Typed compositional quantum computation with lenses

Jacques Garrigue
Nagoya University
Graduate School of Mathematics
Japan

Takafumi Saikawa
Nagoya University
Graduate School of Mathematics
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 us 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.

Keywords: quantum programming, lens, Coq, MathComp

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 [12]. 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 [7, 11, 13], or more abstractly using string diagrams representing computations in a symmetric monoidal category [2]. 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*, 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 duplicated and instantiated as part of another circuit.

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 gives a semantical representation of 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, which incurs very little overhead. It is not only intellectually satisfactory, but also offers the potential of making proofs more scalable without having to 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. *Lenses* 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 [3]. We choose a simpler view in which lenses are just injections between two finite sets of wires. *Finite functions* over n-tuples of bits can encode a n-qubit quantum state. *Currying* of such functions, along a lens, provides a direct representation of tensor products. *Polymorphism* appears to be sufficient 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 on. We were also able to prove a number of examples, such as the correctness of Shor coding [10] (formalized for the first time, albeit only for an error-free channel at this point), the Greenberger-Horne-Zeilinger (GHZ) state preparation [4], and the reversed list circuit [13].

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 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 $\{0,\ldots,n-1\}$[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

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

States other than basis states are again superpositions, which are linear combinations of norm 1 belonging to the n-ary tensor power of $\mathbb{C}^2$

$$(\mathbb{C}^2)^{\otimes n} := \mathbb{C}^{2^n}$$

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

[1]This notation is a variant of $[n]$ standing for the integer interval $[0,n]$, used for the simplex category, which includes n while $[n[$ excludes it.

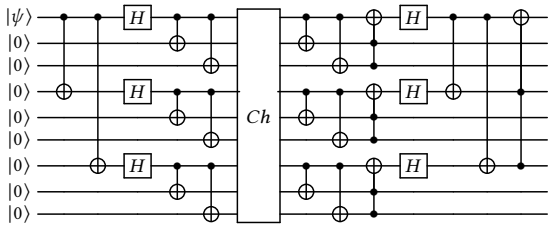


Figure 1. Shor's 9-qubit code

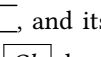
2.2 Unitary transformations

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 $\langle a \mid a\rangle$, 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 often larger subcircuits being placed over those wires that they act on. A quantum state is input from the left end of a circuit, acted on by gates from left to right, and output from the right end. As an example, we show Shor's 9-qubit error correction code (Figure 1).

The primitive operations in a quantum circuit are quantum gates. In the above Shor's code, three kinds of gates appear, namely Hadamard —[H]—, Controlled Not (CNOT)  , and its three-qubit variant Toffoli  . The large box [Ch] denotes an arbitrary unitary transformation modelling a possibly erroneous channel. The above gates can be expressed as matrices with respect to the standard basis (lexicographically ordered), for example:

$$—[H]— = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad \text{  } = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

A gate can be composed in a circuit by, first padding irrelevant wires by taking the Kronecker product with an

identity matrix, and second sandwiching the padded gate with the action of a permutation σ on the index of tensors and its inverse, whose matrix representation (tensor permutation matrix) we denote as $U_{2^{\otimes n}}(\sigma)$ and $U_{2^{\otimes n}}(\sigma^{-1})$ [9]. For example, to describe the leftmost CNOT gate in the Shor's code, we first pad CNOT with $I_{2^7} = I_{128}$ and apply the tensor permutation matrix that exchanges the second and fourth qubits.

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

Here (24) denotes a permutation between 2 and 4.

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.

We want to map the m ports 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 lens_{n,m} := {ℓ : [n[^m | uniq ℓ}.
```

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 to omit them.

The operation using a lens to feed a part of the input data to a subcircuit (or subprogram) is called *focusing*. The following operations on lenses are basic and required to define focusing.

