.com/t6s/qecc` [9]
archived at `swh:1:dir:d4d158675180ee276e730bd7f67a9122a6472eb3`

Funding This research was partially supported by JSPS KAKENHI grant No. JP22H00520 and MEXT Q-LEAP grant No. JPMXS0120319794.

1 Introduction

Quantum computation is a theory of computation whose unit of information is the states of a quantum particle, called a quantum bit. A quantum bit is unlike a classical bit in that the former may retain many values at the same time, albeit they ultimately can only be observed as probabilities, while the latter has a single value. This possibility of a multitude of values is preserved by pure quantum computation, and destroyed by a measurement of the probability.

These properties of quantum bits and computation are commonly modelled in terms of unitary transformations in a Hilbert space [19]. Such a transformation is constructed by composing both sequentially and parallelly various simple transformations called quantum gates.

Many works have been built to allow proving quantum algorithms in such settings [15, 18, 20], or more abstractly using string diagrams representing computations in a symmetric monoidal category [5]. We investigate whether some type-theoretic insights could help in describing and proving properties of quantum computations, in particular those denoted by so-called quantum circuits.

Our main goal is to reach *compositionality* inside a semantical representation of computations. We wish it both at the level of definitions and proofs, with as little overhead as possible.

Definitional compositionality means that it should be possible to turn any (pure) quantum circuit into an abstract component, which can be instantiated repeatedly in various larger circuits.

© Jacques Garrigue and Takafumi Saikawa;
licensed under Creative Commons License CC-BY 4.0

Proof compositionality means that the proof of functional properties about (pure) quantum circuits should be statable as a generic lemma about the corresponding abstract component, so that one can build proofs of a large circuit by applying this lemma to instances of the component, without having to unfold the concrete definition of the component during the proof.

Abstraction overhead refers to the extra steps required for abstraction and instantiation, both in definitions and proofs.

The approach we have designed represents circuits as linear transformations, and reaches the above goals by cleanly separating the complex linear algebra in computation from the combinatorics of the wiring, using a combinatorial notion of lens. Compared to more abstract approaches, such as the ZX-calculus [4], we are directly working on an explicit representation of states, but we are still able to prove properties in a scalable way that does not rely on automation, as one can compose circuits without adding complexity to the proof.

Our proposal combines several components, which are all represented using dependent and polymorphic types in Coq. *Finite functions* over n-tuples of bits can encode a n-qubit quantum state. *Lenses* are injections between sets of indices, which can be used to describe the wiring of quantum circuits in a compositional way. They are related to the lenses used for view-update in programming languages and databases [7]. *Currying* of functions representing states, along a lens, provides a direct representation of tensor products. *Polymorphism* suffices to correctly apply transformations to curried states. We need this polymorphism to behave uniformly, which is equivalent to morphisms being natural transformations.

Using these components, we were able to provide a full account of pure quantum circuits in Coq, on top of the MathComp library, proving properties from the ground up. We were also able to prove a number of examples, such as the correctness of Shor coding [17] (formalized for the first time, albeit only for an error-free channel at this point), the Greenberger-Horne-Zeilinger (GHZ) state preparation [10], and the reversed list circuit [20].

Our development is available online [9].

The plan of this paper is as follows. In Section 2, we provide a short introduction to quantum states and circuits. In Section 3, we define lenses. In Section 4, we provide the mathematical definition of focusing of a circuit through a lens. In Sections 5 and 6, we explain the Coq definitions of gates and their composition. In Section 7, we introduce some lemmas used in proof idioms that we apply to examples in Section 8. In Section 9, we define noncommutative and commutative monoids of sequential and parallel compositions of gates. We present related works in Section 10 before concluding.

2 Quantum circuits and unitary semantics

In this section, we present basic notions from linear algebra to describe the unitary model of quantum computation, and how they appear in a quantum circuit diagram.

2.1 Quantum states

Let us first recall that pure classical computation can be seen as a sequence of boolean functions acting on an array of bits of type 2^n for some n. Similarly, pure quantum computation is modeled, in terms of linear algebra, as a sequence of unitary transformations that act on a quantum state of type $\mathbb{C}^{2^n}$.

A quantum bit (or *qubit*) is the most basic unit of data in quantum computation. We regard it as a variable of type $\mathbb{C}^2$ and each vector of norm 1 is considered to be a state of the qubit. $\mathbb{C}^2$ has a standard basis $(1, 0), (0, 1)$, which we denote in the context of quantum

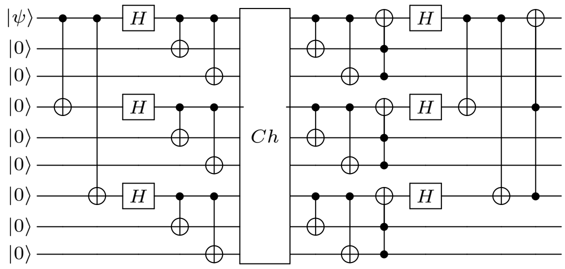


Figure 1 Shor's 9-qubit code.

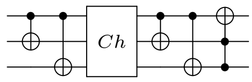


Figure 2 Bit-flip code.

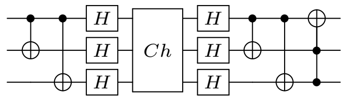


Figure 3 Sign-flip code.

programming $|0\rangle, |1\rangle$, indicating that the state of the qubit is 0 and 1 respectively. Regarding $\mathbb{C}^2$ as the function space $[2] \to \mathbb{C}$, where $[n]$ stands for $\{0, \dots, n-1\}$, we can express the standard basis in the form of functions

$$|0\rangle := x \mapsto \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \qquad |1\rangle := x \mapsto \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

States other than basis states are linear combinations, which we call *superpositions*. The state of a qubit is mapped to a classical bit by an operation called *measurement*, which probabilistically results in values 0 or 1. The measurement of a state in superposition $a\,|0\rangle + b\,|1\rangle$ results in 0 with probability $|a|^2$ and 1 with probability $|b|^2$.

Those definitions naturally extend to n-ary quantum states. The basis states for n qubits are functions

$$|i_1 i_2 \dots i_n\rangle := (x : [2]^n) \longmapsto \begin{cases} 1 & \text{if } x = (i_1, i_2, \dots, i_n) \\ 0 & \text{otherwise} \end{cases}$$

States other than basis states are again superpositions, which are linear combinations of norm 1. In other words, a state is represented by a function of type $\mathbb{C}^{2^n}$, besides the condition on its norm. We hereafter regard this type as the space of states. This type can also be identified with the n-ary tensor power $(\mathbb{C}^2)^{\otimes n}$ of $\mathbb{C}^2$, a usual presentation of states in textbooks.

Similarly to the unary case, a measurement of an n-ary quantum state $\sum_{i \in 2^n} c_i\,|i_1 i_2 \dots i_n\rangle$ results in an array of classical bits $i = (i_1, i_2, \dots, i_n)$ with probability $|c_i|^2$.

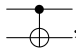
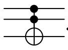
2.2 Unitary transformations

We adopt the traditional view that pure quantum computation amounts to applying unitary transformations to a quantum state. A unitary transformation is a linear function from a vector space to itself that preserves the inner product of any two vectors, that is, $\langle U(a) \mid U(b)\rangle$ is equal to $\langle a \mid b\rangle$ for any unitary U and vectors a and b, if we denote the inner product by $\langle a \mid b\rangle$. Since the norm of a is defined to be $\sqrt{\langle a \mid a\rangle}$, a unitary also preserves the norm condition of quantum states.

