Quantum programming and formal verification (Qbricks)

Christophe Cha

LVP + MTV2, 16-17 nov. 2023
Quantum memory states

- Classical world:

- Quantum world:

\[ \alpha_0 \oplus \alpha_1 \]

with \( \alpha_0, \alpha_1 \in \mathbb{C} \), \( |\alpha_0|^2 + |\alpha_1|^2 = 1 \)
Quantum memory states

- Classical world:
  ![Classical state representation](image)
  One sequence in \( \{0, 1\}^n \) (over \( 2^n \) possible)

- Quantum world:
  \[
  \alpha_0 = 0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad n-1 \\
  \alpha_1 = 0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad n-1 \\
  \vdots \\
  \alpha_{2^n-1} = 0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad n-1
  \]
Quantum memory states

+ Some *strange* rules:

- restricted set of operations: *unitarity*
- destructive measure

- Quantum world:

\[
\begin{align*}
\alpha_0 & : 0 \leftrightarrow 1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow \ldots \leftrightarrow n-1 \\
\alpha_1 & : 0 \leftrightarrow 1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow \ldots \leftrightarrow \times n-1 \\
\vdots & \\
\alpha_{2^{n-1}} & : \times 0 \leftrightarrow \times 1 \leftrightarrow \times 2 \leftrightarrow \times 3 \leftrightarrow \ldots \leftrightarrow \times n-1
\end{align*}
\]
Quantum measurement

- Destructive
- Probabilistic

\[ \alpha_0 \oplus \alpha_1 \]

\[ \text{proba} = |\alpha_0|^2 \]

\[ \text{proba} = |\alpha_1| \]
The hybrid model

A quantum co-processor (QPU), controlled by a classical computer

- classical control flow
- CPU $\Rightarrow$ QPU: quantum computing requests, sent to the QPU
  $\rightarrow$ structured sequenced of instructions: quantum circuits
- QPU $\Rightarrow$ CPU: probabilistic computation results (classical information)
Quantum programming and formal verification (Qbricks)

How to provide intuitive and safe programming experience to a quantum programmer?

- user-friendly programming language
- debugguing/verification apparatus
- trustful compilation process

1. Context and goals
2. Qbricks
User-friendly programming languages

The current quantum programming solutions rely on **sequential descriptions** of elementary quantum operations, similar to classical **assembly programs**.

- The need for **programming** features/primitives ...
  - **High-levelled**, as far as possible
  - With **intuitive** procedural meaning and/but ...
  - ... Formally **interpretable**
  - ... and for characterizing this « formally »

- How to **hold the « big picture »**?
- Unavoidable **side reasoning** in a « formal » setting
  - formal interpretation language on top of the object programming language?

\[
\begin{array}{ccc}
\text{Initial} & \text{Oracle} & \text{Amplification} \\
|0\rangle & H & X \bullet X \quad H \quad X \bullet X \quad H \\
|0\rangle & H & H \bullet H \quad X \bullet X \quad H \\
|0\rangle & H & H \bullet H \quad X \bullet X \quad H \\
|1\rangle & H & H \quad H \quad H \\
\end{array}
\]
A specification preamble:

- **Input parameters**: (size, oracle, etc)
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations
Verification : specifications

A specification preamble:
- **Input parameters** (size, oracle, etc)
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations
A specification preamble:
- **Input parameters** (size, oracle, etc)
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations
Standard debugging techniques fail

<table>
<thead>
<tr>
<th>Potential method</th>
<th>Drawback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion checking ?</td>
<td>Requires (destructive) measurement with highly superposed states</td>
</tr>
<tr>
<td>Final test ?</td>
<td>How to pinpoint error source ?</td>
</tr>
<tr>
<td>Simulation ?</td>
<td>As far as we don’t need a Quantum Computer !</td>
</tr>
</tbody>
</table>
Standard debugging techniques fail...
... the alternative of formal verification