```
Definition lensC_{n,m} : lens_{n,m} → lens_{n,n−m}.
Definition extract_{T,n,m} : lens_{n,m} → T^n → T^m.
Definition merge_{T,n,m} : lens_{n,m} → T^m → T^{n−m} → T^n.
```

`lensC ℓ` is the complementary lens, which is the unique monotone bijection from $[n-m[$ to $[n[\setminus \mathrm{Im}(\ell)$. We will write $\ell^{\complement}$ for `lensC ℓ`. `extract ℓ v` is the image of v along ℓ. `merge ℓ v w` is the inverse of `extract` [2], that is,

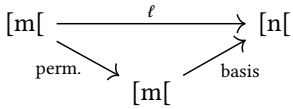
```
Lemma merge_extract : ∀(v : T^n),
  merge (extract ℓ v) (extract ℓ^∁ v) = v.
```

We show the classical case of focusing (`focus1`) as an example. In this case, data is represented by direct products, whose elements are tuples, readily manipulated by `extract` and `merge`.

```
Definition focus1_{T,n,m} (ℓ : lens_{n,m}) (f : T^m → T^m) :
  T^n → T^n :=
  s ↦ merge (f (extract ℓ s)) (extract ℓ^∁ s).
Lemma focus1_in : ∀T, n, m, ℓ, f,
  (extract ℓ) ∘ (focus1 ℓ f) = f ∘ (extract ℓ).
```

`focus1` cannot be directly applied to quantum state transformations, where the state is not represented by direct products but by tensor products. We will see in the next sections that its quantum version can be defined through currying and uncurrying of quantum states, which can both be in turn defined using the three previous operations.

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



Namely, we have the following functions and laws:

```
Definition lens_comp_{n,m,p} :
  lens_{n,m} → lens_{m,p} → lens_{n,p}.
Definition lens_basis_{n,m} : lens_{n,m} → lens_{n,m}.
Definition lens_perm_{n,m} : lens_{n,m} → lens_{m,m}.
Lemma lens_basis_perm : ∀n, m, (ℓ : lens_{n,m}),
  lens_comp (lens_basis ℓ) (lens_perm ℓ) = ℓ.
Lemma mem_lens_basis :
  ∀n, m, (ℓ : lens_{n,m}), lens_basis ℓ =_i ℓ.
```

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

4 Composing pure quantum computation

We want to be able to define quantum circuits part by part and compose them into larger ones. A lens $\ell : \mathtt{lens}_{n,m}$ can be used to compose an m-ary quantum circuit into an n-ary one according to the following observations. One is that a function from 2^n to $\mathbb{C}$ can also be seen as function from 2^m to $\mathbb{C}^{2^{n-m}}$, which itself is a vector space.

$$\mathbb{C}^{2^n} \cong \left(\mathbb{C}^{2^{n-m}}\right)^{2^m}$$

Another is that any linear transformation G on $\mathbb{C}^{2^m}$ can be represented by a matrix, so that it can also be multiplied

[2] In the database literature, the merge operation usually takes the whole input state, but here only the unmodified part is required.

to vectors of T^{2^m} for an arbitrary complex vector space T. We are thus led to endow such G with a polymorphic type of linear transformations indexed by T.

$$G : \forall T : \text{vector sp.}, T^{2^m} \xrightarrow{\text{linear}} T^{2^m}$$

We name the above isomorphism `curry` and its inverse `uncurry`. Along this isomorphism, a gate G can be extended to a larger number of qubits, to become composable in a circuit.

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

The type of G tells 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.

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

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

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

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

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

We proved conversely that this naturality implies the uniqueness of the matrix. Our definition of quantum gates is based on naturality.

5 Defining quantum gates

Using MathComp, we can easily present the concepts described in the previous section. 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 the previous section. It is encoded as a function type $T^{\widehat{n}}$ from n-tuples of some finite type I to a type T (in practice this finite type will always be $[2[= \{0,1\}$).

```
Variables (I : finite type) (dI : I) (K : field).
Definition T^n := I^n -finite-> T.
Definition dpmap_{m,T1,T2} (φ : T1 → T2) (s : T1^m) : T2^m :=
  (v : I^m) ↦ φ (s(v)).
```

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 $\texttt{dpmap}\ \varphi : T_1^{\widehat{n}} \to T_2^{\widehat{n}}$, 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 morlin_{m,n} := ∀ T : Vect_K, T^m -linear-> T^n.
Definition naturality m n (G : morlin_{m,n}) :=
  ∀(T1 T2 : Vect_K), (φ : T1 -linear-> T2),
    (dpmap φ) ∘ (G T1) = (G T2) ∘ (dpmap φ).
Record mor_{m,n} := {G : morlin_{m,n} | naturality G}.
Notation endo_n := (mor_{n,n}).
Definition unitary_mor_{m,n} (G : mor_{m,n}) :=
  ∀s, t, ⟨G_{K^1} s | G_{K^1} t⟩ = ⟨s | t⟩.
```

A crucial fact we rely on here is that, for any vector space T, MathComp defines the vector space of the finite functions valued into it [3], so that $T^{\widehat{n}}$ is a vector space. 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 $\texttt{dpbasis}\ v$, where v is the index of the only 1 in the vector.