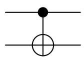
2.3 Quantum circuits

In the same way that classical computation can be expressed by an electronic circuit comprised of boolean gates (AND, OR, etc.), quantum computation is also conveniently presented as a circuit with quantum gates that represent primitive unitary transformations. More generally, a quantum circuit may contain nonunitary operations such as measurement, but we restrict ourselves to pure quantum circuits that contain none of them.

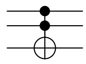
A quantum circuit is a concrete representation of quantum computation, drawn as n parallel wires with quantum gates and other larger subcircuits being placed over those wires. A quantum state is input from the left end of a circuit, transformed by gates and subcircuits on the corresponding wires, and output from the right end. As an example, we show the Shor's 9-qubit error correction code (Figure 1) and its subcomponents (Figures 2 and 3).

The primitive operations in a quantum circuit are quantum gates. In the Shor's code, three kinds of gates appear, namely Hadamard $-\boxed{H}-$, Controlled Not (CNOT) , and Toffoli . The large box $\boxed{Ch}$ denotes an arbitrary unitary transformation modelling a possibly erroneous channel. The gates placed to the left of $\boxed{Ch}$ implement the encoder algorithm of the code, and those to the right the decoder. The unitary operations denoted by these gates can be expressed as matrices with respect to the lexicographically ordered standard basis (e.g. $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ for two qubits):

$$-\boxed{H}- = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$



$$= \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$$



$$= \begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$$

A gate composed in a circuit is represented by a matrix by, first taking the Kronecker product with identity matrices corresponding to irrelevant wires, and second sandwiching it with the matrices that represent the action of a permutation on the index of tensors to reorder the input and output wires. For example, to describe the leftmost CNOT gate in the Shor's code, we first *pad* (append) seven wires to CNOT by taking the Kronecker product with $I_{2^7} = I_{128}$ and apply the permutation (24) to move $\oplus$ from the second wire to the fourth wire. The resulting matrix is:

$$U_{2^9}((42))\begin{bmatrix}I_{128} & 0 & 0 & 0\\ 0 & I_{128} & 0 & 0\\ 0 & 0 & 0 & I_{128}\\ 0 & 0 & I_{128} & 0\end{bmatrix}U_{2^9}((24))$$

where $U_{2^9}((24))$ denotes the matrix representation of (24) that maps the basis vectors $|i_1\, i_2\, i_3\, i_4\, i_5\, i_6\, i_7\, i_8\, i_9\rangle$ to $|i_1\, i_4\, i_3\, i_2\, i_5\, i_6\, i_7\, i_8\, i_9\rangle$, and its inverse $U_{2^9}((42))$ is the same since $(42) = (24)$.

The above method realizes the padding and permutation as linear transformations, resulting in multiplications of huge matrices. Taken literally, this method is compositional in that the embedding of a smaller circuit into a larger one can be iterated, but impractical because of the exponential growth of the dimension of the matrices. A way to avoid this problem is to stick to a symbolic representation based on sums of matrix units, that can ignore zero components, but it is less compositional, in that the representation of the gate is modified to fit an application site, leading to different representations and reasoning at different sites. We aim at solving this problem by separating the wiring part, which is a combinatorics that does not essentially touch quantum states, from the actions of a quantum gate, which is an intrinsic property of the gate itself.

3 Lenses

The first element of our approach is to provide a data structure, which we call a *lens*, that describes the composition of a subcircuit into a circuit. It forms the basis for a combinatorics of composition.

The concept of lens [7] was introduced in the programming language community as a way to solve the *view-update* problem [1], which itself comes from the database community. Lenses are often described as a pair of functions **get** : $S \to V$ and **put** : $V \times S \to S$, which satisfy the laws GetPut : $\mathbf{put}(\mathbf{get}(s), s) = s$ and PutGet : $\mathbf{get}(\mathbf{put}(v, s)) = v$. A more versatile approach adds the concept of complementary view [1, 2], which adds another type C and a function $\mathbf{get}^{\mathtt{C}} : S \to C$, changing the type of **put** to $V \times C \to S$, so that the first law becomes $\mathbf{put}(\mathbf{get}(s), \mathbf{get}^{\mathtt{C}}(s)) = s$.

Our representation of lenses is an instance of the second approach. We want to map the m wires of a subcircuit to the n wires of the external one. This amounts to defining an injection from $[m]$ to $[n]$, which can be represented canonically as a list of m indices in $[n]$, without repetition.

Record $\mathtt{lens}_{n,m} := \{\ell : [n]^m \mid \mathtt{uniq}\ \ell\}$.

Throughout this paper, we use mathematical notations to make our Coq code easier to read. For instance $[n]$ in the above record definition denotes the ordinal type `'I_n` of MathComp, and $[n]^m$ denotes the type of tuples of arity m of this type (i.e. the type `m.-tuple 'I_n`). We also write type parameters as indices, and allow for omitting them.

We call *focusing* the operation using a lens to update a system accoding to changes in a subsystem. The following operations on lenses are basic and required to define focusing.

Definition $\mathtt{extract}_{T,n,m} : \mathtt{lens}_{n,m} \to T^n \to T^m$.
Definition $\mathtt{lensC}_{n,m} : \mathtt{lens}_{n,m} \to \mathtt{lens}_{n,n-m}$.
Definition $\mathtt{merge}_{T,n,m} : \mathtt{lens}_{n,m} \to T^m \to T^{n-m} \to T^n$.

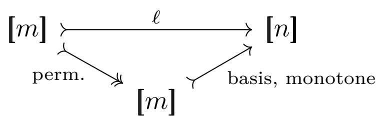
The **get** operation of lens ℓ is `extract` ℓ, which is the projection of T^n onto T^m along ℓ. Each lens ℓ has its complementary lens `lensC` ℓ, which is the unique monotone bijection from $[n-m]$ to $[n] \setminus \mathrm{Im}(\ell)$. We will write $\ell^{\mathtt{C}}$ for `lensC` ℓ. Their composition `extract` $\ell^{\mathtt{C}}$ returns the complementary view. The corresponding **put** operation is `merge` $\ell\ v\ c$. In the following, the lens ℓ will be available from the context, so that we omit it in `extract` and `merge`, and $\mathtt{extract}^{\mathtt{C}}$ denotes `extract` $\ell^{\mathtt{C}}$. The GetPut and PutGet laws become:

Lemma merge_extract : merge (extract v) ($\mathtt{extract}^{\mathtt{C}}$ v) = v.
Lemma extract_merge : extract (merge v_1 v_2) = v_1.
Lemma extractC_merge : $\mathtt{extract}^{\mathtt{C}}$ (merge v_1 v_2) = v_2.

We show the classical case of focusing (`focus1`) as an example (Figure 4). In this case, data is represented by direct products, whose elements are tuples, readily manipulated by `extract` and `merge`. A change on the subsystem of type T^m is thus propagated to the global state of type T^n.

