Qbricks, a framework for formal verification in quantum computing [*]

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perelle
CEA, LIST, Université Paris-Saclay, France
firstname.lastname@cea.fr

Benoît Valiron
LRI, CentraleSupélec, Université Paris-Saclay, Orsay, France
benoit.valiron@lri.fr

Abstract

In this abstract we introduce a core programming framework for formally verified quantum programming, called *Qbricks*.Thanks to it, we could implement a completely verified scale invariant non trivial quantum calculus (phase estimation).

1 Introduction

Context. After important progresses [19] in algorithmic since 1982 [6] and the advent of the first quantum computers, led by great investors (Google, IBM, D-Wawe, etc), the different stages of quantum software development are now identified and start being explored: programming languages (Quipper [10, 11], Q# [26, 27], liqui$|\rangle$ [28],Q$\#$, Qwire [20] etc), compilers [15, 24], assembly languages [5, 25] or optimization [13, 17, 18].

The Qram model.

The standard architecture model (Figure 1) for quantum computing is an hybrid model, called Qram: programs are written thanks to a classical computer, which ensures the execution control flow. This computer also controls a *quantum coprocessor*, by sending it computing instructions formatted as structured sequences of unitary operations (we call such sequences *quantum circuits*).

Figure 1 also illustrates the human-computer interaction in the simplest scenario, when an user requires a value $f(x)$ that requires a single run of the quantum co processor.

[*]This work was partially supported by the French National Research Agency (ANR) under the research project SoftQPRO ANR-17-CE25-0009-02, by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746-484124.

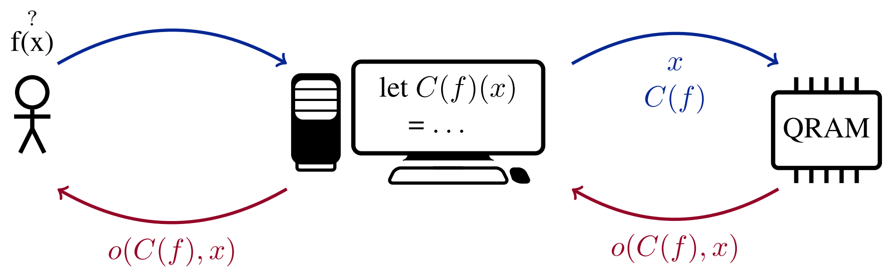

Figure 1: The Qram hybrid model

Problem. A crucial question rises from this model:

How a developer can get insured of the correction of the programs she writes?

This problem is crucial for programming, both in the classical and in the quantum case. Nevertheless, due to the nature of quantum calculus, testing and debugging strategies (which are the most widely used in classical computing) are not transposable in the quantum case. Indeed, the destructive measurement of quantum registers makes impossible to insert run time control assertions (they would require measurements along the execution, which would destruct the state of the memory and falsify the rest of the execution). On an other hand, strategies based on complete executions testing would also fail on this probabilistic frame (testing would concern probability distributions for the output of a circuit, which would require sufficiently many test runs to build such a distribution).

We think the solution to these dead ends should come from formal verification of programs. In addition to offering an alternative to dead ended testing strategy, it has the decisive additional advantages both to enable functional scale-invariant proof certificates and to offer once for all absolute guarantee for the correction of programs.

Goal and challenge. We therefore aim at developing a complete solution for formally certified quantum programming. We want it to meet the following requirements :

- it should provide means for writing programs straightly derived from the description of algorithms in the literature,
- it should enable to clearly distinguish a program from its specification,
- it should provide scale invariant certifications for parametrized families of circuits,
- it should enable, as far as possible, an automatic treatment of proof obligations support.

