Formalizing quantum circuits with MathComp/Ssreflect

Takafumi Saikawa   Jacques Garrigue

November 22, 2021

Table of contents

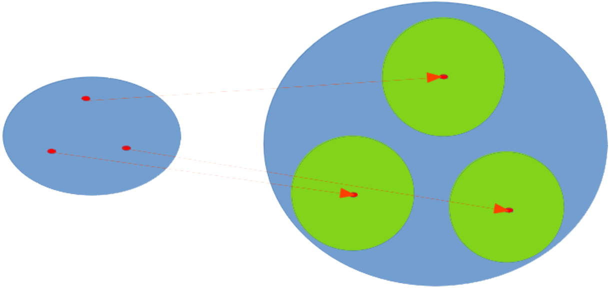
Motivation: Quantum error correction

Classical ECC

- Data are encoded, sent through noisy channel, and decoded back

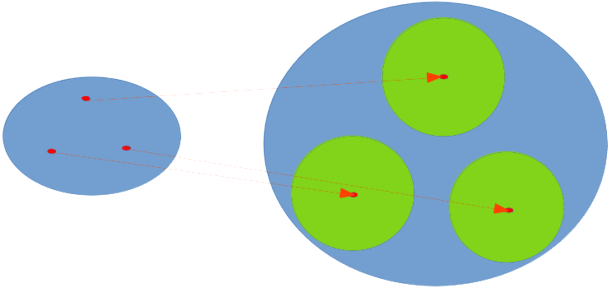
Classical ECC

- Data are encoded, sent through noisy channel, and decoded back
- $\texttt{Encoder} : \texttt{data} \hookrightarrow \texttt{code}$



Classical ECC

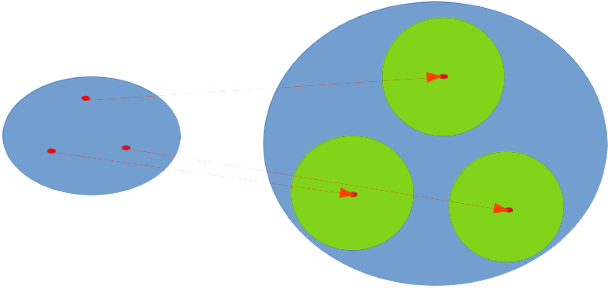
- Data are encoded, sent through noisy channel, and decoded back
- `Encoder : data` $\hookrightarrow$ `code`



- Noises: bit flip / bit erasure

Classical ECC

- Data are encoded, sent through noisy channel, and decoded back
- `Encoder` : `data` $\hookrightarrow$ `code`



- Noises: bit flip / bit erasure
- `Decoder` restores the most likely data from the received value

Plan towards formalizing QECC

Similarly to classical ECC, we plan to do:

2. Formalize encoder, channel and decoder
3. Prove the ability to correct error(s)
4. Information-theoretic analysis

Plan towards formalizing QECC

Similarly to classical ECC, we plan to do:

1. Formalize quantum circuit
2. Formalize encoder, channel and decoder
3. Prove the ability to correct error(s)
4. Information-theoretic analysis

Differences at a quick glance

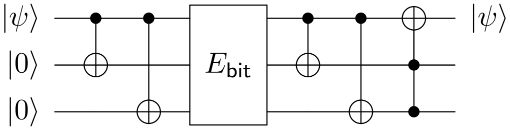
Classical	Quantum
bit $\in \{0,1\}$	qubit $\in \mathbb{C}^2$
functions in $\mathcal{S}et$	unitary morphisms in $\mathcal{H}ilb$
bit flip / bit erasure	bit flip / phase flip / both

$$\left(\text{bit flip} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \text{phase flip} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \text{both} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}\right)$$

Examples

QECCs are written as quantum circuits:

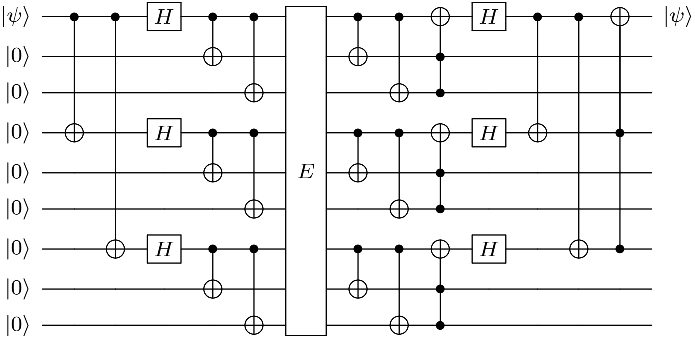
bit-flip correcting code



Examples

Another example:

Shor's 9-qubit code (correcting both flips)



Quantum bit and operator

Differences at a quick glance

Classical	Quantum
bit $\in \{0, 1\}$	qubit $\in \mathbb{C}^2$
functions in $\mathcal{S}\mathrm{et}$	unitary morphisms in $\mathcal{H}\mathrm{ilb}$
bit flip / bit erasure	bit flip / phase flip / both

Differences at a quick glance

Classical	Quantum
bit $\in \{0,1\}$	qubit $\in \mathbb{C}^2$
functions in $\mathcal{S}\mathrm{et}$	unitary morphisms in $\mathcal{H}\mathrm{ilb}$
bit flip / bit erasure	bit flip / phase flip / both

Quantum functions take tensor products, not direct products, as input.

Tensor product

- Direct product

$$X \times Y = \{\langle x, y \rangle \mid x \in X, y \in Y\}$$

Tensor product

- Direct product

$$X \times Y = \{\langle x, y\rangle \mid x \in X, y \in Y\}$$

- Tensor product

$$X \otimes Y = \left\{ \sum_i c_i \langle x_i, y_i \rangle \;\middle|\; c_i \in \mathbb{C}, x_i \in X, y_i \in Y \right\} \Big/ \sim$$

Tensor product

- Direct product

$$X \times Y = \{\langle x, y \rangle \mid x \in X, y \in Y\}$$

- Tensor product

$$X \otimes Y = \left\{ \sum_i c_i \langle x_i, y_i \rangle \,\middle|\, c_i \in \mathbb{C}, x_i \in X, y_i \in Y \right\} \Big/ \sim$$

Tensor product is much bigger!

Tensor power

tensor power

- Tensor power $V^{\otimes n}$
  $=$ iterated tensor product $V \otimes \cdots \otimes V$
- If $V = K^m$, $V^{\otimes n} \cong \mathcal{S}\mathrm{et}(m^n, K)$

```
Variables (I : finType) (R : comRingType).
Definition tpower (n : nat) (T : Type) :=
  {ffun n.-tuple I -> T}.
Definition tpbasis m (vi : m.-tuple I) : tpower m R^o :=
  [ffun vj => (vi == vj)%:R].
```

Qubit

quantum bit

- Qubit $\in \mathbb{C}^2$
- Array of qubits $\in (\mathbb{C}^2)^{\otimes n}$

```
Let R := Reals.Rdefinitions.R.
Let C := [comRingType of R[i]].
Notation "¦ x1 , .. , xn 〉" :=
  (tpbasis _ [tuple of x1 :: .. [:: xn] ..])
```

x1, ..., xn are elements of $\{0, 1\}$

Operator

operator

Operators on qubits are

- linear: addition and scalar action must be preserved

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
```

Operator

operator

Operators on qubits are

- linear: addition and scalar action must be preserved
- unitary: norm must be preserved (not yet)

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
```

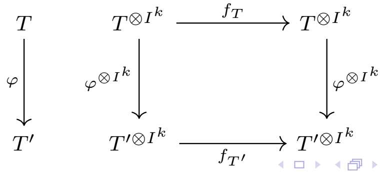
Operator

operator

Operators on qubits are

- linear: addition and scalar action must be preserved
- unitary: norm must be preserved (not yet)
- parametrically linear: explained later

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
```



Example: CNOT

Finite dimensional operators are handily given by matrices:

controlled not

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

Example: CNOT

Finite dimensional operators are handily given by matrices:

controlled not

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

Or, written down using tensor bases:

```
Definition tsquare m := tpower m (tpower m R^o).
Definition ket_bra m (ket bra : tpower m R^o) : tsquare m
  := [ffun vi => ket vi *: bra].
Definition cnot : tsquare C 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⟩.
```

Operators in operation

Problem

- Each gate is fairly simple:

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

Problem

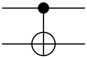
- Each gate is fairly simple:

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

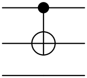
- but when put in a circuit, it becomes a monster:

$$\begin{bmatrix} x & y \\ z & w \end{bmatrix} \otimes \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} xa & xb & ya & yb \\ xc & xd & yc & yd \\ za & zb & wa & wb \\ zc & zd & wc & wd \end{bmatrix}$$

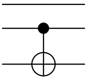
CNOT in 3-qubit circuits



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

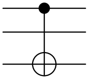


$$= \begin{bmatrix} I_2 & 0 & 0 & 0 \\ 0 & I_2 & 0 & 0 \\ 0 & 0 & 0 & I_2 \\ 0 & 0 & I_2 & 0 \end{bmatrix}$$



$$= \begin{bmatrix} \mathtt{CNOT} & 0 \\ 0 & \mathtt{CNOT} \end{bmatrix}$$

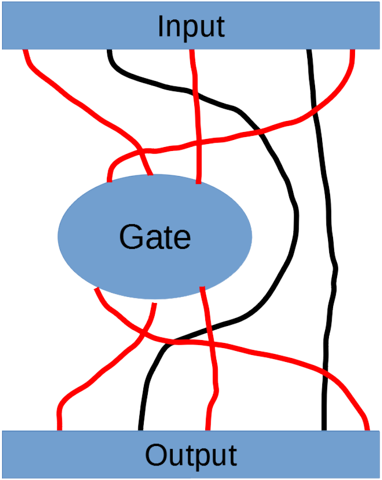
CNOT in 3-qubit circuits



$$= \begin{bmatrix} I_2 & 0 & 0 & 0 \\ 0 & I_2 & 0 & 0 \\ 0 & 0 & X & 0 \\ 0 & 0 & 0 & X \end{bmatrix}$$

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

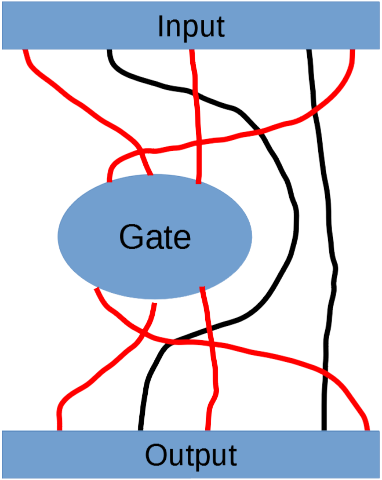
String diagram



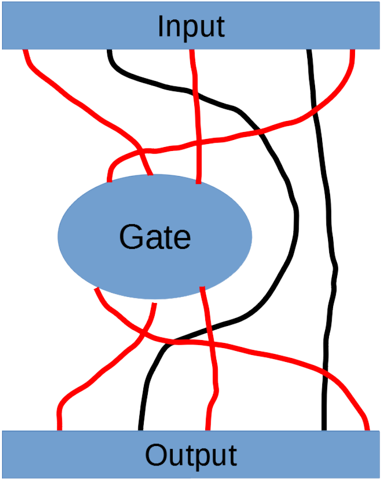
- Natural depiction of gate application
- FP-ish: unused inputs (black) are curried away
- We want to program like this instead of matrices

Lens, curry-uncurry, focus

Lens, curry-uncurry, focus

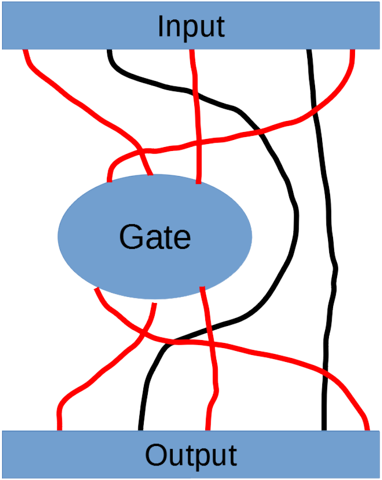


Lens, curry-uncurry, focus



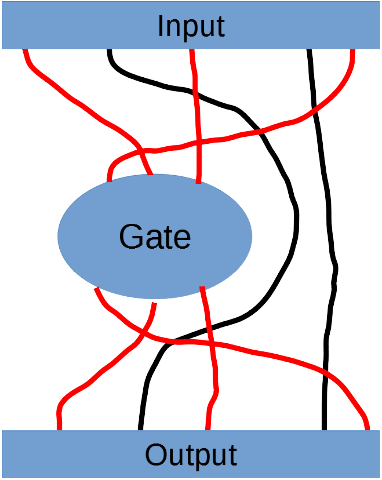
- Lens = choice of wires to be connected to gates; basic combinatorial data

Lens, curry-uncurry, focus



- Lens = choice of wires to be connected to gates; basic combinatorial data
- Currying = quotienting unused wires away

Lens, curry-uncurry, focus



- Lens = choice of wires to be connected to gates; basic combinatorial data
- Currying = quotienting unused wires away
- Focusing = composing curry, gate and uncurry to build the diagram

Lens

lens

lens n m : $\{1, \ldots, m\} \hookrightarrow \{1, \ldots, n\}$