Definition $\mathtt{focus1}_{T,n,m}$ $(\ell : \mathtt{lens}_{n,m})$ $(f : T^m \to T^m)$: $T^n \to T^n$:=
  $s \mapsto$ merge $(f$ (extract $s))$ ($\mathtt{extract}^{\mathtt{C}}$ s).
Lemma focus1_in : $\mathtt{extract} \circ (\mathtt{focus1}_\ell\ f) = f \circ \mathtt{extract}$.

It is also often useful to compose lenses, or to factorize a lens into its basis (the monotone part) and permutation part.



Namely, we have the following functions and laws:

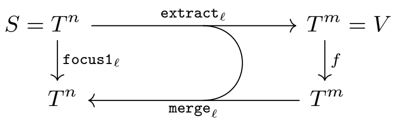


Figure 4 Classical focusing.

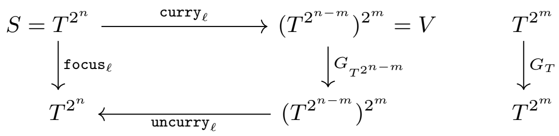


Figure 5 Quantum focusing.

Definition $\texttt{lens\_comp}_{n,m,p}$: $\texttt{lens}_{n,m} \to \texttt{lens}_{m,p} \to \texttt{lens}_{n,p}$.
Definition $\texttt{lens\_basis}_{n,m}$: $\texttt{lens}_{n,m} \to \texttt{lens}_{n,m}$.
Definition $\texttt{lens\_perm}_{n,m}$: $\texttt{lens}_{n,m} \to \texttt{lens}_{m,m}$.
Lemma lens_basis_perm : lens_comp (lens_basis ℓ) (lens_perm ℓ) $= \ell$.
Lemma mem_lens_basis : lens_basis $\ell =_i \ell$.

where $\ell_1 =_i \ell_2$ means that ℓ_1 and ℓ_2 are equal as sets.

4 Quantum focusing

We are going to define actions of lenses on quantum states and operators. The classical operators `merge` and `extract` introduced in the previous section play an important role in the definition.

In the quantum case, the **get** operation must not discard the irrelevant part of an input state, unlike the classical one that was defined as a projection. Such a quantum **get** and the corresponding **put** operations can be defined in a form of currying and uncurrying:

Definition $\texttt{curry}_{T,n,m}$: $\texttt{lens}_{n,m} \to T^{2^n} \to (T^{2^{n-m}})^{2^m}$.
Definition $\texttt{uncurry}_{T,n,m}$: $\texttt{lens}_{n,m} \to (T^{2^{n-m}})^{2^m} \to T^{2^n}$.

The type parameter T is intended to vary over $\mathbb{C}$-modules, whose archetypical example is $\mathbb{C}$ itself. The result of applying $\texttt{curry}_\ell$ to an input state $\sigma \in T^{2^n}$ is a function that takes two indexing tuples $v \in 2^m$ and $w \in 2^{n-m}$ and returns $\sigma(\texttt{merge}\ v\ w)$, the evaluation of σ at the combined index of v and w along ℓ. Its inverse $\texttt{uncurry}_\ell$ is defined similarly as $\sigma(\texttt{extract}\ v)(\texttt{extract}^{\texttt{C}}\ v)$ for $\sigma \in (T^{2^{n-m}})^{2^m}$ and $v \in 2^n$.

We verify that `curry` and `uncurry` form an isomorphism by cancellation lemmas:

Lemma curryK : $\texttt{uncurry}_\ell \circ \texttt{curry}_\ell = \mathrm{id}_{T^{2^n}}$.
Lemma uncurryK : $\texttt{curry}_\ell \circ \texttt{uncurry}_\ell = \mathrm{id}_{(T^{2^{n-m}})^{2^m}}$.

When specialized to $T := \mathbb{C}$, we can further follow another isomorphism derived from the adjunction between the category **Set** of sets and **Vect** of vector spaces, showing that our `curry` is actually equivalent to the currying for tensor products in **Vect**:

$$\mathbb{C}^{2^n} \cong \left(\mathbb{C}^{2^{n-m}}\right)^{2^m} = \mathbf{Set}\left(2^m, \mathbb{C}^{2^{n-m}}\right) \cong \mathbf{Vect}\left(\mathbb{C}^{2^m}, \mathbb{C}^{2^{n-m}}\right).$$

An m-qubit quantum gate G is a linear transformation on $\mathbb{C}^{2^m}$, and it can be represented by a matrix. The action of this matrix on a 2^m-dimensional vector is computed only by scalar multiplications and additions. Therefore, the action can be extended to T^{2^m} for any $\mathbb{C}$-module T. We are thus led to endow such G with a polymorphic type of linear transformations indexed by T.

$$G : \forall T : \mathbb{C}\text{-module}, T^{2^m} \xrightarrow{\text{linear}} T^{2^m}$$

Along the `curry-uncurry` isomorphism above, a gate G can be applied to a larger number of qubits, to become composable in a circuit. This realizes quantum focusing (Figure 5).

$$\texttt{focus}_\ell\ G := \Lambda T.(\texttt{uncurry}_\ell \circ G_{T^{2^{n-m}}} \circ \texttt{curry}_\ell)$$

So far, the type of G has told that each instance G_T is linear and can be represented by a matrix, but not that they are the same matrix for any T. We impose the uniqueness of the matrix as an additional property as follows.

$$\exists M : \mathcal{M}_{2^m}(\mathbb{C}),\ \forall T : \mathbb{C}\text{-module},\ \forall s : T^{2^m},\ G_T(s) = Ms.$$

Here the multiplication Ms is defined for $s = (s_1, \ldots, s_{2^m})^t$ and $M = (M_{(i,j)})_{i,j}$ as

$$Ms := \sum_{1 \leq j \leq 2^m} (M_{(1,j)} s_j, \ldots, M_{(2^m,j)} s_j)^t.$$

This existence of a unique matrix representation implies the uniformity of the actions of G, which amounts to naturality with respect to the functor $(-)^{2^m}$:

$$\begin{array}{ccccc} T & \quad & T^{2^m} & \xrightarrow{G_T} & T^{2^m} \\ \forall\varphi \big\downarrow & & \varphi^{2^m} \big\downarrow & & \big\downarrow \varphi^{2^m} \\ T' & & T'^{2^m} & \xrightarrow[G_{T'}]{} & T'^{2^m} \end{array}$$

We proved conversely that this naturality implies the uniqueness of the matrix. We shall incorporate naturality, instead of a matrix, in our definition of quantum gates.

5 Defining quantum gates

Using MathComp, we can easily present the concepts described in the previous sections. From here on, we fix K to be a field, and denote by K^1 the one-dimensional vector space over K to distinguish them as different types.

We first define quantum states as the double power T^{2^n} discussed in Section 2. It is encoded as a function type $T^{\widehat{n}}$ from n-tuples of some finite type I to a type T. For qubits, we shall have $I = [\mathbf{2}] = \{0, 1\}$, but we can also naturally represent qutrits (quantum information units with three states) by choosing $I = [\mathbf{3}]$.

`Variables` $(I : \text{finite type})\ (dI : I)\ (K : \text{field})\ (T : K\text{-module})$.
`Definition` $T^{\widehat{n}} := I^n \xrightarrow{\text{finite}} T$.
`Definition` $\texttt{dpmap}_{m,T_1,T_2}\ (\varphi : T_1 \to T_2)\ (s : T_1^{\widehat{m}}) : T_2^{\widehat{m}}$:= $\varphi \circ s$.