State of the art. Some efforts have been recently made towards formal verification of quantum programs, yet none of them satisfy all the identified requirements: the model-checking approach [8, 29] is fully automatic but scale sensitive. Approaches based on Coq proof assistant, such as [3] or Qwire [14, 20, 22], use formal proofs and are therefore better fitted for

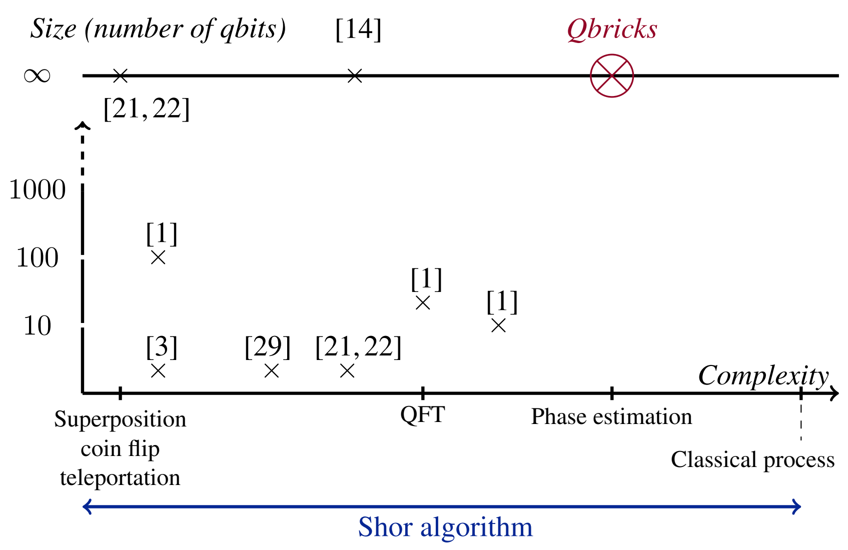

Figure 2: Certified quantum circuits in the literature

proving big calculi. So far, the approach mainly came with illustration by small examples (eg.: coin_flip [21], teleportation [3], etc). Recently, a fragment of Qwire, called sQIRE [14], was used to exhibit a verified parametrized implementation of the Deutsch-Jozsa algorithm. Furthermore, this approach requires from the user to fully write the proofs, using the complex type annotations system of Coq.

Matthew Amy developed a powerful semantic framework, the path-sums semantics [1, 2] that he used for certified compilation and thanks to which he could prove the correction of scale sensitive quantum calculi (including up to 100 qubits). These different achievements are reported in Figure 2. The horizontal axis represents the structural complexity of circuits, from a simple state superposition to the complete implementation of an actual algorithm. Many of the mentioned elements being steps in the implementation of Shor algorithm, we can use it as a milestone. On the vertical axis is the size, *ie* the number of qubits, of the verified quantum circuits. The scale is logarithmic and we include the infinite limit, corresponding to parametrized families of circuits. Our aim was to enable scale-invariant specifications and verifications for complex circuits. As a proof of concept for our framework we implemented a fully verified version of the phase estimation algorithm, which we also figured on Figure 2.

2 *Qbricks* working program

In order to reach the goals mentioned above, we are developing a core quantum certified programming language, called *Qbricks*. It is embedded in the Why3 verification environment [7] and enables writing and manipulating (families of) quantum circuits. In addition, *Qbricks* comes with semantical tools. They enable to interpret quantum circuits as mathematical objects, to write formal specifications for these circuits and to formally verify these specifications.

Annotated programming and dual *Qbricks* semantics. The aim of *Qbricks* is to enable completely specified and proved programming in the Qram model. In the present state of its development, specifications and proofs concern quantum circuits, which corresponds to fragments of code for the quantum co-processor (or to the basis case scenario, where the expected outcome requires a single run of the quantum co-processor).

In future works, we plan to implement a semantic interpretation of the measurement operation and the classical control flow at stake in the Qram model.

The overall framework is represented in Figure 3. It extends the Qram model with *Qbricks*. As in the model from Figure 1, an agent aiming at computing function f programs it on a classical computer. To do so, she makes use of our *Quantum circuits* package, providing functions for writing and manipulating (families of) quantum circuits. She includes in her code some specifications, to get $f(x)$ as a result. Why3 indeed provides a specification language, enabling the annotation of code with, for example, pre- and post-conditions or loop invariants.

These specifications are logical predicates, built on a semantic interpretation of quantum circuits (the *Path-sum semantics* $[1,2]$, denoted by $(\!|\cdot|\!)$). In a nutshell, path-sums are an elegant representation for tensor product and tensor product combination in a complex vectorial space. The implementation of this semantics is deeply rooted in mathematical material written in Why3 language, that we specifically developed for this aim (package *Mathematical libraries*, consisting in definitions and lemmas for linear algebra, binary arithmetic, complex number theory, etc).

