

TOWARD CERTIFIED QUANTUM PROGRAMMING

A PREPRINT

Christophe Chareton

LRI, Centrale Supélec, Université Paris-Saclay, France
christophe.chareton@lri.fr

Sébastien Bardin

CEA, LIST, Université Paris-Saclay
sebastien.bardin@cea.fr

François Bobot

CEA, LIST, Université Paris-Saclay
francois.bobot@cea.fr

Valentin Perrelle

CEA, LIST, Université Paris-Saclay
valentin.perrelle@cea.fr

Benoit Valiron

LRI, Centrale Supélec, Université Paris-Saclay
benoit.valiron@lri.fr

ABSTRACT

While recent progress in quantum hardware open the door for significant speedup in certain key areas, quantum algorithms are still hard to implement right, and the validation of such quantum programs is a challenge. Early attempts either suffer from the lack of automation or parametrized reasoning, or require the user to write specifications and algorithms far from those presented in research articles or textbooks, and as a consequence, no significant quantum algorithm implementation has been currently verified in a scale-invariant manner. We propose QBRICKS, the first development environment for certified quantum programs featuring clear separation between code and proof, scale-invariance specification and proof, high degree of proof automation and allowing to encode quantum programs in a natural way, i.e. close to textbook style. This environment features a new domain-specific language for quantum programs, namely QBRICKS-DSL, together with a new logical specification language QBRICKS-SPEC. Especially, *we introduce and intensively build upon HOPS, a higher-order extension of the recent path-sum semantics, used for both specification (parametrized, versatile) and automation (closure properties)*. QBRICKS builds on best practice of formal verification for the classic case and tailor them to the quantum case. To illustrate the opportunity of QBRICKS, we implement the first scale-invariant verified implementations of non-trivial quantum algorithms, namely phase estimation (QPE) – the purely quantum part of Shor algorithm for integer factoring – and Grover search. It proves by fact that applying formal verification to real quantum programs is possible and should be further developed.

1 Introduction

Quantum computing is a young research field. Indeed, its birth act is usually dated 1982, when Richard Feynman [1] raised the idea of simulating the quantum mechanics phenomena by storing information in particles and controlling them according to quantum mechanics laws. This initial idea has become one over many application fields for quantum computing (cryptography [2], deep learning [3], optimization [4, 5], solving linear systems [6], etc). In all these domains there are now quantum algorithms beating best known classical algorithm by either quadratic or even exponential factors. In parallel to the rise of quantum algorithms, the design of quantum hardware has moved from lab-benches [7] to programmable, 50-qubits machines designed by industrial actors [8, 9] reaching the point where quantum computers would beat classical computers for specific tasks [8]. This has stirred a shift from a theoretical

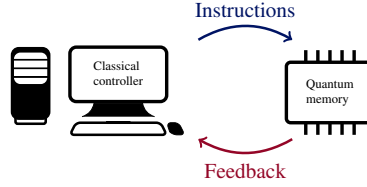


Figure 1: Scheme of the hybrid model

standpoint on quantum algorithms to a more programming-oriented view with the question of their concrete coding and implementation.

In this context, a particularly important problem is the adequacy between the mathematical description of the algorithm and its concrete implementation as a program.

A glimpse at quantum programming Quantum algorithms are commonly described within the *quantum co-processor model* [10], a.k.a. the hybrid model, where a *classical* computer controls a *quantum* co-processor holding a quantum memory (cf. Figure 1). The co-processor is able to apply a fixed set of elementary operations to update and query (*measure*) the quantum memory. Importantly, while measurement allows to retrieve classical (probabilistic) information from the quantum memory, it also modifies it (*destructive effect*). The state of the quantum memory is represented by a vector in a Hilbert space: the core of a quantum algorithm consists in successfully setting the memory in a specific *quantum state*.