This construction, $(-)^{\widehat{n}}$, can be regarded as a functor with its action on functions provided by `dpmap`, that is, any function $\varphi : T_1 \to T_2$ can be extended to `dpmap` $\varphi : T_1^{\widehat{n}} \to T_2^{\widehat{n}}$, which are drawn as the vertical arrows in the naturality square in the previous section.

We next define quantum gates as natural transformations (or *morphisms*).

`Definition` $\texttt{morlin}_{m,n} := \forall\ T : K\text{-module},\ T^{\widehat{m}} \xrightarrow{\text{linear}} T^{\widehat{n}}$.
`Definition` $\texttt{naturality}_{m,n}\ (G : \texttt{morlin}_{m,n})$:=
  $\forall (T_1\ T_2 : K\text{-module}), \forall (\varphi : T_1 \xrightarrow{\text{linear}} T_2),\ (\texttt{dpmap}\ \varphi) \circ (G\ T_1) = (G\ T_2) \circ (\texttt{dpmap}\ \varphi)$.
`Record` $\texttt{mor}_{m,n} := \{G : \texttt{morlin}_{m,n} \mid \texttt{naturality}\ G\}$.
`Notation` $\texttt{endo}_n := (\texttt{mor}_{n,n})$.
`Definition` $\texttt{unitary\_mor}_{m,n}\ (G : \texttt{mor}_{m,n})$:= $\forall s, t, \langle G_{K^1}\ s \mid G_{K^1}\ t \rangle = \langle s \mid t \rangle$.

A crucial fact we rely on here is that, for any K-module T, MathComp defines the K-module of the finite functions valued into it, so that $T^{\widehat{n}}$ is a K-module. This allows us to define the type `morlin` of polymorphic linear functions between $T^{\widehat{m}}$ and $T^{\widehat{n}}$, and further combine it with naturality into the types $\texttt{mor}_{m,n}$ of morphisms from $(-)^{\widehat{m}}$ to $(-)^{\widehat{n}}$ and $\texttt{endo}_n$ of endo-morphisms.

We leave unitarity as an independent property, called `unitary_mor`, since it makes sense to have non-unitary morphisms in some situations.

Concrete quantum states can be expressed directly as functions in $(K^1)^{\widehat{n}}$, or as a linear combination of computational basis vectors $|v\rangle$, where $v : I^n$ is the index of the only 1 in the vector.

$$\texttt{Definition}\ |v\rangle : (K^1)^{\widehat{n}} := (v' : I^n) \mapsto \begin{cases} 1 & \text{if } v = v' \\ 0 & \text{otherwise} \end{cases}$$

For a concrete tuple, we also write $|i_1, \ldots, i_n\rangle$ for $|[\texttt{tuple}\ i_1; \ldots; i_n]\rangle$. This representation of states allows us to go back and forth between computational basis states and indices, and is amenable to proofs.

Using this basis, one can also define a morphism from its matrix representation (expressed as a nested double power, in column-major order). We define the CNOT gate as mapping from computational basis indices to column vectors, using $v[i]$ as a notation for the ith element of the tuple v, aka $\texttt{tnth}\ v\ i$. The expression `ket_bra k b` stands for the product of a column vector and a row vector, resulting in an $m \times n$ matrix (written $|k\rangle\langle b|$ in the Dirac notation). We use it to define the Hadamard gate as a sum of matrix units. Both matrices are then fed to `dpmor` to obtain morphisms.

$$\texttt{Definition dpmor}_{m,n} : ((K^1)^{\widehat{n}})^{\widehat{m}} \to \texttt{mor}_{m,n}.$$

$$\texttt{Definition ket\_bra}_{m,n}\ (k : (K^1)^{\widehat{m}})\ (b : (K^1)^{\widehat{n}}) : ((K^1)^{\widehat{n}})^{\widehat{m}} := v \mapsto (k\ v) \cdot b.$$
$$\texttt{Definition cnot} : \texttt{endo}_2 := \texttt{dpmor}\ (v : [2]^2 \mapsto |v[0], v[0] \oplus v[1]\rangle).$$
$$\texttt{Definition hadamard} : \texttt{endo}_1 :=$$
$$\texttt{dpmor}\ \left(\frac{1}{\sqrt{2}}(\texttt{ket\_bra}\ |0\rangle\ |0\rangle + \texttt{ket\_bra}\ |0\rangle\ |1\rangle + \texttt{ket\_bra}\ |1\rangle\ |0\rangle - \texttt{ket\_bra}\ |1\rangle\ |1\rangle)\right).$$

As explained in Section 4, naturality for a morphism is equivalent to the existence of a uniform matrix representation.

$$\texttt{Lemma naturalityP} : \texttt{naturality}\ G \longleftrightarrow \exists M,\ \forall T, s,\ G_T\ s = (\texttt{dpmor}\ M)_T\ s.$$

On the right hand side of the equivalence we use the extensional equality of morphisms, which quantifies on T and s. By default, it is not equivalent to Coq's propositional equality; however the two coincide if we assume functional extensionality and proof irrelevance, two relatively standard axioms inside Coq.

$$\texttt{Lemma morP} : \forall (F, G : \texttt{mor}_{m,n}),\ (\forall T, s,\ F_T\ s = G_T\ s) \longleftrightarrow F = G.$$

While our development distinguishes between the two equalities, in this paper we will not insist on the distinction, and just abusively write $F = G$ for extensional equality too. Only in Section 9 will we use those axioms to prove and use the above lemma.

6 Building circuits

The currying defined in Section 4 allows us to compose circuits without referring to a global set of qubits. This is obtained through two operations: (sequential) composition of morphisms, which just extends function composition, and focusing through a lens, which allows us to connect the wires of a gate into a larger circuit.

```
Definition ●_{n,m,p}   : mor_{m,p} → mor_{n,m} → mor_{n,p} .
Definition focus_{n,m} : lens_{n,m} → endo_m → endo_n .
```

To define `focus`, we combine currying and polymorphism into `focuslin` as we did in Section 4, and add a proof of naturality.

```
Definition focuslin_{n,m} (ℓ : lens_{n,m}) (G : endo_m) : morlin_{n,n} :=
  ΛT. (uncurry ℓ)_T ∘ G_{T^{\widehat{n-m}}} ∘ (curry ℓ)_T .
Lemma focusN ℓ G : naturality (focuslin ℓ G).
Definition focus_{n,m} ℓ G := (a morphism packing focuslin ℓ G and focusN ℓ G).
```

In particular, `focus` and sequential composition satisfy the following laws, derived from naturality and lens combinatorics.

```
Lemma focus_comp : focus_ℓ (F • G) = (focus_ℓ F) • (focus_ℓ G).
Lemma focusM : focus_(lens_comp ℓ ℓ') G = focus_ℓ (focus_ℓ' G).
Lemma focusC : ℓ and ℓ' disj. → (focus_ℓ F) • (focus_ℓ' G) = (focus_ℓ' G) • (focus_ℓ F).
Lemma unitary_comp : unitary_mor F → unitary_mor G → unitary_mor (F • G).
Lemma unitary_focus : unitary_mor G → unitary_mor (focus_ℓ G).
```

The law `focus_comp` states that the sequential composition of morphism commutes with focusing. Similarly, `focusM` states that the composition of lenses commutes with focusing. The law `focusC` states that the sequential composition of two morphisms focused through disjoint lenses (i.e. lenses whose codomains are disjoint) commutes. The last two lemmas are about unitarity. Since all circuits can be built from unitary basic gates using sequential composition and `focus`, they are sufficient to guarantee unitarity for all of them.