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

This representation of states allows 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). We then use `ket_bra` to define the CNOT gate as a sum of matrix units, so that it can be fed to `tsmor` to obtain a morphism.

```
Definition tsmor_{m,n} : ((K^1)^m)^n → mor_{m,n}.
Definition ket_bra_{m,n} (k : (K^1)^m) (b : (K^1)^n) : ((K^1)^m)^n
  := v ↦ (k v) · b.
Definition cnot : ((K^1)^2)^2 :=
  ket_bra |0,0⟩ |0,0⟩ + ket_bra |0,1⟩ |0,1⟩ +
  ket_bra |1,0⟩ |1,1⟩ + ket_bra |1,1⟩ |1,0⟩.
```

Here $|i_1, \ldots, i_n\rangle$ is a notation for the computational basis vector `(dpbasis [tuple i1; ...; in])`.

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

```
Lemma naturalityP : ∀m, n, (G : morlin_{m,n}),
  naturality G ⟷ ∃M, ∀T, s, G_T s = (tsmor M)_T s.
```

[3] The proof script is a bit more general, as it works for (left) modules over rings (`lmodType`).

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.

```
Lemma morP : ∀m, n, (F, G : mor_{m,n}),
                (∀T, s, F_T s = G_T s) ⟷ F = G.
```

While our development distinguishes between the two equalities, only assuming those axioms where needed, in this paper we will not insist on the distinction, and just write $F = G$ for extentional equality too.

6 Building circuits

The currying defined in section 4 allows 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 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, and add a proof of naturality.

```
Variables (n m : ℕ) (ℓ : lens_{n,m}).
Definition curry_T (s : T^n) : (T^{n-m})^m :=
  (v : I^m) ↦ ((w : I^{n-m}) ↦ s (merge ℓ v w)).
Definition uncurry_T (s : (T^{n-m})^m) : T^n :=
  (v : I^n) ↦ s (extract ℓ v) (extract ℓ^C v).
Definition focuslin (G : endo_m) : morlin_{n,n} :=
  ΛT. (uncurry ℓ)_T ∘ G_{T^{n-m}} ∘ (curry ℓ)_T.
```

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

```
Lemma focus_comp : ∀n, m, (F, G : endo_m), (ℓ : lens_{n,m}),
  focus l (F • G) = (focus ℓ F) • (focus ℓ G).
Lemma focusM : ∀n, m, p,
  ∀(ℓ : lens_{n,m}), (ℓ' : lens_{m,p}), (G : endo_p),
  focus (lens_comp ℓ ℓ') G = focus ℓ (focus ℓ' G).
Lemma focusC : ∀n, m, p,
  ∀(ℓ : lens_{n,m}), (ℓ' : lens_{n,p}), (F : endo_m), (G : endo_n),
  ℓ and ℓ' disjoint →
  (focus ℓ F) • (focus ℓ' G) = (focus ℓ' G) • (focus ℓ F).
Lemma unitary_comp : ∀m, n, p, (F : mor_{n,p}), (G : mor_{m,n}),
  unitary_mor F →
  unitary_mor G → unitary_mor (F • G).
Lemma unitary_focus : ∀n, m, (ℓ : lens_{n,m}), (G : endo_m),
  unitary_mor G → unitary_mor (focus ℓ G).
```

Since all circuits can be built from unitary basic gates using sequential composition and focus, lemmas unitary_comp and unitary_focus are sufficient to guarantee unitarity.