<table>
<thead>
<tr>
<th>Potential method</th>
<th>Drawback</th>
<th>Testing/Assertion checking</th>
<th>Formal verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion checking ?</td>
<td>Requires (destructive) measurement with highly superposed states</td>
<td>executions/simulations</td>
<td>static analysis, no need to execute</td>
</tr>
<tr>
<td>Final test ?</td>
<td>How to pinpoint error source ?</td>
<td>bounded parameters</td>
<td>scale insensitive/any instance</td>
</tr>
<tr>
<td>Simulation ?</td>
<td>As far as we don’t need a Quantum Computer !</td>
<td>statistical arguments</td>
<td>absolute, mathematical guarantee</td>
</tr>
</tbody>
</table>

Build on best practice of formal verification for the classical case and tailor them to the quantum case
Guaranteeing the compilation stack

let $C(f)(x) = \ldots$

$f_0 \to C_1 \to f_1 \to C_2 \to f_2 \to C_3 \to f_3 \to C_4 \to f_4 \to C_t \to f_t$
Final circuit:

- Which physical operations is it based on?
- Does it respect the target topological constraints?
- Optimization?
  - How efficiently?
  - Metrics?
- Functional (quasi) equivalence wrt $C_0$
  - Which notion of distance?
- Does it include error correction?
  - How robustly?

\[
\forall S(C_0, f_t \circ \ldots \circ f_4 \circ f_3 \circ f_2 \circ f_1 \circ f_0(C_0))
\]

\[
S(C_0,C_t)
\]
Quantum programming and formal verification (Qbricks)

How to provide intuitive and safe programming experience to a quantum programmer?

- user-friendly programming language
- debugguing/verification apparatus
- trustful compilation process

1. Context and goals

2. Qbricks
Deductive verification scheme

- Deductive verification: annotate (object language) code with (formal language specifications)
  - preconditions
  - postconditions
  - loop invariants...

- Called functions bring their specifications as guaranteed theorems for proving the specs of calling functions

```python
def abs_succ (a:float):
    ensures{1≤ result}
    abs (a +1)
```

Object language

Specifications

Compilation

Formal semantics

Automated provers
(SMT solvers)
CVC3/Alt-Ergo,
Z3...

Proof assistants
(Coq, Isabelle/HOL,...)
Programs representation $\rightarrow$ path-sums

Standard interpretation as matrices:

- Cumbersome
- Requires higher-order reasoning
Programs representation → path-sums

**Algorithm: Quantum order-finding**

**Inputs:** (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j \mod N\rangle$, for $x$ co-prime to the $L$-bit number $N$, (2) $t = 2L + 1 + \lceil \log \left( \frac{2}{x} + \frac{1}{2} \right) \rceil$ qubits initialized to $|0\rangle$, and (3) $L$ qubits initialized to the state $|1\rangle$.

**Outputs:** The least integer $r > 0$ such that $x^r = 1 \pmod N$.

**Runtime:** $O(L^3)$ operations. Succeeds with probability $O(1)$.

**Procedure:**

1. $|0\rangle|1\rangle$ initial state
2. \(-\frac{1}{\sqrt{2^L}} \sum_{j=0}^{2^L-1} |j\rangle|1\rangle\) create superposition
3. \(-\frac{1}{\sqrt{2^L}} \sum_{j=0}^{2^L-1} |j\rangle|x^j \mod N\rangle\) apply $U_{x,N}$
   \[= \frac{1}{\sqrt{2^L}} \sum_{i=0}^{2^L-1} \sum_{j=0}^{2^L-1} e^{2\pi i j i / r} |j\rangle|u_i\rangle\]
4. \(-\frac{1}{\sqrt{r}} \sum_{x=0}^{r-1} \frac{s}{r} |x\rangle|u_s\rangle\) apply inverse Fourier transform to first register
5. $\frac{s}{r}$ measure first register
6. $\rightarrow r$ apply continued fractions algorithm

*Shor-OF (from N & C, p. 232)*

**Body:**

- An intertwined sequence of intermediate state representations
- A list of **function** applications declarations
Programs representation $\to$ path-sums

**Algorithm: Quantum order-finding**

**Inputs:** (1) A black box $U_{x,N}$ which performs the transformation $|j,k\rangle \to |j, x^j \mod N\rangle$, for $x$ co-prime to the $L$-bit number $N$. (2) $t = 2L + 1 + \log(2 + \frac{1}{x})$ qubits initialized to $|0\rangle$, and (3) $L$ qubits initialized to the state $|1\rangle$.