Major *quantum programming languages* such as Liqui|⟩ [11], Q# [12], Quipper [13, 14], and the rich ecosystem of existing quantum programming frameworks [15] provide dedicated features for interacting with the quantum memory and well-suited for implementing quantum algorithms. They usually embed these features within a standard classical programming language, with forth (send quantum instructions) and backs (get measurement results) between the classical control loop (classic computer) and the quantum part (co-processor).

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.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} j\rangle U^j u\rangle$	
	$= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u} j\rangle u\rangle$	result of blackbox
4.	$\rightarrow \tilde{\varphi}_u\rangle u\rangle$	apply inverse Fourier transform
5.	$\rightarrow \tilde{\varphi}_u$	measure first register

Figure 2: Quantum phase estimation (QPE) algorithm [16, p. 225]

The problem with quantum algorithms A quantum algorithm, i.e. a sequence of elementary operations, is usually provided in the form of Figure 2. Starting from an initial state, the algorithm describes a series of high-level operations which, once composed, realize the desired state. Each high-level operation may itself be described in a similar way, until one reaches elementary operations. The description of the algorithm is therefore both the specification — the global memory-state transformation — and the way to realize it — the list of elementary operations, or *quantum circuit*.

A major issue is then to verify that the circuit generated by the code written as an implementation of a given algorithm is indeed a run of this algorithm.

The case for quantum formal verification While testing and debugging are the common verification practice in classic programming, they become extremely complicated in the quantum case. Indeed, debugging and assertion checking are virtually made impossible by the destructive aspect of quantum measurement, the probabilistic nature

of quantum algorithms seriously impedes system-level quantum testing and, finally, classic emulation of quantum algorithms is (strongly believed to be) intractable.

On the other hand, nothing prevents *a priori* the formal verification of quantum programs. Formal methods and formal verification [17] design a wide range of techniques aiming at proving the correctness of a system with absolute, mathematical guarantee – reasoning over all possible inputs and paths of the system, with methods drawn from logic, automated reasoning and program analysis. The last two decades have seen an extraordinary blooming of the field, with significant case-studies ranging from pure mathematics [18] to complete software architectures [19, 20] and industrial systems [21, 22]. In addition to offering an alternative to testing, formal verification has in principle the decisive additional advantages to both enable scale-invariant proof certificates and offer once-for-all absolute guarantees for the correction of programs.

We will focus on a formal technique called *deductive verification* [23–26], that, we argue, is well suited for quantum formal verification (cf. Section 3).

Goal and challenges *Our goal is to provide a programming framework for quantum computers together with a formal verification apparatus for certifying programs.* Such a framework should satisfy the following principles:

Close to algorithmic description: It should enable to specify and code algorithms in a way that directly matches their usual description from the literature, in order both to lower implementation & certification time and to increase confidence in the specification;

Separation of concerns: It should enable a clear distinction between the code and the specification, in order to decouple implementation from certification – in particular, specification should be optional and it should be possible to add it to a program at a later stage;

Scale-invariant proofs: It should allow scale-invariant specifications and proofs, so as to enable the generic certification of parametrized algorithms. This is crucial as quantum algorithms are always described as parametrized families of circuits;

Proof automation: It should, as far as possible, provide automatic proof means. Indeed, the certification of a program should be as painless as possible to the programmer in order to be adopted.

These requirements raise several challenges from the formal verification point of view. Indeed, while questions about semantic, properties, specification and efficient verification algorithms have been largely investigated in the standard case, everything remains to be done in the quantum case. For example:

- **non-standard data:** quantum algorithms rely heavily on *amplitudes* (generalization of probabilities to arbitrary *complex numbers*), not studied at all in standard verification;
- **second-order reasoning:** we are interested here in *parametrized circuit-building programs*, i.e. programs that do not describe fixed-size circuits but *families* of circuits. We therefore want to be able to assert — and check— specifications satisfied by such families.

The major scientific questions at stake here are: (1) How to specify quantum programs in a natural way? (2) How to support efficient proof automation for quantum programs?

As a matter of fact, prior works on quantum circuit formal verification do not fully reach these goals.

