# A formalization of the Quipper quantum programming language

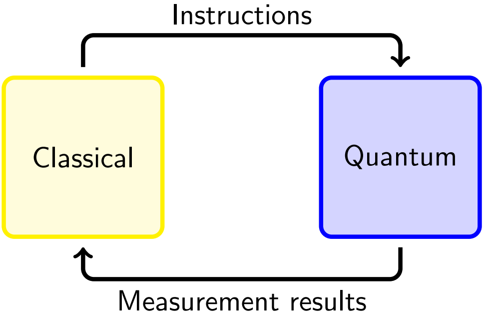
Henri Chataing, Neil J. Ross & Peter Selinger

Dalhousie University & École Polytechnique

2014 TYPES Meeting

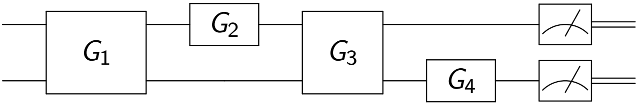
Quantum computing is computing based on the laws of quantum physics.

The standard model of quantum computing is Knill's *Qram model*, in which a classical computer is connected to a quantum device.



The instructions for the quantum device are arranged in a quantum circuit.

The gates that compose quantum circuits can be *unitaries*, which are reversible operations, or *measurements*, which are probabilistic operations.

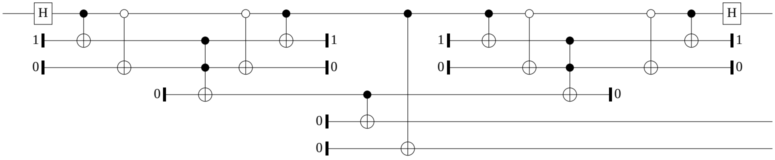


Quipper is a programming language for quantum computing, implemented as an embedded language within Haskell.

Several non-trivial algorithms from the quantum computing literature have been implemented in Quipper.

Quipper is a circuit description language.

Quipper's circuit as data paradigm.




```
circuit :: [Qubit] -> Circ ([Qubit], [Qubit])
circuit qs = do
  y <- with_computed subcircuit $ \subcircuit -> do
    qc_copy subcircuit
  return (qs, y)
```

Quipper's type system does not guarantee that quantum programs are physically meaningful.

```
self_control :: Qubit -> Circ Qubit
self_control q = do
  qnot_at q ‘controlled‘ q
  return q
```

Goals:

- Define a type-safe language, Proto-Quipper, that will serve as a basis for the development of Quipper as a stand-alone language.

Chosen features for Proto-Quipper:

- Have a type system to *enforce the physics* (draw inspiration from the *quantum lambda calculus*).
- Capture Quipper's *circuits as data* paradigm.

Simplifying assumption:

- No measurements (all circuits are therefore reversible).

The Proto-Quipper language:

$$\texttt{Type} \quad A, B ::= \quad 1 \mid \textbf{bool} \mid A \otimes B \mid A \multimap B \mid !A \mid$$

$$\textbf{qubit} \mid \mathrm{Circ}(T, U)$$

$$\texttt{QDataType} \quad T, U ::= \quad \textbf{qubit} \mid 1 \mid T \otimes U$$

$$\texttt{Term} \quad a, b, c ::= \quad \ldots \mid q \mid (t, C, a) \mid \mathit{box}^T \mid \mathit{unbox} \mid \mathit{rev}$$

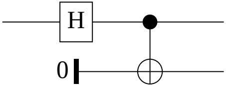
$$\texttt{QDataTerm} \quad t, u ::= \quad q \mid * \mid \langle t, u \rangle$$

Some basic built-in gates:

- $\texttt{HAD} := \mathit{unbox}(q, \mathit{HAD}, q)$
- $\texttt{CNOT} := \mathit{unbox}(\langle q_1, q_2 \rangle, \mathit{CNOT}, \langle q_1, q_2 \rangle)$
- $\texttt{INIT0} := \mathit{unbox}(*, 0, q)$

A Proto-Quipper term (not quite) for `subcircuit`:

$$\texttt{subcircuit} := \mathit{box}^{\textbf{qubit}}(\lambda x.\texttt{CNOT}(\texttt{HAD}\ x, \texttt{INIT0}\ *))$$

$(q,$ 

$, \langle q, q' \rangle)$

Proto-Quipper's operational semantics supposes a circuit constructor.

