Toward certified quantum programming

Christophe Chareton
Sébastien Bardin, François Bobot, Valentin Perrelle (CEA) and Benoît Valiron (LRI)
Take away

- Quantum computers (are going to / will ...) arrive
  → How to write correct programs?

- Need specification and verification mechanisms
  - scale invariant
  - close to quantum algorithm descriptions
  - well distinguished from code itself
  - largely automated

- We are developing Qbricks as a first step towards this goal
  - core building circuit language
  - dual semantics
  - high level specification framework

- Certified implementation of the phase estimation algorithm (quantum part of Shor)
Outline

Context

The case for verification of quantum algorithms

Qbricks

Circuit language
Dual semantics
Derive proof obligations
Toward further automation

Case study: phase estimation algorithm

Conclusion
Dreams of quantum computing

How long until quantum computing is for everyone?

Applications:
- Machine learning
- Chemistry
- Optimisation
- Cryptography
- Scheduling
- etc
Context

Dreams of quantum computing

How long until quantum computing is for everyone?

If quantum mechanics hasn’t profoundly shocked you, you haven’t understood it yet.

(Niels Bohr)

If you think you understand quantum mechanics, you don’t understand quantum mechanics.

— Richard P. Feynman —
They are coming

IBM will soon launch a 53-qubit quantum computer

Google claims to have reached quantum supremacy

Researchers say their quantum computer has calculated an impossible problem for ordinary machines
They are coming

IBM will soon launch a 53-qubit quantum computer

Noisy Intermediate Scale Quantum (NISQ - J. Preskill 2018)

- Algorithm have to deal with noise
- Limited ressources:
  - 50 - 1000 qubits
  - limited circuit depth

Google claims to have reached quantum supremacy

Researchers say their quantum computer has calculated an impossible problem for ordinary machines
They are coming

IBM will soon launch a 53-qubit quantum computer

Theoretical results:

- Computing gains Vs best-known classical algorithms:
  - Quantum walks: from linear to logarithmic
  - Grover-based searches: from $e^n$ to $e^{n/2}$
  - Shor-like algorithms: from exponential to linear
  - Quantum simulation: from exponential to linear

- "Absolute" theoretical quantum supremacy?

  $\text{BQP} \subseteq \text{NP}$
  $\text{BPP} \subseteq \text{BQP}$

  $\text{BQP} \supseteq \text{NP}$
  $\text{BPP} \overset{?}{=} \text{BQP}$

Researchers say their quantum computer has calculated an impossible problem for ordinary machines.
Quantum computing milestone history

1982 Feynman intuition
1994 Shor
2009, 2013 HHL, VQE
Quantum computing milestone history

- Feynman intuition (1982)
- Shor (1994)
- HHL, VQE (2009, 2013)
- Shor implementation (2001)
- D-Wave One (2012)
- IBM Q (2015, 2019)
Quantum computing milestone history


2001: Shor implement
2012: D-Wave One
2019: IBM Q

- High-level programming language
  - Some exist: Q#, liqui⟩, Quipper, etc
- Compilers
- Assembly-like language
- Optimizers

\[ 15 = 3 \times 5 \]
Quantum computing milestone history

- Feynman intuition (1982)
- Shor (1994)
- HHL VQE (2009-2013)

- Shor implement 15=3*5 (2001)
- D-Wave One (2012)
- IBM Q (2015-2019)

- Specification
- High-level programming language
- Verification tools
- Compilers
- Assembly-like language
- Optimizers
Quantum information

Classical world:

\[
\alpha_0 \oplus \alpha_1
\]

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

Quantum world

\[1\] In a \( 2^n \) dimension vector space, \( |k\rangle_n \) designates the \( k^{th} \) canonical basis vector.
Context

Quantum information

- Classical world:
  0 1 2 3 ... n−1
  One sequence in \{0, 1\}^n (over 2^n possible)

- Quantum world\(^1\)

\[ |α⟩_n = \bigoplus_{k=0}^{2^n-1} α_k |k⟩_n \]

\(^1\)In a 2^n dimension vector space, |k⟩_n designates the k^{th} canonical basis vector.
Context

Quantum information

- Classical world:

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & \ldots & n-1
\end{array}
\]

One sequence in \(\{0,1\}^n\) (over \(2^n\) possible)

- Quantum world\(^1\)

\[
|\alpha\rangle_n = \bigoplus_{k=0}^{2^n-1} \alpha_k |k\rangle_n
\]