```
Record lens := mkLens
  {lens_t :> m.-tuple 'I_n ; lens_uniq : uniq lens_t}.
```

Currying

curry and uncurry

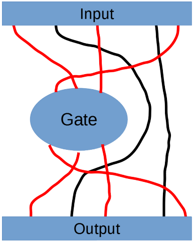
$$\texttt{curry} : T^{2^n} \overset{\cong}{\longleftrightarrow} \left(T^{2^{n-m}}\right)^{2^m} : \texttt{uncurry}$$

$$\left(T^{2^n} = \mathcal{S}\mathrm{et}(2^n, T) \cong \mathcal{S}\mathrm{et}(2^m, \mathcal{S}\mathrm{et}(2^{n-m}, T))\right)$$

```
Variables (T : lmodType R) (n m : nat) (l : lens n m).
Definition curry (st : tpower n T) :
  tpower m (tpower (n-m) T) :=
  [ffun v : m.-tuple I =>
     [ffun w : (n-m).-tuple I =>
        st (merge_indices l v w)]].
Definition uncurry (st : tpower m (tpower (n-m) T)) :
  tpower n T :=
  [ffun v : n.-tuple I =>
     st (extract l v) (extract (lothers l) v)].
```

focus

focus




```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
Definition focus_fun (tr : endo m) : endofun n :=
  fun T (v : tpower n T) => uncurry l (tr _ (curry l v)).
```

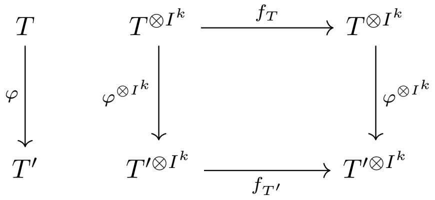
Parametric linearity and naturality

Operator

operator

Operators on qubits must be linear, unitary and, moreover, parametrically linear

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
```



focus

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
Definition focus_fun (tr : endo m) : endofun n :=
  fun T (v : tpower n T) => uncurry l (tr _ (curry l v)).
Lemma focus_is_linear n m l tr T :
   linear (@focus_fun n m l tr T).
Definition focus n m l tr : endo n :=
  fun T => Linear (@focus_is_linear n m l tr T).
```

focus

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
Definition focus_fun (tr : endo m) : endofun n :=
  fun T (v : tpower n T) => uncurry l (tr _ (curry l v)).
Lemma focus_is_linear n m l tr T :
   linear (@focus_fun n m l tr T).
Definition focus n m l tr : endo n :=
  fun T => Linear (@focus_is_linear n m l tr T).
```

- We know from the type that elements of `endo` are linear for each `T`

focus

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
Definition focus_fun (tr : endo m) : endofun n :=
  fun T (v : tpower n T) => uncurry l (tr _ (curry l v)).
Lemma focus_is_linear n m l tr T :
   linear (@focus_fun n m l tr T).
Definition focus n m l tr : endo n :=
  fun T => Linear (@focus_is_linear n m l tr T).
```

- We know from the type that elements of `endo` are linear for each `T`
- But the matrix representing the linearity may be different between `T` s

focus