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 dpsingle_{k,T} (v : I^k) (s : T) : T^k :=
  (v' : I^k) ↦ { s  if v = v'
               { 0  otherwise .
Definition dpmerge v : (K^1)^m --linear--> (K^1)^n :=
  uncurry ℓ ∘ dpmap (dpsingle (extract ℓ^C v)).
Lemma focus_dpbasis : ∀(G : endo_m), (v : I^n),
  (focus ℓ G)_{K^1} (dpbasis v) =
  dpmerge v (G_{K^1} (dpbasis (extract ℓ v))).
Lemma dpmerge_dpbasis : ∀(v : I^n), (v' : I^n),
  dpmerge v (dpbasis v') =
  dpbasis (merge ℓ v' (extract ℓ^C v)).
Lemma decompose_scaler : ∀k, (s : (K^1)^n),
  s = ∑_{v:I^k} s(v) · dpbasis v.
```

The function dpsingle is a variant of dpbasis that applies to nested vector spaces. While the definition of dpmerge may seem complex, it is only introduced and eliminated through the following two lemmas. 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 (dpbasis v') 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 : ∀v, v', (H : i ∈ ℓ),
  (merge ℓ v v')[i] = v[lens_index H].
Lemma tnth_extract : ∀T, (v : T^n), (j : [m[),
  (extract ℓ v)[j] = v[ℓ[j]].
Lemma mem_lensC : (i ∈ ℓ^C) = (i ∉ ℓ).
Lemma mem_lens_comp :
 ∀p, (ℓ' : lens_{m,p}), (H : i ∈ ℓ),
  (i ∈ lens_comp ℓ ℓ') = (lens_index H ∈ ℓ').
```

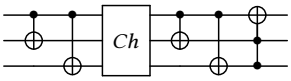


Figure 2. Bit-flip code

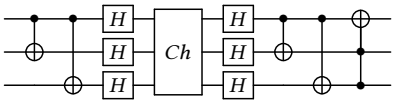


Figure 3. Sign-flip code

```
End lens_index.
Lemma tnth_mergeC :
 ∀n, m, (ℓ : lens_{n,m}), i, v, v', (H : i ∈ ℓ^C),
   (merge ℓ v v')[i] = v'[lens_index H].
```

Here we write $v[i]$ for `tnth` v i, i.e. the ith component of the tuple v. 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 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 $\mathtt{Co} = \mathbb{C}^1$. The indices are now in $\mathtt{I} = [2[= \{0, 1\}$. 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 application of the above definition of compositition, we first show how Shor's 9-qubit error correction code can be presented in our framework.

Shor's code is known by the circuit diagram in Figure 1. This circuit consists of two smaller components: bit-flip and sign-flip codes (Figures 2 and 3).

We can see in Shor's code that three bit-flip codes are placed in parallel, and sandwiched by one sign-flip code. This can be expressed straightforwardly as the following Coq code.

```
Notation tsapp ℓ G := (focus ℓ (tsmor G)).
Definition bit_flip_enc : endo_3 :=
  tsapp [lens 0; 2] cnot • tsapp [lens 0; 1] cnot.
Definition bit_flip_dec : endo_3 :=
  tsapp [lens 1; 2; 0] toffoli • bit_flip_enc.
Definition hadamard_3 : endo_3 :=
  tsapp [lens 2] hadamard • tsapp [lens 1] hadamard
  • tsapp [lens 0] hadamard.
Definition sign_flip_dec := bit_flip_dec • hadamard3.
Definition sign_flip_enc := hadamard3 • bit_flip_enc.
Definition shor_enc : endo_9 :=
  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 : endo_9 := ...
```

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

```
Theorem shor_code_id : ∀i,
 (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 tsmor_cnot : ∀i, j, tsmor cnot |i, j⟩ = |i, i + j⟩.
Lemma tsmor_toffoli_00 : ∀i, tsmor toffoli |0, 0, i⟩ = |0, 0, i⟩.
Lemma hadamardK : ∀T, involutive (tsmor hadamard)_T.
Lemma bit_flip_enc_ok : ∀i, j, k,
  bit_flip_enc |i, j, k⟩ = |i, i + j, i + k⟩.
Lemma bit_flip_toffoli :
  (bit_flip_dec • bit_flip_enc) =
  tsapp [lens 1; 2; 0] toffoli.
Lemma sign_flip_toffoli :
  (sign_flip_dec • sign_flip_enc) =
  tsapp [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. `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 beginnings of `bit_flip_enc_ok` and `shor_code_id`, in figures 4 and 5, 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.

Look first at figure 4. Simplifying on line 2 reveals the application of the two `cnot` gates. Rewriting with `focus_dpbasis` on line 4 applies the first gate directly to a basis vector. `simpl_extract` on line 7 is a helper tactic that 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 `tsmor_cnot`. 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

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

Figure 4. Excerpt of interactive proof of `bit_flip_enc_ok`