In Figure 3, the post-condition, that the semantics of the circuit $C(f)$ applied to x is equal to $f(x)$, is written

$$\mathit{requires}\{\,(\!|\mathit{result}|\!) = f(x)\,\} \tag{1}$$

where:

- *requires* is a Why3 keyword introducing post-conditions for a function
- *result* is a Why3 keyword designating the output of this function.

Proof obligation generation and support. From this annotated code, why3 compilation generates the proof obligations, which are mathematical formulas. In the Figure, the proof obligation generated from the specification in Equation 1 is given as

$$PO : \forall y.(\!|C(f)(y)|\!) = f(y) \tag{2}$$

Note that it is built by universally quantifying the variable x appearing free in Equation 1. It requires the semantics of $C(f)$ to be equivalent to f.

Now, to certify the program, one has to prove that this proof obligation is satisfied by the current logical context *Ct*. This logical context is made of:

- the content of the called Why3 modules. In our case they are the mathematical libraries developed for the definition and use of the path-sum semantics, forming a mathematical theory written *Th* in Figure 3,

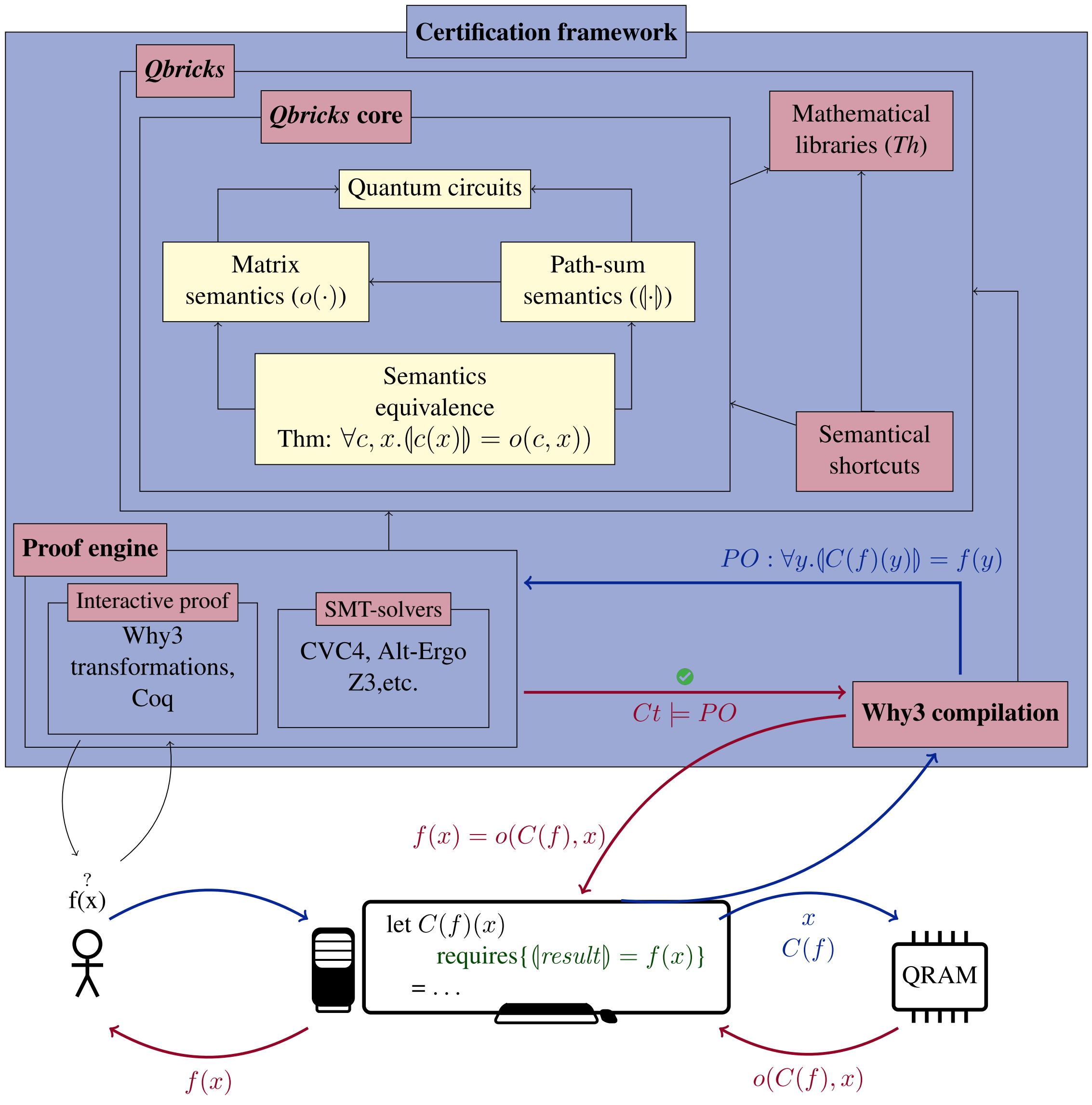
Certification framework
Qbricks
Qbricks core
Mathematical libraries (*Th*)
Quantum circuits
Matrix semantics ($o(\cdot)$)
Path-sum semantics ($(\!|\cdot|\!)$)
Semantics equivalence Thm: $\forall c, x. (\!|c(x)|\!) = o(c, x))$
Semantical shortcuts
Proof engine
Interactive proof
Why3 transformations, Coq
SMT-solvers
CVC4, Alt-Ergo Z3,etc.
$PO : \forall y. (\!|C(f)(y)|\!) = f(y)$
$Ct \models PO$
Why3 compilation
$f(x) = o(C(f), x)$
$\overset{?}{\text{f(x)}}$
let $C(f)(x)$ requires$\{(\!|result|\!) = f(x)\}$ $= \dots$
x
$C(f)$
QRAM
$f(x)$
$o(C(f), x)$