Prior attempts We summarize in Table 3 the state of the art against the requirements laid above for proving properties of *quantum circuits*. Efforts for proving properties of *general* quantum programs involving measurements are considered in Section 9. Model-checking approaches [27, 28] are fully automatic but highly scale-sensitive. Methods based on proof assistants (here, Coq), such as [29] or Qwire [30, 31], lack proof automation (interactive proving) and deeply mix code and specification. Moreover, the underlying matrix semantic makes it cumbersome to specify and prove programs – so far the approach only came with illustration by small examples (e.g. `coin_flip` [32], teleportation [29], etc). More recently, Matthew Amy developed a powerful framework for reasoning over quantum circuits, the path-sums semantics [33, 34], which can be made functional and which is closed with respect to the functional composition and the Kronecker product. Thanks to their good closure properties, path-sums can be used to prove the correction of problem instances larger than existing methods (up to 100 qubits). Yet, the method is still unable to address scale-invariance: being completely disconnected from the structure of the program describing the circuit, they can address one specific, fixed circuit, but they cannot handle a circuit parametrized by a problem instance.

Previous practical verifications of quantum implementations are reported in Figure 4, where the horizontal axis gives the structural complexity of the quantum circuits, and the vertical axis gives the size in qubits (∞ corresponds to the

	[27, 28]	[29]	[30]	[33, 34]	QBRICKS
• Separate specification from code	✓	✗	✓	✗	✓
• Scale invariance	✗	✓	✓	✗	✓
• Specifications fitting algorithm	✗	✗	✗	✓	✓
• Automate proofs	✓	✗	✗	✓	➔

Table 3: Formal verification of quantum circuits

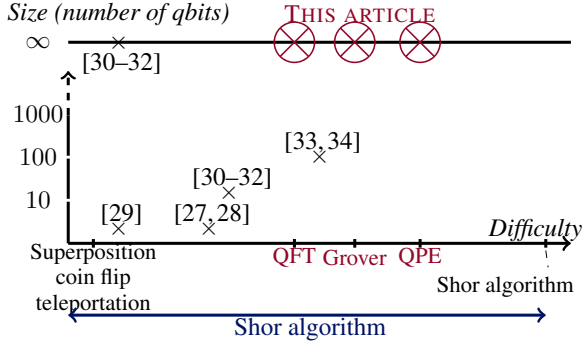


Figure 4: Certified quantum circuits from the literature

parametrized case). *No significant quantum algorithm (e.g., Shor) has been formally verified so far.* Also, the only existing scale-invariant proof concerns a toy example (the coin_flip protocol in [32]).

Finally, we indicate where our proposal stands: *we can verify QFT, Grover and QPE in a parametrized (scale-invariant) manner,* making a clear progress w.r.t. prior proofs.

Proposal and contributions We propose QBRICKS, the first development environment for certified quantum programs featuring clear separation between code and proof, scale-invariant specification and proof, high degree of proof automation and allowing to encode quantum programs in a natural way, i.e. close to textbook style. QBRICKS builds on best practice of formal verification for the classic case (separation of concerns, flexible logical specification language, proof automation, domain-based specialization) and tailors them to the quantum case.

More precisely, QBRICKS builds upon the domain-specific language QBRICKS-DSL together with the logical specification language QBRICKS-SPEC. QBRICKS-SPEC is expressive enough to offer *higher-order* certification of programs: certification of parametrized programs. The *key cornerstone* behind QBRICKS is the new notion of *higher-order path-sums (HOPS)*, an extension of the well-defined path-sum semantics enabling parametrized reasoning while keeping good closure properties – HOPS prove extremely useful both as a specification mechanism and as an automation mechanism.