+ Some strange rules:

- no cloning
- destructive measure
- operations restricted to unitary

\(^1\)In a \(2^n\) dimension vector space, \(|k\rangle_n\) designates the \(k^{th}\) canonical basis vector
The QRAM model

- A quantum co-processor (QRAM), controlled by a classical computer
  - Classical control flow
  - Quantum computing request, sent to the QRAM
- Structured sequences of instructions: quantum circuits
The QRAM model

- A quantum co-processor (QRAM), controlled by a classical computer
  - Classical control flow
  - Quantum computing request, sent to the QRAM
- Structured sequences of instructions: quantum circuits

Does the circuit fit the computation need?
The case for verification of quantum algorithms

Outline

Context

The case for verification of quantum algorithms

Qbricks
  Circuit language
  Dual semantics
  Derive proof obligations
  Toward further automation

Case study: phase estimation algorithm

Conclusion
The case for verification of quantum algorithms

How do we check them?

Quantum phase estimation (from Nielsen & Chuang)

\[
\begin{align*}
\ket{0}
\end{align*}
\]

initial state
create superposition
apply black box
result of black box
apply inverse Fourier transform
measure first register

\[
\begin{align*}
\frac{1}{\sqrt{2}} \sum_{j=0}^{2^n-1} |j\rangle |u\rangle \\
\rightarrow \\
\frac{1}{\sqrt{2}} \sum_{j=0}^{2^n-1} |jU^{1/2} |u\rangle \\
= \\
\frac{1}{\sqrt{2}} \sum_{j=0}^{2^n-1} e^{(2\pi i j^n) u} |j\rangle |u\rangle \\
\rightarrow |\tilde{\psi}_u\rangle |u\rangle \\
\rightarrow |\tilde{\psi}_u\rangle
\end{align*}
\]

Quantum phase estimation (from Nielsen & Chuang)

Quipper QFT circuit building function

```
quft_internal :: [Qubit] -> Circ [Qubit]
quft_internal [] = return []
quft_internal [x] = do
  hadamard x
  return [x]
quft_internal (x:xs) = do
  xs' <- qft_internal xs
  x' <- hadamard x
  return (x':xs')
where
  -- Auxiliary function used by 'qft'.
  rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
  rotations _ [] = return []
  rotations c (q:qs) n = do
    qs' <- rotations c qs n
    q' <- rGate ((n + 1) * length qs) q 'controlled' c
    return (q':qs')
```
The case for verification of quantum algorithms

How do we check them?

- Quantum programming is tricky and non-intuitive
- No means to control an execution
- Tests are expensive and often statistical

---

Quantum phase estimation (from Nielsen & Chuang)

\[
|0\rangle\langle u| \\
\rightarrow \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} |j\rangle\langle u| \\
\rightarrow \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} |j\rangle U^j |u\rangle \\
= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} c^{2 \pi i j u} |j\rangle |u\rangle \\
\rightarrow |\psi_u\rangle |u\rangle \\
\rightarrow |u\rangle
\]

Implementing initial state
create superposition
apply black box
result of black box
apply inverse Fourier transform
measure first register

Quipper QFT circuit building function

\[
\text{qft\_internal} :: [\text{Qubit}] -> [\text{Qubit}] \\
\text{qft\_internal} [] = \text{return} [] \\
\text{qft\_internal} [x] = \text{do} \\
\text{hadamard} x \\
\text{return} [x] \\
\text{qft\_internal} (x:xs) = \text{do} \\
x's' <- \text{qft\_internal} xs \\
x's'' <- \text{rotations} x x's' \\
x' <- \text{hadamard} x \\
\text{return} (x':xs'')
\]

Where

- Auxiliary function
- \text{rotations} :: [\text{Qubit}] -> [\text{Qubit}]
- \text{rotations} _ [] _ = \text{return} []
- \text{rotations} c (q:qs) n = 
  \text{q}' <- \text{rotations} c qs \\
  \text{q} <- \text{r\_gate} ((n + 1)) \\
  \text{return} (q':qs')

If you think you understand quantum mechanics, you don’t understand quantum mechanics.

— Richard P. Feynman —
The case for verification of quantum algorithms

How do we check them?