Figure 3: The Qram hybrid model.

- the specifications for functions defined in the current module. This is of importance as typically, specifications concern a quantum circuit C defined by combination (either sequential or parallel) of simpler quantum circuits $\{C_k\}$. Then, proof support for specifications of C can rely on the specified implementations of circuits $\{C_k\}$.

In addition to generating proof obligations, Why3 gives some means for this proof support. Indeed it enables to send proof obligations, together with their logical contexts, to a bench of SMT-solvers (CVC4, Alt-Ergo, Z3,etc.) for automatised verification.

Why3 also provides an user interface in which one can enter a series of interactive commands to simplify proof obligations so as to help SMT-solvers. A third possibility is to call the Coq proof assistant. Using these two last possibilities requires the agent to interact with provers.

Verified program. These proof support facilities establish that the generated proof obligation is a theorem of the current logical context (written $Ct \models PO$ in Figure 3).

In addition to path-sum semantics, *Qbricks* contains a complete implementation of the *matrix semantics*, which is the standard reference in quantum mechanics. So as to use it as a truth reference for our certified implementations, we developed it together with an implemented proof equivalence for path-sum semantics and with translation functions between these two semantics.

3 Achievements

Case study. Thanks to *Qbricks*, we implemented a parametrized, verified and scale invariant implementation of the phase estimation [4, 16] circuit. This algorithm is one of the major routines in quantum computing. It constitutes, for example, the purely quantum part in Shor algorithm [23] (integer factoring), but it is also at the heart of, e.g., HHL [12] logarithmic linear system solving algorithm or quantum simulation [9]. It takes as parameter a matrix U, an eigen-vector $|v\rangle$ and answers the eigenvalue of U corresponding to $|v\rangle$.

We propose an implementation of the phase estimation algorithm in the *core case*, i.e. when the phase is a dyadic multiple of π – this is the case where the algorithm returns an exact value instead of an approximation. The formal statement can be found in Table 1. The function inputs the black box U and the size n: these are the parameters. The function also inputs two ghost arguments v and k: they only serve in the specifications. The first pre-condition states that the number of qubits is non-negative, the second that k is between 0 and 2^n, and the third that circuit U has an eigen-vector v of eigenvalue $e^{i\frac{k}{2^n}\pi}$. The first post-condition states that the resulting circuit has n more wires than the input circuit, and the second states the ultimate desired specification: the result circuit actually computes $|k\rangle$.