The circuit constructor is assumed to be able to perform some basic operations: appending gates, reversing circuits, ...

The reduction will be defined on closures $[C, t]$ consisting of a term $t$ of the language and a *circuit state* $C$ representing the circuit currently being built.

The operational semantics of Proto-Quipper (a selection):

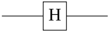
$$\frac{\mathtt{Spec}_{\mathrm{FQ}(v)}(T) = t \quad \mathtt{new}(\mathrm{FQ}(t)) = D}{[C, box^T(v)] \to [C, (t, D, vt)]}$$

$$\frac{[D, a] \to [D', a']}{[C, (t, D, a)] \to [C, (t, D', a')]}$$

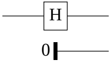
$$\frac{bind(v, u) = \mathfrak{b} \quad \mathtt{Append}(C, D, \mathfrak{b}) = (C', \mathfrak{b}') \quad \mathrm{FQ}(u') \subseteq \mathrm{dom}(\mathfrak{b}')}{[C, (unbox(u, D, u'))v] \to [C', \mathfrak{b}'(u')]}$$

$$\texttt{subcircuit} := box^{\textbf{qubit}}(\lambda x.\texttt{CNOT}(\texttt{INIT0}\ *, \texttt{HAD}\ x))$$

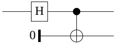
$$[\ \cdot\ , \texttt{subcircuit}] \twoheadrightarrow [\ \text{———}\ , \texttt{CNOT}(\texttt{INIT0}\ *, \texttt{HAD}\ q))]$$

$\twoheadrightarrow$ [  , $\texttt{CNOT}(\texttt{INIT0}\ *, q))$]



$\twoheadrightarrow$ [  , $\texttt{CNOT}(q', q))$]



$\twoheadrightarrow$ [  , $\langle q', q\rangle$]



$$\twoheadrightarrow [\ \cdot\ , (q, C, \langle q', q\rangle)]$$

For each of the constants $box^T$, $unbox$, and $rev$, we introduce a type:

- $A_{box^T}(T, U) = !(T \multimap U) \multimap ! \mathrm{Circ}(T, U)$,
- $A_{unbox}(T, U) = \mathrm{Circ}(T, U) \multimap !(T \multimap U)$, and
- $A_{rev}(T, U) = \mathrm{Circ}(T, U) \multimap ! \mathrm{Circ}(U, T)$.

And a typing rule, for $c \in \{box^T, unbox, rev\}$:

$$\frac{!A_c(T, U) <: B}{!\Delta; \emptyset \vdash c : B}$$

The type system of Proto-Quipper (a selection):

$$\frac{A <: B}{!\Delta, x : A; \emptyset \vdash x : B}\ (ax_c) \qquad \frac{}{!\Delta; \{q\} \vdash q : \mathbf{qubit}}\ (ax_q)$$

$$\frac{\Gamma, x : A; Q \vdash b : B}{\Gamma; Q \vdash \lambda x.b : A \multimap B}\ (\lambda_1) \qquad \frac{!\Delta, x : A; \emptyset \vdash b : B}{!\Delta; \emptyset \vdash \lambda x.b :\ !^{n+1}(A \multimap B)}\ (\lambda_2)$$

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash a :\ !^n A \quad \Gamma_2, !\Delta; Q_2 \vdash b :\ !^n B}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash \langle a, b \rangle :\ !^n(A \otimes B)}\ (\otimes\text{-i})$$

$$\frac{Q_1 \vdash t : T \quad !\Delta; Q_2 \vdash a : U \quad \mathrm{In}(C) = Q_1 \quad \mathrm{Out}(C) = Q_2}{!\Delta; \emptyset \vdash (t, C, a) :\ !^n\,\mathrm{Circ}(T, U)}\ (circ)$$

Proto-Quipper is a type-safe language, It enjoys *subject reduction* and *progress*.

*Subject reduction*: If $\Gamma; \mathrm{FQ}(a) \vdash [C, a] : A, (Q'|Q'')$ is a valid typed closure and $[C, a] \to [C', a']$, then $\Gamma; \mathrm{FQ}(a') \vdash [C', a'] : A, (Q'|Q'')$ is a valid typed closure.

References:


- A.S. Green, P. Lefanu Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. *An introduction to quantum programming in quipper.*
- A.S. Green, P. Lefanu Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. *Quipper: A scalable quantum programming language.*
- P. Selinger and B. Valiron. *Quantum lambda calculus.*