- Testing is difficult . . .
- What about full verification? allows to handle
  - Infinite state space
  - absolute guarantee
The case for verification of quantum algorithms

[A parte] Annotated code and deductive verification

- Provides absolute guarantee
- Automates proofs
- Industrial successes
- Verify wide-spread languages (C, Java, caml ...)

Three main ingredients:
- operational semantics
- specification language
- proof engine
The case for verification of quantum algorithms

State of affairs in quantum computing

Three main ingredients:

- **operational semantics:** matrices $\rightarrow$ matrix product, from Heisenberg (1925), Dirac (1939), path-sums (2018)

- **specification language:** ???

- **proof engine:** ???
The case for verification of quantum algorithms

Our strategy

- Build on best practice of formal verification for the classical case
  - separation of concerns
  - scale invariant verification
  - proof automation
  - domain-based specialization
  - flexible specification language

- Tailor them to the quantum case
  - dual semantics (truth reference + specifications)
  - specific reasoning rules
  - dedicated lemmas (1000+) libraries
The case for verification of quantum algorithms

State of the art

<table>
<thead>
<tr>
<th></th>
<th>QMC</th>
<th>Coq</th>
<th>Qwire (Coq)</th>
<th>Path-sums</th>
<th>Qbricks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate spec.</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Scale inv.</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Spec. fitting</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Automate proofs</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

Table: Formal verification of quantum circuits
The case for verification of quantum algorithms

State of the art, achievements in quantum formal verification

Size (number of qbits)

\[ \infty \]

Qwire

\[ 1000 \]

\[ 100 \]

\[ 10 \]

Superposition

coin flip

teleportation

Coq

Qwire

QMC

Path-sums

QFT

Phase estimation

Shor algorithm

\[ \otimes \]

OUR CONTRIBUTION
Outline

Context

The case for verification of quantum algorithms

Qbricks

- Circuit language
- Dual semantics
- Derive proof obligations
- Toward further automation

Case study: phase estimation algorithm

Conclusion
The quantum case : Back to basics

1. $|0\rangle|u\rangle$ initial state
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|u\rangle$ create superposition
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle U^{\dagger}|u\rangle$ apply black box
   $= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u} |j\rangle|u\rangle$ result of black box
4. $\rightarrow |\tilde{\varphi}_u\rangle|u\rangle$ apply inverse Fourier transform
5. $\rightarrow \tilde{\varphi}_u$ measure first register

Algorithm for the quantum phase estimation
The quantum case: Back to basics

Algorithm for the quantum phase estimation

1. $|0\rangle|u\rangle$
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|u\rangle$
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle U^j |u\rangle$
   $= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u} |j\rangle|u\rangle$
4. $\rightarrow |\tilde{\varphi_u}\rangle|u\rangle$
5. $\rightarrow \tilde{\varphi_u}$

- Initial state
- Create superposition
- Apply black box
- Result of black box
- Apply inverse Fourier transform
- Measure first register

A sequence of operations
Intermediate assertions, describing the state of the system at each step
## The quantum case: Back to basics

### Algorithm for the quantum phase estimation

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>initial state create superposition</td>
</tr>
<tr>
<td>2.</td>
<td>apply black box</td>
</tr>
<tr>
<td>3.</td>
<td>result of black box</td>
</tr>
<tr>
<td>4.</td>
<td>apply inverse Fourier transform</td>
</tr>
<tr>
<td>5.</td>
<td>measure first register</td>
</tr>
</tbody>
</table>

Derive function specifications, e.g.:

```plaintext
let create_superposition (state)
    pre: |u⟩ is a ket vector
    pre: state = |0⟩|u⟩
    post: state = \( \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j⟩|u⟩ \)
    = (* The program *)
```

- Deductive verification (of annotated programs)
- → a domain specific language, embedded in the Why3 environment
  - build on best practice for the classical case (automation, separation of concerns, . . .)
  - + interface with SMT-solvers
Qbricks – Dual semantics

Circuit building functions

\[ U^{2^{l-1}} \quad U^{2^{l-2}} \quad \cdots \quad U^{2^0} \]

\[ \text{Rev}(\text{QFT} (n)) \]

\textbf{type} quantum_circuit_pre =
  Phase real | Rx real | Ry real | Rz_real | Cnot
  | Sequence quantum_circuit_pre quantum_circuit_pre
  | Parallel quantum_circuit_pre quantum_circuit_pre