In the end, we bring the following contributions:

Framework. A programming and verification framework, that is: on one hand, a core domain-specific language (QBRICKS-DSL) for describing families of quantum circuits, with an expressive power analogous to that of preexisting languages such as Quipper [14] or QWIRE [30]; on the other hand, a logical, domain-specific, specification language (QBRICKS-SPEC), tightly integrated with QBRICKS-DSL to specify properties of parametrized programs representing families of quantum circuits;

Higher-Order Path-Sums. A flexible semantics integrated with QBRICKS-SPEC and building upon the recent path-sum semantics [33, 34]. Our semantics, called *higher-order path-sums (HOPS)* (Section 3.3), retains the compositional and closure properties of regular path-sums while allowing *versatility* and *parametricity* of both specifications and proofs: HOPS expressions not only contain regular path-sum constructs but also general terms from QBRICKS-DSL. Especially, HOPS provides a unified and powerful way to reason about many essential quantum concepts (Section 3.4);

Automation This framework is embedded in the Why3 deductive verification tool [26, 35], providing proof automation mechanisms dedicated to the quantum case – this material is grounded in standard mathematics theories —linear algebra, arithmetic, complex numbers, binary operations, etc.— with 450+ definitions and 1,000+ lemmas (Table 12). The Why3 embedding comes with a series of semantic shortcuts designed to increase the overall level of proof automation, based on high-level composition rules and circuit subclasses with simple HOPS semantics (Section 6);

Case studies We present a scale-invariant proven implementation of *quantum phase estimation (QPE)* [36, 37] in Section 7. This algorithm is one of the major routines in quantum computing. It constitutes, for example, the purely quantum part in Shor algorithm [2] (integer factoring), but it is also at the heart of, e.g., HHL [6] logarithmic linear system solving algorithm or quantum simulation [38]. We also discuss several proofs of the Quantum Fourier Transform (QFT) —a subroutine of QPE, and we present a scale-invariant proven implementation of the Grover algorithm [39]. Altogether, this illustrates the genericity of our approach and

the strong interest of HOPS as a specification mechanism. *These implementations constitute the first scale-invariant, non-trivial proven quantum programs ever published.*

Discussion The scope of this paper is limited to proving properties of quantum circuits; although a goal we have in mind, we do not claim to support right now the interaction with the classical computer, nor the probabilistic side-effect resulting from the measurement. Also, we do not claim any novelty in the proofs of QPE or Grover by themselves, but rather the first parametric correctness proofs of the circuits produced by programs implementing QPE or Grover against their specifications. Note that these are second-order proofs, since the programs —and therefore the circuits— are parametrized by problem instances. Finally, it should be noted that QBRICKS manipulates circuits as objects (as in qPCF [40]) rather than as functions (as in Quipper [14]). In particular, qubits are not addressable objects. However, this is not a problem if one only considers circuits: quantum circuits are in general described as sequences of blocks.

That said, *we present the first non trivial, scale-invariant proofs of significant quantum programs – where prior works were limited to toy examples.* It proves by fact that applying formal verification to real quantum programs is possible and should be further developed. From a formal methods point of view, our results show that the general methodologies and tools developed for the classical case can be reused to a large extent to the quantum case, if properly tuned – setting the ground for further developments.

We want also to clarify the novelty of the development of deductive verification for quantum computation. Instead of a settled, off-the-shelf utility, deductive verification is a general framework. The fact that it could indeed be tamed for proving properties related to linear algebra over complex numbers was not clear up front. The fact that it was possible to shape it to play along so well with automated theorem provers was even a surprise to some of us. We believe it is clearly a novel result worth broadcasting.

2 Quantum Algorithms and Programs

In classical computing, the smallest piece of information lives in a discrete set consisting of one of two possible states (usually represented as 0 and 1). The equivalent in quantum computation [16] is the so-called *quantum bit* (or *qubit*), denoted in the Dirac notation with $|0\rangle$ and $|1\rangle$. However, unlike the conventional case, the state of a quantum bit is described by *amplitudes* over the two elementary states, i.e. linear combinations $\alpha_0|0\rangle + \alpha_1|1\rangle$ where α_0 and α_1 are any *complex values* satisfying $|\alpha_0|^2 + |\alpha_1|^2 = 1$. In general, a piece of quantum data is represented with a normalized vector in a finite-dimensional Hilbert space.