```
1    (shor_dec • shor_enc) ¦i,0,0,0,0,0,0,0,0⟩
2  rewrite /=.
3  = focus [lens [0; 3; 6] sign_flip_dec (... (focus [lens 0; 3; 6] sign_flip_enc ¦i,0,0,0,0,0,0,0,0⟩) ...)
4  transitivity (focus [lens 0; 3; 6] (sign_flip_dec • sign_flip_enc) ¦i,0,0,0,0,0,0,0,0⟩).
5  rewrite focus_comp /= focus_dpbasis.
6  = focus [lens [0; 3; 6] sign_flip_dec (...
7      (dpmerge [lens 0; 3; 6] (shor_input i)
8         (sign_flip_enc (dpbasis (extract [lens 0; 3; 6] (shor_input i))))) ...)
9  set sfe := sign_flip_enc _ _.
10 = focus [lens [0; 3; 6] sign_flip_dec (... (dpmerge [lens 0; 3; 6] (shor_input i) sfe) ...)
11 rewrite (decompose_scaler sfe) !linear_sum /=.
12 = \sum_(t : 3.-tuple I) focus [lens [0; 3; 6] sign_flip_dec (...
13     (dpmerge [lens 0; 3; 6] (shor_input i) (sfe t *: dpbasis t)) ...)
14 apply eq_bigr => t _.
15 rewrite !linearZ_LR /= dpmerge_dpbasis.
16   sfe t *: focus [lens [0; 3; 6] sign_flip_dec (...
17     (dpbasis (merge [lens 0; 3; 6] t (extract (lensC [lens 0; 3; 6]) (shor_input i)))) ...)
18 = sfe t *: focus [lens [0; 3; 6] sign_flip_dec
19     (dpbasis (merge [lens 0; 3; 6] t (extract (lensC [lens 0; 3; 6]) (shor_input i))))
20 congr (_ *: focus _ sign_flip_dec _ _).
21 case: t => -[|a [|b [|c []]]] Ht //=.
22 simpl_merge.
23   focus [lens 0; 1; 2] bit_flip_dec (...
24     (focus [lens 6; 7; 8] bit_flip_enc ¦ a, 0, 0, b, 0, 0, c, 0, 0 ⟩) ...)
25 = ¦ a, 0, 0, b, 0, 0, c, 0, 0 ⟩
```

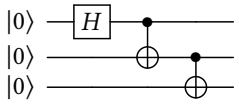
Figure 5. Excerpt of interactive proof of `shor_code_id`

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 `tsmor_cnot` uses a property of the specific gate.

The proof of `shor_code_id` in figure 5 is more involved as the Hadamard gates introduce superpositions. This time, simplification on line 2 results in a large expression composing all the subcircuits involved in `shor_dec` and `shor_enc`. `transitivity` on line 4 changes the right hand side of the goal to an expression identical to the left-hand side, but omitting the ... parts which contain the bit-flip encoders and decoders, once we have rewritten it with `focus_comp`. The following steps will rewrite both the left-hand side and the right-hand side of the goal in the same way, but we only show the left-hand side. Again we use `focus_dpbasis` on line 5, but foreseeing that `sign_flip_enc` will result in a large superposition, we name the resulting vector as `sfe` on line 9, and decompose it with `decompose_scaler` on line 11. This results in the sum $\sum_{t \in [2[^3} \texttt{sfe}_t \, |t\rangle$. We push the sum outside by linearity (`linear_sum`), resulting in the expression at line 12. Since the

left-hand side and the right-hand side share the same shape, we can apply `eq_bigr`, and prove an equality between the bodies of the two sums. From line 16 on we write both sides of the goal. We push the scaling factor $\texttt{sfe}_t$ out, so that we can use `dpmerge_dpbasis`. We then use congruence to get rid of the outer `focus [lens 0; 3; 6] sign_flip_dec` part, and decompose the bit-vector t into 3 bits a, b, c, so that the input to the `bit_flip_enc` circuits is now $|a, 0, 0, b, 0, 0, c, 0, 0\rangle$. The rest of the proof uses various commutations to pair the bit-flip encoders and decoders, and finally applies `bit_flip_toffoli`, `sign_flip_toffoli` and $\texttt{tsmor\_toffoli}_{00}$ to conclude.

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 $|1\rangle^{\otimes n}$. 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 $|0\rangle^{\otimes n}$. For instance, for 3 qubits this gives the following circuit.



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 => tsmor hadamard
  | m.+1 =>
    tsapp (lens_pair (succ_neq (m:[m.+1[))) cnot •
    focus (lensC (lens_single (m.+1:[m.+2[))) (ghz m)
  end.
```

