A type-theoretic account of quantum computation (extended abstract)

Jacques Garrigue
Nagoya University
Graduate School of Mathematics
Japan

Takafumi Saikawa
Nagoya University
Graduate School of Mathematics
Japan

1 Introduction

Quantum computation is usually presented either in terms of unitary transformations in a Hilbert space [7], or more abstractly as string diagrams representing computations in a symmetric monoidal category [1]. Many works have built on these bases to allow proving quantum algorithms in such settings [4, 6, 8]. 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 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 [2]. 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 it to be parametric, which is equivalent to morphisms being natural transformations.

Subtypes allow one to pack properties of morphism together with their representation.

Using these components, we were able to provide a full account of pure quantum circuits, proving properties from the ground up. We were also able to prove a number of examples, such as the correctness of Shor coding [5] (only for an error-free channel at this point), or of the reverse circuit [8].

Our development is available online [3].

2 Lenses and classical focusing

Our most basic data structure is the lens, which describes which part of the state we want to access.

```
Record lens (n m : nat) := {ℓ : (I_n)^m | uniq ℓ}.
```

Building on the MathComp library, this defines a record whose main component is a tuple of length m, containing natural numbers smaller than n (the ordinal type $\mathtt{I}_n$). It is packed with a proof that this tuple contains no repetition. All together, this provides a canonical representation for injective functions from $\mathtt{I}_m$ to $\mathtt{I}_n$. Thanks to the coercion between `lens` and `tuple`, the underlying function can be accessed as if it were a tuple.

The operations required to define updates are complement, extraction, and merge[1].

```
Parameter lensC n m : lens n m → lens n (n - m).
Parameter extract T n m : lens n m → T^n → T^m.
Parameter merge T n m : lens n m → T^m → T^(n-m) → T^n.
```

In the classical case, we can view data as a tuple, so that functions `extract` and `merge` allow one to use lenses for update:

```
Definition focus1 T n m (l : lens n m) (f : T^m → T^m)
  : T^n → T^n := fun v =>
  merge (f (extract l v)) (extract (lensC l) v).
Lemma focus1_in T n m l f v :
  extract l (@focus1 T n m l f v) = f (extract l v).
```

While these operations cannot be directly applied to quantum states, we will see that they can help us define currying and uncurrying on those states, which is an essential ingredient of focusing in quantum computation.

It is also often useful to compose lenses, or decompose a lens into its inclusion and permutation part. Namely, we have the following functions and laws.

```
Parameter lens_comp n m p :
  lens n m -> lens m p -> lens n p.
Parameter lens_basis n m : lens n m -> lens n m.
Parameter 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 `l1 =i l2` means that `l1` and `l2` are equal as sets.

3 Pure quantum computation

We adopt the traditional view that pure quantum computation amounts to applying unitary transformations to a quantum state. An individual qubit is represented by a vector in $\mathbb{C}^2$. A quantum state composed of n qubits can be described by the n-times iterated tensor product $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2$, also noted $(\mathbb{C}^2)^{\otimes n}$, which is itself isomorphic to $\mathbb{C}^{2^n}$. Unitary

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

transformations on this space are linear transformations that preserve the complex norm.

A quantum circuit is a concrete representation of quantum computation where quantum gates, themselve unitary tranformations, are applied sequentially to specified qubits.

We want to be able to define quantum circuits part by part and compose them into larger ones. A lens ℓ : `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 on $\mathbb{C}^{2^m}$ can be represented by a matrix, so that it can be applied to vectors of T^{2^m}, for an arbitrary complex vector space T, and we can give an arbitrary linear transformation the type

$$G : \forall T : \text{vector sp.}, T^{2^n} \longrightarrow T^{2^n}$$

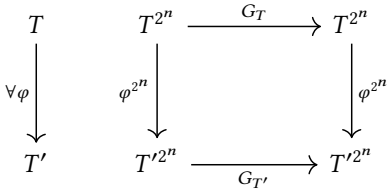
We name the above isomorphism `curry` and its inverse `uncurry`. Along this isomorphism, a gate G can be extended to a larger number of qubits, to become composable in a circuit.

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

The type of G alone does not require that it is defined by a matrix.

$$\exists M,\ \forall T : \text{vector sp.},\ \forall v : T^{2^n},\ G_T(v) = Mv.$$

This property, which we call linear parametricty, is equivalent to naturality with respect to the functor $(-)^{2^n}$:



Our definition of quantum gates is based on naturality.

4 Defining quantum gates

Using MathComp, we can easily encode the concepts described in the previous section. `dpower n T` is the direct-power of a type `T` indexed by n-tuples of some finite type. It can be used to represent focused quantum states.

```
Variables (I : finite type) (dI : I) (R : field).
Definition dpower n T := I^n -fin-> T.
Definition morfun m n :=
  ∀ T : Vect_R, dpower m T → dpower n T.
Definition morlin m n :=
  ∀ T : Vect_R, dpower m T -lin-> dpower n T.
Definition dpmap m T1 T2 (f : T1 → T2)
  (nv : dpower m T1) : dpower m T2 :=
  (v : I^m) -fin-> f(nv(v)).
Definition naturality m n (f : morlin m n) :=
  ∀(T1 T2 : Vect_R), ∀(h : T1 -lin-> T2), ∀ v,
    dpmap h (f T1 v) = f T2 (dpmap h v).
Record mor m n := {φ : morlin m n | naturality φ}.
Notation endo n := (mor n n).
```

A crucial fact we rely on is that, for any commutative ring, MathComp defines the left-module of the finite functions valued into it. This allows us to state that our morphisms should be linear. We can then define naturality in a direct way, and require it for all morphisms. We leave unitarity as an independent property, since it makes sense to have non-unitary morphisms in some situations.

5 Building circuits

The currying defined in section 3 allows to compose circuits without referring to a global set of qubits. This is obtained through two operations: (vertical) 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. We also allow to create a new gate from a given matrix (expressed as a nested direct-power).