Séminaire Gallium — Christophe Chareton — p. 18
Specification and verification

- Decorate Qbricks code with specifications
- Interprete circuit as functions transforming quantum states

\[
\begin{align*}
\text{Decorate Qbricks code with specifications} & \\
\text{Interprete circuit as functions transforming quantum states} & \\
\end{align*}
\]

\[
x: \text{quantum\_state} \quad \text{semantics} \quad C: \text{quantum\_circuit} \quad \mapsto \quad \lbrack C, x \rbrack: \text{quantum\_state}
\]

- Path-sum semantics, general form

\[
\begin{align*}
\text{Path-sum semantics, general form} & \\
C, \ket{k}_n \quad \mapsto \quad \frac{1}{\sqrt{2^r}} \sum_{j=0}^{2^r-1} \phi(h(k), j) \ket{\text{ket}(i, j)}_n
\end{align*}
\]

- Three separated parameters, whith recursive definitions:

  - \( r: \text{int} \)
  - \( \phi: \text{int} \to \text{int} \to \text{complex} \)
  - \( \text{ket}: \text{int} \to \text{int} \to \text{int} \)
Specified circuit building

Three separated parameters:
- \( r: \text{int} \)
- \( \text{ph}: \text{int} \rightarrow \text{int} \rightarrow \text{complex} \)
- \( \text{ket}: \text{int} \rightarrow \text{int} \rightarrow \text{int} \)

functions \( r \) (sum\_range), \( \text{ph} \) (phase\_part) and \( \text{ket} \) (ket\_part) are defined by recursion for circuits,
Specified circuit building

- Three separated parameters:
  - \( r: \text{int} \)
  - \( \text{ph}: \text{int} \rightarrow \text{int} \rightarrow \text{complex} \)
  - \( \text{ket}: \text{int} \rightarrow \text{int} \rightarrow \text{int} \)

- Functions \( r \) (sum_range), \( \text{ph} \) (phase_part) and \( \text{ket} \) (ket_part) are defined by recursion for circuits,

- They specify circuit lifted constructors

```ocaml
let function sequence (d e: quantum_circuit): quantum_circuit
  requires{size d = size e}
  ensures{sum_range result = sum_range d + sum_range e}
  ensures{size result = size d}
  ensures{forall bv x bv y: int->int. forall i :int.
    ket_part result bv x bv y i = if 0<= i < size result then
    ket_part e (ket_part d bv x bv y) (shift bv y (sum_range d)) i else 0}
  ensures{forall bv x bv y: int->int. phase_part result bv x bv y =
    (phase_part e (ket_part d bv x bv y) (shift bv y (sum_range d)))}
  = {to_pre = Sequence (to_pre d) (to_pre e)}
```
Specified circuit building

- Three separated parameters:
  - $r : \text{int}$
  - $\text{ph} : \text{int} \rightarrow \text{int} \rightarrow \text{complex}$
  - $\text{ket} : \text{int} \rightarrow \text{int} \rightarrow \text{int}$

- Functions $r$ (sum_range), $\text{ph}$ (phase_part) and $\text{ket}$ (ket_part) are defined by recursion for circuits,

- They specify circuit lifted constructors

- And the circuit building functions

```haskell
let function hadamard(): quantum_circuit
    ensures\{size_result = 1\}
    ensures\{sum_range result = 1\}
    ensures\{forall bx, by: \text{int} -> \text{int}, forall i: \text{int}.
        0 <= i < size result -> ket_part result bx by i = bx by i\}
    ensures\{forall bx, by: \text{int} -> \text{int}, binary bx -> binary by ->
        phase_part result bx by = value (dyadic (bx 0 * by 0) 1)\} =
    sequence (rz p 1) (ry (dyadic 1 3))
```
Generating proof obligations (why3)

Compilation generates proof obligations

```ocaml
let function hadamard(): quantum_circuit
ensures{size result = 1}
ensures{sum_range result = 1}
ensures{forall bx, by: int->int. forall i:int.
    0 <= i < size result -> ket_part result bx by i = by i}
ensures{forall bx, by: int->int. binary bx -> binary by ->
    phase_part result bx by = value (dyadic (bx 0 * by 0) 1)}
= sequence (rzp 1) (ry (dyadic 1 3))
```

Séminaire Gallium — Christophe Chareton — p. 21
Generating proof obligations (why3)