```
Variables (n m : nat) (l : lens n m).
Definition endofun m := forall T : lmodType R,
  tpower m T -> tpower m T.
Definition endo m := forall T : lmodType R,
  {linear tpower m T -> tpower m T}.
Definition focus_fun (tr : endo m) : endofun n :=
  fun T (v : tpower n T) => uncurry l (tr _ (curry l v)).
Lemma focus_is_linear n m l tr T :
   linear (@focus_fun n m l tr T).
Definition focus n m l tr : endo n :=
  fun T => Linear (@focus_is_linear n m l tr T).
```

- We know from the type that elements of `endo` are linear for each `T`
- But the matrix representing the linearity may be different between `T` s
- And `focus` changes `T`

Parametric linearity

We want our `(f : endo m)` to be represented by a single matrix:

```
Definition tsendo_fun m (M : tsquare m) : endofun m :=
  fun T v =>
    [ffun vi : m.-tuple I =>
     \sum_(vj : m.-tuple I) (M vi vj : R) *: v vj]%R.
Hypothesis endo_parametric (f : endo m) :
  exists M, forall T, f T =1 tsendo M T.
```

Parametric linearity

We want our `(f : endo m)` to be represented by a single matrix:

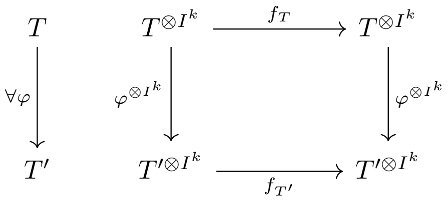
```
Definition tsendo_fun m (M : tsquare m) : endofun m :=
  fun T v =>
    [ffun vi : m.-tuple I =>
     \sum_(vj : m.-tuple I) (M vi vj : R) *: v vj]%R.
Hypothesis endo_parametric (f : endo m) :
  exists M, forall T, f T =1 tsendo M T.
```

Instead of directly axiomatizing this hypothesis, we can rephrase it without the existential reference to a matrix: naturality

naturality

naturality

For `(R : ringType) (T T' : lmodType R) (f : endo m)`,




```
Definition map_tpower m T T' f (nv : tpower m T)
  : tpower m T' := [ffun v : m.-tuple I => f (nv v)].
Definition naturality m (f : endo m) :=
  forall T T' (phi : {linear T -> T'}) (v : tpower m T),
    map_tpower phi (f T v) = f T' (map_tpower phi v).
```

Naturality works!

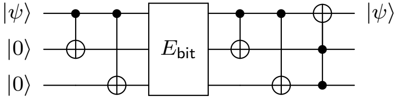
```
Lemma naturalityP m (f : endo m) :
  naturality f
  <-> exists M, forall T, f T =1 tsendo M T.
Lemma focus_naturality n m l tr :
  naturality tr -> naturality (@focus n m l tr).
Lemma focusC (l' : lens n p) tr tr' (v : tpower n T) :
  [disjoint l & l'] ->
  naturality tr -> naturality tr' ->
  focus l tr _ (focus l' tr' _ v) =
  focus l' tr' _ (focus l tr _ v).
Lemma focusM (l' : lens m p) tr (v : tpower n T) :
  naturality tr ->
  focus (lens_comp l l') tr _ v
    = focus l (focus l' tr) _ v.
```

Perspective

Defining gates

Some gates are already defined; more should be done:

```
Definition cnot : tsquare 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⟩.
Definition hadamard : tsquare 1 :=
  (1 / Num.sqrt 2%:R)%:C *:
    (ket_bra ¦0⟩ ¦0⟩ + ket_bra ¦0⟩ ¦1⟩ +
     ket_bra ¦1⟩ ¦0⟩ - ket_bra ¦1⟩ ¦1⟩).
Definition bit_flip (chan : endo 3) : endo 3 :=
  focus [lens 0; 1] (tsendo cnot) \v
  focus [lens 0; 2] (tsendo cnot) \v chan \v
  focus [lens 0; 1] (tsendo cnot) \v
  focus [lens 0; 2] (tsendo cnot) \v
  focus [lens 1; 2; 0] (tsendo toffoli).
```



Back to the original plan towards QECC

1. Formalize quantum circuit
2. QECC: encoder, channel and decoder
3. Prove the ability to correct error(s)
4. Information-theoretic analysis

Category theory

- Monoidal categories with R-linearity
- Naturality and parametricity in other classes of structured types
- Category actions

$$(\texttt{tpower L}) \in \mathcal{C}\mathrm{at}(\mathcal{M}\mathrm{at}_R, \mathcal{LM}\mathrm{od}_R)$$