7 Proving correctness of circuits

Once we have defined a circuit by combining gates through the above functions, we want to prove its correctness. Usually this involves proving a relation between the input and the output of the transformation, which can be expressed as a behavior on computational basis vectors. In such situations, the following lemmas allow the proof to progress.

```
Variables (n m : ℕ) (ℓ : lens_{n,m}).
Definition dpmerge : I^n → (K^1)^{\widehat{m}} --linear--> (K^1)^{\widehat{n}} .
Lemma focus_dpbasis : (focus_ℓ G)_{K^1} |v⟩ = dpmerge v (G_{K^1} |extract v⟩).
Lemma dpmerge_dpbasis : dpmerge v |v'⟩ = |merge v' (extract^C v)⟩.
Lemma decompose_scaler : ∀(σ : (K^1)^{\widehat{n}}), σ = ∑_{v:I^k} σ(v) · |v⟩ .
```

The function `dpmerge` embeds the result of a quantum gate applied to a part of the system into the whole system, using the input computational basis vector for complement; this can be seen as an asymmetric variant of the **put** operation. It is defined using $\mathtt{uncurry}_\ell$ and `dpmap`. It is only introduced and eliminated through the two lemmas following. The helper law `focus_dpbasis` allows one to apply the morphism G to the local part of the basis vector v. The result of this application must then be decomposed into a linear combination of (local) basis vectors, either by using the definition of the gate, or by using `decompose_scaler`. One can then use linearity to obtain terms of the form `dpmerge` v $|v'\rangle$ and merge the local result into the global quantum state. Linear algebra computations have good support in MathComp, so we do not need to extend it much.

Extraction and merging only rely on lens-related lemmas, orthogonal to the linear algebra part. We have not yet developed a complete theory of lenses, but we have many such lemmas. The following ones are of particular interest:

```
Section lens_index.
Variables (n m : ℕ) (i : [n]) (ℓ : lens_{n,m}).
Definition lens_index (H : i ∈ ℓ) : [m].
Lemma tnth_lens_index : ∀(H : i ∈ ℓ), ℓ[lens_index H] = i.
Lemma tnth_merge : ∀(H : i ∈ ℓ), (merge v v′)[i] = v[lens_index H].
Lemma tnth_extract : (extract v)[j] = v[ℓ[j]].
Lemma mem_lensC : (i ∈ ℓ^C) = (i ∉ ℓ).
Lemma mem_lens_comp : ∀(H : i ∈ ℓ), (i ∈ lens_comp ℓ ℓ′) = (lens_index H ∈ ℓ′).
End lens_index.
Lemma tnth_mergeC : ∀(H : i ∈ ℓ^C), (merge v v′)[i] = v′[lens_index H].
```

The expression `lens_index` H, where H is a proof that i is in ℓ, denotes the ordinal position of i in ℓ, hence the statement of `tnth_lens_index`. It is particularly useful in `tnth_merge` and `tnth_mergeC`, where it allows one to prove equalities of tuples and lenses through case analysis on the boolean expression $i \in \ell$ (using `mem_lensC` for conversion).

Using these two techniques we have been able to prove the correctness of a number of pure quantum circuits, such as Shor's 9-qubit code or the GHZ preparation.

8 Concrete examples

When working on practical examples we move to more concrete settings. Namely, we use $\mathbb{C}$ as the coefficient field, which can also be seen as the vector space `Co` $= \mathbb{C}^1$. The indices are now in `I` $= [2] = \{0, 1\}$. In this section we use Coq notations rather than the mathematical ones of the previous sections, so as to keep close to the actual code.

As an example, let us recall the circuit diagram of Shor's code (Figure 1). It consists of two smaller components, bit-flip and sign-flip codes (Figures 2 and 3), in such a way that three bit-flip codes are placed in parallel and surrounded by one sign-flip code. This construction can be expressed straightforwardly as the following Coq code.

```
Definition bit_flip_enc : endo3 := focus [lens 0; 2] cnot • focus [lens 0; 1] cnot.
Definition bit_flip_dec : endo3 := focus [lens 1; 2; 0] toffoli • bit_flip_enc.
Definition hadamard3 : endo3 :=
  focus [lens 2] hadamard • focus [lens 1] hadamard • focus [lens 0] hadamard.
Definition sign_flip_dec := bit_flip_dec • hadamard3.
Definition sign_flip_enc := hadamard3 • bit_flip_enc.
Definition shor_enc : endo9 :=
 focus [lens 0; 1; 2] bit_flip_enc • focus [lens 3; 4; 5] bit_flip_enc •
 focus [lens 6; 7; 8] bit_flip_enc • focus [lens 0; 3; 6] sign_flip_enc.
Definition shor_dec : endo9 := ...
```

We proved that Shor's code is the identity on an error-free channel:

```
Theorem shor_code_id : (shor_dec • shor_enc) |i,0,0,0,0,0,0,0,0⟩ = |i,0,0,0,0,0,0,0,0⟩.
```

The proof is compositional, relying on lemmas for each subcircuit.

```
Lemma cnotE : cnot |i,j⟩ = |i,i+j⟩.
Lemma toffoliE00 : toffoli |0,0,i⟩ = |0,0,i⟩.
Lemma hadamardK : ∀T, involutive hadamard_T.
```

```
1    bit_flip_enc ¦ i, j, k ⟩
2  rewrite /=.
3  = focus [lens 0; 2] cnot (focus [lens 0; 1] cnot ¦ i, j, k ⟩)
4  rewrite focus_dpbasis.
5  = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k]
6                             (cnot ¦ extract [lens 0; 1] [tuple i; j; k]⟩)
7  simpl_extract.
8  = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k] (cnot ¦ i, j ⟩))
9  rewrite cnotE.
10 = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k] ¦ i, i + j ⟩)
11 rewrite dpmerge_dpbasis.
12 = focus [lens 0; 2] cnot ¦ merge [lens 0; 1] [tuple i; i + j]
13                             (extract (lensC [lens 0; 1]) [tuple i; j; k]) ⟩
14 simpl_merge.
15 = focus [lens 0; 2] cnot ¦ i, i + j, k ⟩
```

Figure 6 Excerpt of interactive proof of `bit_flip_enc_ok`.

```
Lemma bit_flip_enc_ok : bit_flip_enc |i,j,k⟩ = |i,i+j,i+k⟩.
Lemma bit_flip_toffoli : bit_flip_dec • bit_flip_enc = focus [lens 1;2;0] toffoli.
Lemma sign_flip_toffoli: sign_flip_dec • sign_flip_enc = focus [lens 1;2;0] toffoli.
```

The notation $(i : [m])$ in expressions (here in `flip`) denotes that we have a proof that $i \in [m]$; in the actual code one uses specific function to build such dependently-typed values. The first 3 lemmas describe properties of the matrix representation of gates, and involve linear algebra computations. The proof of `HadamardK` also involves some real computations about $\sqrt{2}$. The remaining 3 lemmas and the theorem do mostly computations on lenses. In total, there were about 100 lines of proof.