The definition works by composing `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 as follows:

```
Definition ghz_state n : (ℂ^1)^(n+1) :=
  (1 / Num.sqrt 2)%:C *:
  (dpbasis [tuple 0 | i < n + 1] +
   dpbasis [tuple 1 | i < n + 1]).
```

where `%:C` injects $1/\sqrt{2}$ from $\mathbb{R}$ into $\mathbb{C}$, and $[\texttt{tuple } F\, i \mid i < n]$ denotes the n-tuple whose ith element is $F\, i$. Then the correctness property is:

```
Lemma ghz_ok : ∀n,
  ghz n (dpbasis [tuple 0 | i < 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 6, 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. Lemma `eq_from_tnth` on line 4 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 $i \in \texttt{lens\_pair}(\texttt{succ\_neq}(n : [n + 1[))$, 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 12 a goal on which we can use congruence, and since $\texttt{flip } 0 = 1$, we conclude with `eq_lens`. The second case, when $i \notin \texttt{lens\_pair}(\texttt{succ\_neq}(n : [n + 1[))$, 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 (n : [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 [13]. 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 to use lemma `morP` of section 5.

As a first step, we define the non-commutative monoid of morphisms, using the sequential (*vertical* in category-theoretic terminology) composition as monoid operation and the identity morphism as unit element. By declaring its associativity and unitality laws as a canonical structure, we can use the corresponding m-ary big operator.

```
Variable (n : ℕ).
Canonical comp_monoid :=
  Monoid.Law 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 to define some circuits in a more compact way. It will also allow to connect with the commutative version.

```
    merge (lens_pair (succ_neq (n : [n.+1[])) [tuple 1; flip 0]
      (extract (lensC (lens_pair (succ_neq (n : [n.+1[])))
         [tuple if i != n.+1 then 1 else 0 | i < n.+2]) = [tuple 1 | _ < n.+2]
  apply eq_from_tnth => i; rewrite [RHS]tnth_mktuple.
  case/boolP: (i \in lens_pair (succ_neq (n : [n.+1[])) => Hi.
    Hi : i \in lens_pair (succ_neq (n : [n.+1[])
    ============================
    tnth (merge (lens_pair (succ_neq (n : [n.+1[])) [tuple 1; flip 0]
            (extract (lensC (lens_pair (succ_neq (n : [n.+1[])))
               [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
  rewrite tnth_merge -[RHS](tnth_mktuple (fun=>1) (lens_index Hi)).
    tnth [tuple 1; flip 0] (lens_index Hi) = tnth [tuple 1 | _ < 2] (lens_index Hi)
  by congr tnth; eq_lens.
    Hi : i \notin lens_pair (succ_neq (n : [n.+1[])
    ============================
    tnth (merge (lens_pair (succ_neq (n : [n.+1[])) [tuple 1; flip 0]
            (extract (lensC (lens_pair (succ_neq (n : [n.+1[])))
               [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
  rewrite -mem_lensC in Hi.
  rewrite tnth_mergeC tnth_extract tnth_mktuple.
    Hi : i \in lensC (lens_pair (succ_neq (n : [n.+1[])))
    ============================
    (if tnth (lensC (lens_pair (succ_neq (n : [n.+1[]))) (lens_index Hi) < n.+1 then 1 else 0) = 1
  rewrite tnth_lens_index ifT //.
    i != n.+1
  move: Hi; rewrite mem_lensC !inE negb_or => /andP[] _.
    i != lift ord0 (n : [n.+1[]) -> i != n.+1
  by apply/contra => /eqP Hj; apply/eqP/val_inj; rewrite /= bump0n.
```