- Compilation generates proof obligations
- Calling a function provides its postconditions as axioms

```
let function hadamard(): quantum_circuit
  ensures[size_result = 1]
  ensures[sum_range_result = 1]
  ensures[for all bvx bvy: int -> int. for all 0 <= i < size_result -> ket_part_result]
  ensures[for all bvx bvy: int-> int. binary phase_part_result bvx bvy = value]
  sequence (zp 1) (ry (dyadic 1 3))
```

```
axiom H7 :
  forall bvx1:int -> int, bvy1:int -> int.
  forall il:int.
  ket_part_result bvx1 bvy1 il
  = (if 0 <= il \& il < depth_result
    then ket_part_result (ry (dyadic 1 3))
        (((fun (y0:quantum_circuit) (y1:int -> int) (y2:int -> int) (y3:int -> int)
          -> ket_part_result y0 y1 y2 y3)
    @ rzp_1)
    @ bvx1)
    @ bvy1)
    (((fun (y0:int -> int) (y1:int) (y2:int) -> shift y0 y1 y2)
      @ bvy1)
    @ sum_range_ (rzp_1))
  il
  else 0)
```
Qbricks – Derive proof obligations

Supporting proof obligations

- Proof obligations may be sent to SMT-solvers,
- and they can be eased, if needed, by interactive transformations
Outline

Context

The case for verification of quantum algorithms

Qbricks

Circuit language
Dual semantics
Derive proof obligations
Toward further automation

Case study: phase estimation algorithm

Conclusion
Path-sum semantics nice properties

- Path-sum semantics satisfies nice properties

Lemma (Properties of function $Ps$)

- **Linear decomposition.** Let $c$ be a quantum circuit, then for any ket $|u\rangle$ of length $s_c$,

\[
Ps(c, |u\rangle) = \sum_{k=0}^{2^{s_c-1}} (u(k))Ps(c, |k\rangle_{s_c})
\]

- **Compositions.** Let $c$ and $c'$ be quantum circuits, let $|u\rangle$ and $|u'\rangle$ be ket of respective lengths $s_c$ and $s_{c'}$. Then,

  - $Ps(\parallel(c, c'), |u\rangle \otimes |u'\rangle) = (Ps(c, |u\rangle) \otimes (Ps(c', |u'\rangle))$
  - if $c$ and $c'$ have the same size then $Ps(\text{sequence}(c, c'), |u\rangle) = Ps(c', Ps(c, |u\rangle))$

- They enable local reasoning without reference to $r$, $ph$ and $ket$
Abstract specification: the *eigen* example

**Definition (Eigen predicate)**

Let \( c \) be a quantum circuit, then for any ket \(|u\rangle\) of length \( s_c \) and for any complex number \( v \), we say that \(|u\rangle\) is an *eigenvector* for \( c \) with associated eigenvalue \( v \), and we write \( \text{eigen}(c, |u\rangle, v) \), iff

\[
\text{Ps } (c, |u\rangle) = v|u\rangle
\]

**Lemma (Eigen sequence composition)**

Let \( c \) and \( c' \) be quantum circuits such that \( s_c = s_{c'} \), let \(|u\rangle\) be a ket of length \( s_c \). Then for any complex values \( v, v' \) such that

- \( \text{eigen}(c, |u\rangle, v) \)
- \( \text{eigen}(c', |u\rangle, v') \)

then

\( \text{eigen(sequence}(c, c'), |u\rangle, vv') \)
Abstract specification and quantum algorithms

Quantum algorithms (phase estimation, Grover, quantum simulation, etc.) are often parametrized by an oracle quantum circuit, respecting a given property.

This specification may not (nicely) translate in terms of $r$, $p$, $h$ and $ket$.

→ abstract specifications

Example, the phase estimation algorithm:

**Input:** an unitary operator $U$ and an eigenstate $|v\rangle$ of $U$

**Output:** the eigenvalue associated to $|v\rangle$

Also: reversed circuit specification, controlled operations specifications, bit permutation specifications, etc.
Specific language fragments and simplified path-semantics

Predicates may characterize *Qbricks* fragments with simplified path-sum semantics:

<table>
<thead>
<tr>
<th>Spec.</th>
<th>generating syntax</th>
<th>Semantics</th>
<th>Design input</th>
</tr>
</thead>
</table>
| $T$   | $\{Rx, Ry, Rz, Ph, Cnot\}$ | $r: \text{int}$  
$\text{ph}: \text{int} \to \text{int} \to \text{complex}$  
$\text{ket}: \text{int} \to \text{int} \to \text{int}$ |                                            |
| flat  | $\{Rz, Ph, Cnot\}$ | $\text{ph}: \text{int} \to \text{complex}$  
$\text{ket}: \text{int} \to \text{int}$ | easy specification |
| diag  | $\{Rz, Ph\}$       | $\text{ph}: \text{int} \to \text{complex}$ | easier specification  
iterators                           |
Case study: phase estimation algorithm