To give a better idea of how the proofs proceed, we show a few steps of the beginning of `bit_flip_enc_ok`, in Figure 6, interspersing tactics on a gray background between quantum state expressions and equations. Lines beginning with an "`=`" symbol state that the expression is equal to the previous one.

Simplifying on line 2 reveals the focused application of the two `cnot` gates. Rewriting with `focus_dpbasis`, on line 4, applies the first gate directly to a basis vector. The helper tactic `simpl_extract`, on line 7, computes the tuple obtained by `extract` (MathComp is not good at computing in presence of dependent types). It results here in the vector $|i,j\rangle$, which we can rewrite with `cnotE`. As a result, on line 10, `dpmerge` is applied to a basis vector, so that we can rewrite it with `dpmerge_dpbasis`. Again, on line 14, we use a helper tactic `simpl_merge`, which uses the same code as `simpl_extract` to simplify the value of the merge expression. We obtain $|i,i+j,k\rangle$ as result after the first gate, and can proceed similarly with the second gate to reach $|i,i+j,i+k\rangle$.

As we explained above, our approach cleanly separates computation on lenses from linear algebra parts. Namely, in the above proof we have three logical levels: `focus_dpbasis` and `dpmerge_dpbasis` let one get in and out of a `focus` application; `simpl_extract` and `simpl_merge` are doing lens computations; and finally `cnotE` uses a property of the specific gate.

The proof of `shor_code_id` is more involved as the Hadamard gates introduce superpositions. The code can be found in the file `qexamples_shor.v` of the accompanying development, and is about 30 lines long. We will just explain here the main steps of the proof. The basic idea is to pair the encoders and decoders, and to turn them into Toffoli gates, which happen to be identities when the extra inputs are zeros. The first goal is to prove that

```
(shor_dec • shor_enc) |i,0,0,0,0,0,0,0,0⟩ =
focus [lens 0;3;6] (sign_flip_dec • sign_flip_enc) |i,0,0,0,0,0,0,0,0⟩
```

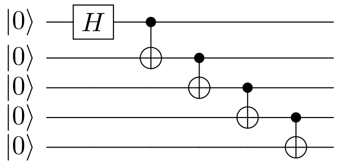


Figure 7 5-qubit GHZ state preparation.

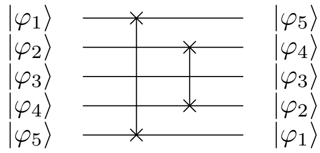


Figure 8 5-qubit reversed state circuit.

If we expand the compositions on both sides, we see that they both start by applying `focus [lens 0;3;6] sign_flip_enc` to the input. We can use `focus_dpasis` and `simpl_extract` to progress, but due to the Hadamard gates in `sign_flip_enc`, the state of the corresponding 3 qubits becomes non-trivial. However, we can use `decompose_scaler` to see this state as a sum of unknown computational basis vectors, and progress using linear algebra lemmas, to reach the bit-flip part of the circuit. Once we do that, the remainder of the proof consists in using `focusC` to reorder the bit-flip encoders and decoders, so that the corresponding ones are sequentially paired. We can then use `focus_comp` to produce applications of `bit_flip_dec • bit_flip_enc`, which can be converted to Toffoli gates by `bit_flip_toffoli`. Then we observe that in the input the ancillaries are all zeros, so that the result of each gate is the identity, which concludes the first part of the proof. Then we can proceed similarly to prove that the remaining composition of the sign-flip encoder and decoder is the identity, which concludes the proof.

Another interesting example is the Greenberger-Horne-Zeilinger (GHZ) state preparation. It is a generalization of the Bell state, resulting in a superposition of $|0\rangle^{\otimes n}$ and $|1\rangle^{\otimes n}$, which denote states composed of n zeroes and ones, respectively. As a circuit, it can be expressed by the composition of one Hadamard gate followed by n CNOT gates, each one translated by 1 qubit, starting from the state $|0\rangle^{\otimes n}$. The 5-qubit case is shown in Figure 7.

We can write the transformation part as follows in our framework (for an arbitrary n):

```
Lemma succ_neq n (i : [n]) : (i : [n + 1]) ≠ (i + 1 : [n + 1]).
Fixpoint ghz n :=
  match n as n return endo_n.+1 with
  | 0 => hadamard
  | m.+1 => focus (lens_pair (succ_neq (m:[m.+1]))) cnot •
            focus (lensC (lens_single (m.+1:[m.+2]))) (ghz m)
  end.
```

The definition works by composing $\texttt{ghz}(m)$, which has type $\texttt{endo}_n$ (since $n = m + 1$), with an extra CNOT gate. Note that we use dependent types, and the recursion is at a different type. The lemma `succ_neq` is a proof that $i \neq i + 1$ in $[n + 1]$. It is used by `lens_pair` to build the lens $[\texttt{lens}\ m; m + 1]$ from $[2]$ to $[m + 2]$. `lens_single` builds a singleton lens, so that `lensC (lens_single (m.+1:[m.+2]))` is the lens from $[m + 1]$ to $[m + 2]$ connecting the inner circuit to the first $m + 1$ wires. We can express the target state and correctness property as follows:

```
Definition ghz_state n : (ℂ^1)^{\widehat{n+1}} := 1/√2 · (|0⟩^{⊗(n+1)} + |1⟩^{⊗(n+1)}).
Lemma ghz_ok : ghz n |0⟩^{⊗(n+1)} = ghz_state n.
```

Due to the nesting of lenses, the proof includes a lot of lens combinatorics, and is about 50 lines long. We only show the last few lines of the proof in Figure 9, as they include typical steps. They prove the action of the last CNOT gate of the circuit when it propagates a 1 to the last qubit of the state. The notation `[tuple F i | i < n]` denotes the n-tuple whose ith

```
1    lp := lens_pair (succ_neq (n : [n.+1]))
2    =======================================
3    merge lp [tuple 1; 1]
4      (extract (lensC lp) [tuple if i != n.+1 then 1 else 0 | i < n.+2])
5    = [tuple 1 | _ < n.+2]
6  apply eq_from_tnth => i; rewrite [RHS]tnth_mktuple.
7  case/boolP: (i \in lp) => Hi.
8    Hi : i \in lp
9    ============================
10   tnth (merge lp [tuple 1; 1]
11     (extract (lensC lp) [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
12 rewrite tnth_merge -[RHS](tnth_mktuple (fun=>1) (lens_index Hi)).
13   tnth [tuple 1; 1] (lens_index Hi) = tnth [tuple 1 | _ < 2] (lens_index Hi)
14 by congr tnth; eq_lens.
15   Hi : i \notin lp
16   ============================
17   tnth (merge lp [tuple 1; 1]
18    (extract (lensC lp)) [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
19 rewrite -mem_lensC in Hi.
20 rewrite tnth_mergeC tnth_extract tnth_mktuple.
21   Hi : i \in lensC lp
22   ============================
23   (if tnth (lensC lp) (lens_index Hi) < n.+1 then 1 else 0) = 1
24 rewrite tnth_lens_index ifT //.
25   i != n.+1
26 move: Hi; rewrite mem_lensC !inE; apply contra.
27   i == n.+1 -> (i == (n : [n.+2])) || (i == (n.+1 : [n.+2]))
28 by move/eqP => Hi; apply/orP/or_intror/eqP/val_inj.
```