The effective function is defined in line 8, calling functions defined in the implementation: function phase_estim is defined as a sequence of two *subcircuits*, phase_estim_pre and rev_place_Qft.

This overall specified implementation is 237 lines long, progressively building circuits from elementary gates to the function from Table 1. Its compilation generates 190 proof obligations, 166 of which are automatically supported by calls to SMT solvers. We proved the 24 non automated proof obligations by inserting 163 online transformations. These statistical data are reported in Table 2.

Conclusion : our contribution. In its actual state of development, our proposition contains:

- a core building circuit language, consisting in a library of Why3 functions. It enables to build and manipulate any (family of) quantum circuit(s), Listing 1: Phase estimation specification

```
let function phase_estim (U : circuit ) (n : int )
                        (ghost v : matrix complex ) (ghost k : int ) : circuit
  requires {$0 < n$}
  requires {$k \in [\![0, 2^n[\![$}
  requires {eigen($U, v, \frac{k}{2^n}$)}
  ensures {$\mathbf{s}_{result} = n + \mathbf{s}_U$}
  ensures {$result \cdot |0\rangle_n \otimes v = |k\rangle_n \otimes v$}
 =  sequence(phase_estim_pre($U, n, v, k, n$), rev_place_Qft($n, \mathbf{s}_U$))
```

Table 1: Core specification for phase estimation

Lines of code	237	Proof obligations (POs)	190
Number of definitions	12	Automated POs	166
Number of lemmas	2	Online transformations	163

Table 2: Statistical data about the *Qbricks* phase estimation implementation.

- an implementation of the mathematical theory supporting quantum circuit semantics. It is approximately 10 000 lines of code long, and contains more than 1 000 proved lemmas in algebra, arithmetic, set theory, complex number theory, bit vector operations, etc,
- the complete implementation of both the path-sum and the matrix semantics, together with their equivalence proof and translation functions.

References

[1] Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018.*, pages 1–21, 2018.

[2] Matthew Amy. *Formal Methods in Quantum Circuit Design*. PhD thesis, University of Waterloo, Ontario, Canada, 2019.

[3] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In *Proceedings 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015.*, pages 71–83, 2015.

[4] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, 1998.

[5] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[6] Richard P. Feynman. Simulating physics with computers. *j-INT-J-THEOR-PHYS*, 21(6–7):467–488, 1982.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European Symposium on Programming*. Springer, 2013.

[8] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. Qmc: A model checker for quantum systems. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 543–547, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Iulia M Georgescu, Sahel Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153, 2014.

[10] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In *International Conference on Reversible Computation*, pages 110–124. Springer, 2013.

[11] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *SIGPLAN*, 2013.

[12] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[13] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified optimization in a quantum intermediate representation. *CoRR*, abs/1904.06319, 2019.

[14] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified optimization in a quantum intermediate representation. *arXiv preprint arXiv:1904.06319*, 2019.

[15] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 1. ACM, 2014.

[16] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. *arXiv preprint quant-ph/9511026*, 1995.

[17] Igor L Markov and Mehdi Saeedi. Constant-optimized quantum circuits for modular multiplication and exponentiation. *arXiv preprint arXiv:1202.6614*, 2012.

[18] D Michael Miller, Robert Wille, and Rolf Drechsler. Reducing reversible circuit cost by adding lines. In *2010 40th IEEE International Symposium on Multiple-Valued Logic*, pages 217–222. IEEE, 2010.

[19] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[20] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices*, 2017.

[21] Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.

[22] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in coq. *arXiv:1803.00699*, 2018.

[23] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[24] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2(49):10–22331, 2018.

[25] Krysta M Svore, Alfred V Aho, Andrew W Cross, Isaac Chuang, and Igor L Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83, 2006.

[26] Krysta M Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *arXiv preprint arXiv:1803.00652*, 2018.

[27] Krysta M Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *arXiv preprint arXiv:1803.00652*, 2018.

[28] Dave Wecker and Krysta M Svore. Liquil>: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467*, 2014.

[29] Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. Model-checking linear-time properties of quantum systems. *ACM Trans. Comput. Logic*, 15(3):22:1–22:31, August 2014.