Outline

Context

The case for verification of quantum algorithms

Qbricks
  Circuit language
  Dual semantics
  Derive proof obligations
  Toward further automation

Case study: phase estimation algorithm

Conclusion
Case study: phase estimation algorithm

**Phase estimation**

**Input:** an unitary operator $U$ and an eigenstate $|v\rangle$ of $U$

**Output:** the eigenvalue associated to $|v\rangle$

- Eigen decomposition
- Solving linear systems
- Shor (with arithmetic assumption and probability)

**Size (number of qbits)**

$\infty$  
Qwire

**Difficulty**

Superposition  
coin flip  
teleportation  
Qwire  
QMC  
Path-sums  
QFT  
Phase estimation  
Shor algorithm

---

OUR CONTRIBUTION
Case study: phase estimation algorithm

**Implementation data**

<table>
<thead>
<tr>
<th></th>
<th>#Lines</th>
<th>#Def.</th>
<th>#Lem</th>
<th>#POs</th>
<th>#Aut.</th>
<th>#Cmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_superposition</td>
<td>42</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>apply_black_box</td>
<td>57</td>
<td>3</td>
<td>1</td>
<td>50</td>
<td>44</td>
<td>46</td>
</tr>
<tr>
<td>QFT</td>
<td>75</td>
<td>3</td>
<td>0</td>
<td>57</td>
<td>51</td>
<td>30</td>
</tr>
<tr>
<td>phase_estimation</td>
<td>63</td>
<td>4</td>
<td>0</td>
<td>72</td>
<td>65</td>
<td>51</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>237</strong></td>
<td><strong>12</strong></td>
<td><strong>2</strong></td>
<td><strong>190</strong></td>
<td><strong>166</strong></td>
<td><strong>163</strong></td>
</tr>
</tbody>
</table>

#Aut.: automatically proven POs — #Cmd: interactive commands

Table: Implementation & verification of phase estimation

- The objective is reached: prove by fact that parametrized formal verification for quantum programs is possible
- Future works: further automate proof fulfillment
Case study: phase estimation algorithm

Comparison of several approaches, QFT algorithm

- Separate specification from code
- Scale invariance
- Specifications fitting algorithm
- Automate proofs

<table>
<thead>
<tr>
<th></th>
<th>QMC</th>
<th>Coq</th>
<th>Qwire (Coq)</th>
<th>Path-sums</th>
<th>Qbricks</th>
</tr>
</thead>
<tbody>
<tr>
<td>QFT (full Qbricks)</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>QFT (Path-sum only)</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>QFT (Matrix only, cf QMC, Coq solutions)</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table: Formal verification of quantum circuits

<table>
<thead>
<tr>
<th></th>
<th>#Lines</th>
<th>#Def.</th>
<th>#Lem</th>
<th>#POs</th>
<th>#Aut.</th>
<th>#Cmd</th>
</tr>
</thead>
<tbody>
<tr>
<td>QFT (full Qbricks)</td>
<td>75</td>
<td>3</td>
<td>0</td>
<td>57</td>
<td>51</td>
<td>30</td>
</tr>
<tr>
<td>QFT (Path-sum only)</td>
<td>87</td>
<td>3</td>
<td>0</td>
<td>73</td>
<td>64</td>
<td>49</td>
</tr>
<tr>
<td>QFT (Matrix only, cf QMC, Coq solutions)</td>
<td>200</td>
<td>8</td>
<td>15</td>
<td>306</td>
<td>285</td>
<td>106</td>
</tr>
</tbody>
</table>

Table: Comparison of several approaches, QFT algorithm
Conclusion

Qbricks: a core development framework for certified quantum programming
- scale invariant
- close to quantum algorithm descriptions
- well distinguished from code itself
- largely automated

Implementation
- Circuit building language
- Dual semantics + equivalence proof
- Shorcuts for further automation
- Certified implementation of the phase estimation algorithm

Future works:
- Further automate proof framework
- Extend Qbricks to measure → Shor