Figure 9 Excerpt of interactive proof of `ghz_ok`.

element is `F i`. Lemma `eq_from_tnth` on line 6 allows index-wise reasoning. The `tnth_mktuple` on the same line extracts the ith element of the tuple comprehension on the right-hand side. We immediately do a case analysis on whether i is involved in the last gate. In the first case, we have $i \in$ `lens_pair`(`succ_neq`($n : [\![n+1]\!]$)), so we can use `tnth_merge` on the left-hand side. On the right-hand side we use `tnth_mktuple` backwards, to introduce a 2-tuple. As a result, we obtain on line 13 a goal on which we can use congruence, and conclude with `eq_lens` as both tuples are equal. The second case, when $i \notin$ `lens_pair`(`succ_neq`($n : [\![n+1]\!]$)), is more involved. By using `mem_lensC` in `Hi`, we can use `tnth_mergeC`, followed by `tnth_extract` and `tnth_mktuple` to reach the goal at line 21. But then the argument to `tnth` is precisely that of `Hi`, so this expression can be rewritten to i by `tnth_lens_index`. From line 25 on it just remains to prove that i cannot be $n+1$, which is true since it is in the complement of `lens_pair` (`succ_neq` ($\mathtt{n} : [\![\mathtt{n}+1]\!]$)).

9 Parallel composition

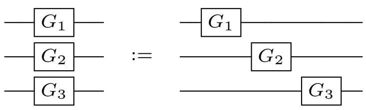
In this section, we extend our theory with noncommutative and commutative monoids of the sequential and parallel compositions of morphisms. Thanks to quantum state currying, we have been able to define focusing and composition of circuits without relying on the Kronecker product. This also means that parallel composition is not primitive in this system. Thanks to `focusC`, morphisms applied through disjoint lenses do commute, but it is harder to extend this to an n-ary construct, as done in CoqQ [20]. Yet it is possible to define parallel composition using MathComp *big operators* by defining a new notion of commuting composition of morphisms. Note that big operators on monoids require axioms based on propositional equality, rather than the extensional equality of morphisms, so in this section we assume functional extensionality and proof irrelevance, which allows us to use lemma `morP` of Section 5.

As a first step, we define the noncommutative monoid of morphisms, using the sequential (*vertical* in category-theoretic terminology) composition as monoid operation and the identity morphism as unit element. Registering the associativity and unitality laws with Hierarchy Builder [6], allows one to use the corresponding m-ary big operator.

```
HB.instance Definition _ := Monoid.isLaw.Build on •_{n,n,n} and idmor_n.
Definition compn_mor m (F : [m] → endo_n) (P : pred [n]) :=
  \big[•_{n,n,n}/idmor_n]_(i<n, P i) F i.
```

By itself, it just allows us to define some circuits in a more compact way. It will also allow us to connect with the commutative version.

The parallel (*horizontal*) composition of morphisms is derived from vertical composition, in the case where the morphisms focused in a circuit have disjoint supports.



(Turn the diagram 90 degrees clockwise to see that it is a "horizontal" composition.)

We construct a commutative monoid whose operation is the horizontal composition, by reifying the notion of focused morphism (inside an n-qubit circuit), using the corresponding lens to express the support.

```
Record foc_endo_n := {(m, ℓ, e) : ℕ × lens_{n,m} × endo_m | ℓ is monotone}.
```

The monotonicity of ℓ in focused morphisms is demanded for the canonicity and strictness of their compositions. The arity m of the morphism is existentially quantified.

The actual Coq definition of `foc_endo` has four fields `foc_m`, `foc_l`, `foc_e`, and `foc_s`, the first three corresponding to m, ℓ, e above, and the last one being the proof that ℓ is monotone. We define `mkFendo`, a "smart constructor" that factorizes a given lens (`lens_basis` and `lens_perm` in Section 3) into its basis (whose monotonicity proof being `lens_sorted_basis`) and permutation to build a focused morphism.

```
Definition mkFendo_{n,m} (ℓ : lens_{n,m}) (G : endo_m) :=
  {| foc_s := lens_sorted_basis ℓ; foc_e := focus (lens_perm ℓ) G |}.
```

Focused morphisms come with both a unit element and an annihilating (zero) element.

```
Definition id_fendo  := mkFendo (lens_empty n) (idmor I K 0).
Definition err_fendo := mkFendo (lens_id n) (nullmor n n).
```

The unit element `id_fendo` has an empty support, and the zero element `err_fendo` has a full support.

A focused morphism can be used as an ordinary morphism at arity n by actually focusing the morphism field e along the lens field ℓ (field projections `foc_l` and `foc_e` are denoted by $.\ell$ and $.e$).

```
Definition fendo_mor (Φ : foc_endo) : endo_n := focus Φ.ℓ Φ.e.
```

We can then define commutative composition `comp_fendo`.

```
Definition par_comp_{p,q} (F : endo_p) (G : endo_q) : endo_{p+q} :=
  (focus lens_left F) • (focus lens_right G)
Definition comp_fendo (Φ Ψ : foc_endo) :=
  { mkFendo (Φ.ℓ ++ Ψ.ℓ : lens_{n,Φ.m+Ψ.m}) (par_comp Φ.e Ψ.e)    if Φ.ℓ and Ψ.ℓ are disjoint
  { err_fendo                                                     otherwise
```

To make composition commutative, we return the zero element whenever the lenses of the two morphisms are not disjoint. If they are disjoint, we return their composition, using the union of the two lenses. We require lenses to be monotone to guarantee associativity.

Using this definition of commutative composition, we can declare the commutative monoid structure on focused morphisms and define their m-ary parallel composition. When the lenses are pairwise disjoint, it coincides with `compn_mor`.

```
HB.instance Definition _ := Monoid.isComLaw.Build on comp_fendo and id_fendo.
Variables (m : ℕ) (F : [m] → foc_endo) (P : pred [m]).
Definition compn_fendo := \big[comp_fendo/id_fendo]_(i<m, P i) F i.
Hypothesis Hdisj : ∀i, j, i ≠ j → (F i).ℓ and (F j).ℓ are disjoint.
Theorem compn_mor_disjoint : compn_mor (fendo_mor ∘ F) P = fendo_mor compn_fendo.
```

To exemplify the use of this commutative monoid, we proved that the circuit that consists of $\lfloor n/2 \rfloor$ swap gates that swap the ith and $(n-i-1)$th of n qubits returns a reversed state (Figure 8).

```
Lemma rev_ord_neq_n (i : [⌊n/2⌋]) : (i : [n]) ≠ (n − i − 1 : [n]).
Definition rev_circuit n : endo_n :=
  compn_mor (i ↦ focus (lens_pair (rev_ord_neq i)) swap) xpredT.
Lemma rev_circuit_ok : ∀(i : [n]),
  proj (lens_single (n − i − 1 : [n])) (rev_circuit n σ) = proj (lens_single i) σ.
```

Here `rev_ord_neq` produces an inequality in $[n]$, which we can use to build the required pair lens to apply `swap`.