Figure 6. Excerpt of interactive proof of ghz_ok

The parallel (*horizontal*) composition of morphisms is derived from vertical composition. We can construct a commutative monoid whose operation is the horizontal composition, by introducing a notion of focused morphism.

```
Variable (n : ℕ).
Record foc_endo :=
  {(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 of the lens (and of the morphism) is existentially quantified.

`foc_endo` in the Coq code 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_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) :=
```

$$\begin{cases} \texttt{mkFendo}\ (\Phi.\ell \mathbin{+\!\!+} \Psi.\ell : \texttt{lens}_{n,\Phi.m+\Psi.m})\ (\texttt{par\_comp}\ \Phi.e\ \Psi.e) & \\ \qquad\qquad \text{if } \Phi.\ell \text{ and } \Psi.\ell \text{ are disjoint} & \\ \texttt{err\_fendo} \qquad\qquad \text{otherwise} & \end{cases}$$

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.

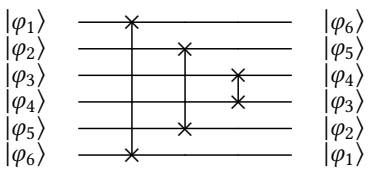


Figure 7. Reversed state circuit (e.g. for six qubits)

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`.

```
Canonical compf_monoid :=
  Monoid.Law on comp_fendo and id_fendo.
Canonical compf_comoid :=
  Monoid.ComLaw on comp_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 7).

```
Lemma rev_ord_neq n (i : [⌊n/2⌋[) :
  (i : [n[) ≠ (n - i - 1 : [n[).
Definition rev_circuit n : endo_n :=
  compn_mor (i ↦ tsapp (lens_pair (rev_ord_neq i)) swap)
            xpredT.
Lemma rev_circuit_ok : ∀n, (i : [n[), s,
  proj (lens_single (n - i - 1 : [n[)) (rev_circuit n s) =
  proj (lens_single i) s.
```

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 [6]. 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. Many approaches support not only pure quantum computation but also hybrid quantum-classical computation, and allow one to use a form of Hoare logic to prove properties. These are features we have not yet considered.

Qiskit [8] is a framework for writing quantum programs in Python. While it does not allow to write proofs, it has the ability to turn a circuit into a gate, allowing to reuse it in other circuits, so that it has definitional compositionality.

QWIRE [7] and SQIR [5] 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 [5] 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 seems to justify 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, appears to be shorter than the proof in SQIR [5]. The main difference seems to be 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.

CoqQ [13] 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* [11] 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 [1] 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 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.

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 work around. 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 appears to be lightweight.

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 lives 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. Finally, while we have only presented here the symmetric version of `focus`, one can also build a function `asym_focus` taking two lenses, and going from $\texttt{mor}_{m,p}$ to $\texttt{mor}_{m+n,p+n}$. It has interesting applications to represent computations seen as string diagrams [2], yet we are still looking for the right lemmas to work with it.

References

[1] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-Building Quantum Programs. In *Programming Languages and Systems, ESOP 2021 (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6 arXiv:2003.05841

[2] Bob Coecke and Aleks Kissinger. 2017. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning.* Cambridge University Press. https://doi.org/10.1017/9781316219317

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

[4] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going Beyond Bell's Theorem.* Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

[5] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318

[6] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. 2022. Formal Verification of Quantum Programs: Theory, Tools and Challenges. arXiv:2110.01320 [cs.LO] https://arxiv.org/abs/2110.01320

[7] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. 846–858. https://doi.org/10.1145/3009837.3009894

[8] Qiskit contributors. 2023. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

[9] Christian Rakotonirina. 2005. Tensor Permutation Matrices in Finite Dimensions. arXiv:math/0508053 [math.GM] https://arxiv.org/abs/math/0508053

[10] Peter W. Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A* 52 (Oct. 1995), R2493–R2496. Issue 4. https://doi.org/10.1103/PhysRevA.52.R2493

[11] Dominique Unruh. 2021. Quantum and classical registers. *CoRR* abs/2105.10914 (2021). arXiv:2105.10914 https://arxiv.org/abs/2105.10914

[12] Mingsheng Ying. 2016. *Foundations of Quantum Programming* (1st ed.). Morgan Kaufmann Publishers Inc.

[13] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. CoqQ: Foundational Verification of Quantum Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 29 (Jan. 2023), 33 pages. https://doi.org/10.1145/3571222