**Outputs:** The least integer $r > 0$ such that $x^r = 1 \mod N$.

**Runtime:** $O(L^2)$ operations. Succeeds with probability $O(1)$.

**Procedure:**

1. $(0)|1\rangle$

2. $\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j| |1\rangle$

3. $\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j| |x^j \mod N\rangle$

4. $\frac{1}{\sqrt{r}} \sum_{i=0}^{r-1} e^{2\pi is/r} |j\rangle |u_s\rangle$

5. $s/r$

6. $r$

Shor-OF (from N & C, p. 232)

**Body:**

- An intertwined sequence of intermediate state representations
- A list of function applications declarations
Path-sum semantics [Amy 2019]: **formalizing** vector based circuit representations

\[ |x\rangle \rightarrow \frac{1}{\sqrt{2^r}} \sum_{y \in BV_r} e^{i \pi \text{ph}(x,y)} |k(x,y)\rangle \]

- \( r \) : int
- \( \text{ph} \) : Polynomial of symbolic dyadic fractions
- \( k \) : Symbolic binary functions
Programs representation $\rightarrow$ path-sums

Path-sum semantics [Amy 2019]: formalizing vector based circuit representations

$$|x\rangle \rightarrow \frac{1}{\sqrt{2^r}} \sum_{y \in BV_r} e^{i \pi ph(x,y)} |k(x,y)\rangle$$

\[ r : \text{int} \]
\[ ph : \text{Polynomial of symbolic dyadic fractions} \]
\[ k : \text{Symbolic binary functions} \]
MAJOR ACHIEVEMENTS

- a core development framework for parametrized verified quantum programming
- first ever verified implementation of Shor order finding algorithm (95% proof automation)
Toward a formally verified stack: first prototype

Imbricks code for qft(k)
- 7 lines of codes
- 13 lines of specifications
- Functional specs:
\[ |y\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{-2\pi i k x/N} |x\rangle \]
- Performance specs: \( \text{Size} \leq c \cdot k^2 \)
- Well-formedness

12 interactive commands to guide the proof

Mathematical theorems library

IBM simulator
<table>
<thead>
<tr>
<th>k</th>
<th>Lines of code</th>
<th>Simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>4''</td>
</tr>
<tr>
<td>20</td>
<td>200</td>
<td>10''</td>
</tr>
<tr>
<td>80</td>
<td>3200</td>
<td>Non feasible</td>
</tr>
</tbody>
</table>

Standard Oqasm IR for instances of k:

QBRICKS

Oqasm

Simulation
Towards a formally verified stack: first prototype

Imbricks code for qft(k)
- 7 lines of codes
- 13 lines of specifications
- Functional specs:
- Performance specs: Size $\leq c \cdot k^2$
- Well-formedness

Hybrid SR + reasoning

Formal reasoning driven syntax

Proof acceleration

Mathematical theorems library

Imbricks $\rightarrow$ QBRICKS $\rightarrow$ Oqasm $\rightarrow$ Simulation

Standard Oqasm IR for instances of k:

<table>
<thead>
<tr>
<th>k</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>80</td>
<td>3200</td>
</tr>
</tbody>
</table>

IBM simulator

<table>
<thead>
<tr>
<th>k</th>
<th>Simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4''</td>
</tr>
<tr>
<td>20</td>
<td>10''</td>
</tr>
<tr>
<td>80</td>
<td>Non feasible</td>
</tr>
</tbody>
</table>
Toward a formally verified stack: first prototype

Conclusion

- **Qbricks**, a tool for formally verified quantum programming
  - «good» **proof performances**
  - 2 main pillars
    - deductive verification/SMT solvers
    - path-sum semantics
- Ongoing **integration** in a development stack
- Main challenges/obligations
  - refine **symbolic representation**
  - Co-design object language/spec language/semantics