10 Related works

There are many works that aim at the mechanized verification of quantum programs [14]. Here we only compare with a number of like-minded approaches, built from first principles, i.e. where the formalization includes a model of computation based on unitary transformations, which justifies the proof steps.

Qiskit [16] is a framework for writing quantum programs in Python. While it does not let one write proofs, it has the ability to turn a circuit into a gate, allowing one to reuse it in other circuits, so that it has definitional compositionality.

QWIRE [15] and SQIR [11] define a quantum programming language and its Hoare logic in Coq, modeling internally computation with matrices and Kronecker products. QWIRE and SQIR differ in their handling of variables: in QWIRE they are abstract, handled through higher-order abstract syntax, but in SQIR, which was originally intended as an intermediate language for the compilation of QWIRE, they are concrete natural numbers, denoting indices of qubits. The authors note in their introduction [11] that "[abstract variables] necessitate a map from variables to indices, which we find confounds proof automation". They go on remarking that having a distinct semantics for pure quantum computation, rather than relying only on the density matrices needed for hybrid computations, considerably simplifies proofs; this justifies our choice of treating specifically the pure case. While QWIRE satisfies definitional compositionality, this is not the case for SQIR, as circuits using fixed indices cannot be directly reused. We have not proved enough programs to provide a meaningful comparison, yet it is noteworthy that our proof of GHZ, which uses virtually no automation, is about half the size of the proof in SQIR [11]. The main difference is that we are able to solve combinatorics at the level of lenses, while they have to work all along with a symbolic representation of matrices, that is a linear combination of matrix units (Dirac's notation), to avoid working directly on huge matrices.

VyZX [12, 13] formalizes the ZX-calculus in Coq, on top of SQIR. Its goal is to prove graph-rewriting rules, and ultimately to build a verified optimizer for the ZX-calculus. However, as they state themselves, the graphical nature of the calculus appears to be a major difficulty, and only restricted forms of the rules are proved at this point. Since the ZX-calculus itself enjoys compositionality, albeit at the graph level, this is a promising line of work. It would be interesting to see if our approach can make proving such graph-rewriting rules easier. As preliminary experiment, we have proved the triangular identity involving a cup and a cap, by defining an asymmetric version of focusing. More generally, finding a nice way to compose graphs is essential, and concepts such as lenses could have a role there.

CoqQ [20] builds a formalized theory of Hilbert spaces and n-ary tensor products on top of MathComp, adding support for the so-called *labelled Dirac notation*. Again they define a Hoare logic for quantum programs, and are able to handle both pure and hybrid computations. While the labelled Dirac notation allows handling commutation comfortably, it does not qualify as compositional, since it is based on a fixed set of labels, i.e. one cannot mix programs if they do not use the same set of labels.

Unruh developed a quantum Hoare logic and formalized it in Isabelle, using a concept of *register* [18] for which he defines a theory, including operations such as taking the complement of a register. His registers in some meaning generalize our `focus` function, as they allow focusing between arbitrary types rather than just sets of qubits. Since one can compose registers, his approach is compositional, for both definitions and proofs, and the abstraction overhead is avoided through automation. However, while each application of `focus` to a lens can be seen as a register, he has not separated out a concrete combinatorics based on finite objects similar to our notion of lens.

In a slightly different direction, Qbricks [3] uses the framework of *path-sums* to allow the automatic proof of pure quantum computations. The notion of path is more expressive than that of computational basis state, and allows one to represent many unitary transformations as maps from path to path, making calculations easier. It would be interesting to see whether it is possible to use them in our framework.

Most approaches above support not only pure quantum computation but also hybrid quantum-classical computation. While we have concentrated here on pure computation, we have already extended our approach to the density-matrix interpretation required to support hybrid computations, and verified that it commutes with focusing. Practical applications are left to future work.

Note also that, while some of the above works use dependent types to represent matrix sizes for instance, they all rely on ways to hide or forget this information as a workaround. On the other hand, our use of dependent types is strict, only relying on statically proved cast operators to adjust types where needed, yet it is lightweight enough for practical use.

Some other aspects of our approach can be related to programming language theory. For instance, the way we shift indices during currying is reminiscent of De Bruijn indices, and our `merge` operation shifts indices in the precise same way as the record concatenation defined in the label-selective λ-calculus [8]. This suggests that our currying of quantum states is actually similar to the currying occurring in that calculus.

11 Conclusion

We have been able to build a compositional model of pure quantum computation in Coq, on top of the MathComp library, by using finite functions, lenses, and focusing. We have applied the development to prove the correctness of several quantum circuits. An interesting remark is that, while we started from the traditional view of seeing quantum states as tensor products, our implementation does not rely on the Kronecker product for composing transformations. Since the Kronecker product of matrices can be cumbersome to work with, this is a potential advantage of this approach.

Many avenues are open for future work. First we need to finish the proof of Shor's code, this time for erroneous channels; paper proofs are simple enough but the devil is in the details. Next, building on our experience, we would like to formalize and abstract the algebraic theory of lenses. Currently we rely on a large set of lemmas developed over more than a year, without knowing their interdependencies; such a theory would have both theoretical and practical implications. Third, we are interested in the category-theoretic aspects of this approach, and would like to give an account of `focus`, explaining both the relation between a lens and its action, and the structural properties of focusing.

References

1 F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981. doi:10.1145/319628.319634.

2 Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863543.1863572.

3 Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems, ESOP 2021*, volume 12648 of *Lecture Notes in Computer Science*, pages 148–177, Cham, March 2021. Springer International Publishing. doi:10.1007/978-3-030-72019-3_6.

4 Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70583-3_25.

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

6 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.34.

7 J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. doi:10.1145/1232420.1232424.

8 Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994. doi:10.1145/174675.174434.

9 Jacques Garrigue and Takafumi Saikawa. QECC: Quantum Computation and Error-Correcting Codes. Software, swhId: `swh:1:dir:d4d158675180ee276e730bd7f67a9122a6472eb3` (visited on 2024-08-21). URL: `https://github.com/t6s/qecc`.

10 Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going beyond bell's theorem. In Menas Kafatos, editor, *Bell's Theorem, Quantum Theory and Conceptions of the Universe*, pages 69–72. Springer Netherlands, Dordrecht, 1989. `doi:10.1007/978-94-017-0849-4_10`.

11 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. `doi:10.1145/3434318`.

12 Adrian Lehmann, Ben Caldwell, and Robert Rand. VyZX : A vision for verifying the ZX calculus, 2022. `doi:10.48550/arXiv.2205.05781`.

13 Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. VyZX: Formal verification of a graphical quantum language, 2023. `doi:10.48550/arXiv.2311.11571`.

14 Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools and challenges, 2022. `doi:10.1145/3624483`.

15 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 846–858, 2017. `doi:10.1145/3009837.3009894`.

16 Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. `doi:10.5281/zenodo.2573505`.

17 Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, October 1995. `doi:10.1103/PhysRevA.52.R2493`.

18 Dominique Unruh. Quantum and classical registers. *CoRR*, abs/2105.10914, 2021. `doi:10.48550/arXiv.2105.10914`.

19 Mingsheng Ying. *Foundations of Quantum Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016. `doi:10.1016/C2014-0-02660-3`.

20 Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. CoqQ: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. `doi:10.1145/3571222`.