```
Parameter comp_mor n m p :
                    mor m p -> mor n m -> mor n p.
Parameter focus n m : lens n m -> endo m -> endo n.
Parameter tsmor n m : dpower n (dpower m R) -> mor m n.
Notation "f \v g" := (comp_mor f g).
```

The definition of `focus` uses currying and polymorphism.

```
Variables (n m : nat) (l : lens n m).
Definition curry (T : Vect_R) (st : dpower n T)
  : dpower m (dpower (n-m) T) :=
  (v : I^m) -fin-> ((w : I^{n-m}) -fin-> st (merge l v w)).
Definition uncurry T (st : dpower m (dpower (n-m) T))
  : dpower n T :=
  (v : I^n) -fin-> st (extract l v) (extract (lensC l) v).
Definition focus_fun (G : endo m) : morfun n n :=
  λT. uncurry_T ∘ G_{T^{I^{n-m}}} ∘ curry_T.
```

In particular, `focus` satisfies the following laws.

```
Definition eq_mor m n (F G : mor m n) :=
  ∀ T : Vect_R, ∀x, F_T(x) = G_T(x).
Notation "f =e g" := (eq_mor f g).
Lemma focus_comp n m (f g : endo m) (l : lens n m) :
  focus l (f \v g) =e focus l g \v focus l g.
Lemma focusM n m p
  (l : lens n m) (l' : lens m p) (f : endo p) :
  focus (lens_comp l l') f =e focus l (focus l' f).
Lemma focusC n m p (l : lens n m) (l' : lens n p)
  (f : endo m) (g : endo n) : [disjoint l & l'] ->
  focus l f \v focus l' g =e focus l' g \v focus l f.
```

While `focus_comp` is almost trivial to prove, `focusM` and `focusC` are more involved, and rely on computations on lenses.

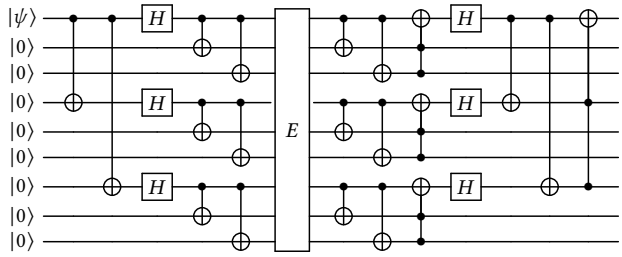


Figure 1. Shor's 9-qubit code

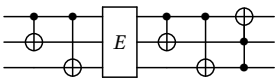


Figure 2. Bit-flip code

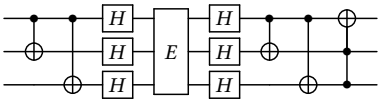


Figure 3. Sign-flip code

6 A concrete example

As an example usage of the above definition of compositions, we 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.

```
Definition bit_flip_enc : endo 3 :=
  focus [lens 0; 2] cnot \v  focus [lens 0; 1] cnot.
Definition bit_flip_dec : endo 3 :=
  focus [lens 1; 2; 0] toffoli \v bit_flip_enc.
Definition hadamard3 : endo 3 :=
  focus [lens 2] hadamard \v focus [lens 1] hadamard
  \v focus [lens 0] hadamard.
Definition sign_flip_dec := bit_flip_dec \v hadamard3.
Definition sign_flip_enc := hadamard3 \v bit_flip_enc.
Definition shor_enc : endo 9 :=
  focus [lens 0; 1; 2] bit_flip_enc \v
  focus [lens 3; 4; 5] bit_flip_enc \v
  focus [lens 6; 7; 8] bit_flip_enc \v
  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.

```
Let shor_input i : 9.-tuple I :=
  [tuple of [:: i; 0%:O; 0%:O; 0%:O; 0%:O;
                   0%:O; 0%:O; 0%:O; 0%:O]].
Lemma shor_code_id i :
  (shor_dec \v shor_enc) Co (dpbasis C (shor_input i))
  = dpbasis C (shor_input i).
```

7 Related works

We compare with some approaches that attempt at proving programs from first principles, using a mathematical model of quantum computation. Most of these 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.

Qwire/SQIR [4] defines a quantum programming language in Coq, modeling internally computation with matrices and Kronecker products. A Hoare logic is also provided that can simplify proofs. This still means doing some heavy matrix caculations.

CoqQ [8] builds a theory of Hilbert spaces and n-ary tensor products on top of MathComp, adding some support for the so-called *labelled Dirac notation*. This allows handling commutation comfortably, but is not fully compositional, in that the notation is based on a fixed set of labels.

Unruh developed a quantum Hoare logic and formalized it in Isabelle, using a concept of *register* [6] which generalizes our lenses, by allowing focusing between arbitrary types rather than just sets of qubits.

To summarize, the only work that appears to be fully compositional in our meaning is Unruh's, but its use of Isabelle means that he had no access to dependent types, and his encoding cannot rely on them.

8 Conclusion

We have used type-theoretical concepts to encode pure quantum computation into Coq, using the MathComp library. 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 tensor products, neither for states, nor for transformations. Since the Kronecker product of matrices can be cumbersome to work with, this is a potential advantage of this approach.

References

[1] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[2] 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.

[3] Jacques Garrigue and Takafumi Saikawa. Qecc proof scripts. https://github.com/t6s/qecc/tree/tyde2023.

[4] 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, page 846–858, 2017.

[5] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995.

[6] Dominique Unruh. Quantum and classical registers. *CoRR*, abs/2105.10914, 2021.
[7] Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2016.
[8] 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.