When joining classical data together, the state of the resulting system lives in the product space of the original state spaces. In quantum computation, it instead lives in the *Kronecker product* (or *tensor product*) of the original state spaces. In the case of two qubits, the tensor space is generated by the basis $|0\rangle \otimes |0\rangle$, $|0\rangle \otimes |1\rangle$, $|1\rangle \otimes |0\rangle$ and $|1\rangle \otimes |1\rangle$, usually represented as $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, or as $|0\rangle_2$, $|1\rangle_2$, $|2\rangle_2$ and $|3\rangle_2$.

Thus, while a classical register of n bits is in one of the 2^n possible combinations of n states in $\{|0\rangle, |1\rangle\}$, the state of a quantum register of n qubits is in any superposition of these 2^n combinations, that is any $|u\rangle_n = \sum_{k=0}^{2^n-1} \alpha_k |k\rangle_n$ such that $\sum_{k=0}^{2^n-1} |\alpha_k|^2 = 1$. Such a vector is called a *ket* and integer n is its length, written l_u .

2.1 Quantum operations

The core of a quantum algorithm consists in the manipulation of a register of qubits. Two main kinds of operations are used. The first kind consists in *quantum gates*: local operations on a fixed number of qubits, whose action consists in the application of a *unitary map* to the corresponding Hilbert space, i.e. a linear, bijective operation preserving the tensor product (i.e. preserving norm and orthogonality). The fact that unitary maps are bijective ensures that every unitary gate admits an *inverse*. Unitary maps are usually represented as *matrices* as the spaces are finite-dimensional. The other main kind of operations is *measurement*: the retrieval of classical information out of the quantum memory. This operation is probabilistic and modifies the global state of the system: measuring the n -qubit system $\sum_{k=0}^{2^n-1} \alpha_k |k\rangle_n$ returns n bits in state k with probability $|\alpha_k|^2$.

Quantum gates might be applied in sequence or in parallel: sequence application corresponds to map composition (or, equivalently, matrix multiplication), while parallel application corresponds to the Kronecker product, or tensor product, of the original maps – or, equivalently, the Kronecker product of their matrix representations.

2.2 Quantum circuits

In a similar way to classical Boolean functions, the application of quantum gates can be written in a diagrammatic notation: the quantum circuits. Qubits are represented with horizontal wires and gates with boxes. Circuits are built compositionally, from a given set of atomic gates and by a small set of circuit combinators, including: parallel and sequential compositions, circuit inverting, controlling, iteration, etc.

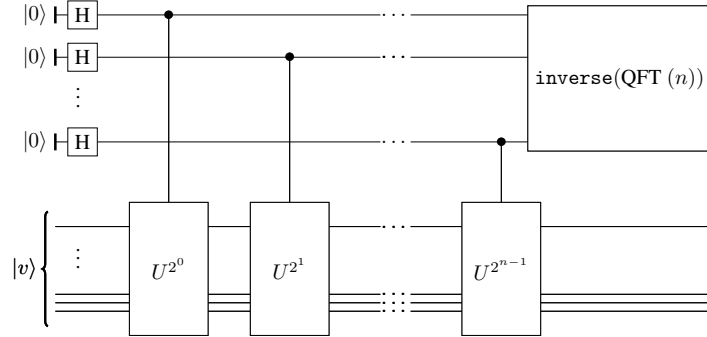


Figure 5: The circuit for QPE

As an example of a quantum circuit, we show in Figure 5 the bird-eye view of the circuit of the phase-estimation algorithm (QPE). Parametrized by n (size) and U (unitary), it is made of several parts. First, a register of n qubits is initialized in state $|0\rangle$, while another one is initialized in state $|v\rangle$. Then comes the circuit itself: a structured sequence of quantum gates. It uses the elementary unary Hadamard gate H and the circuits U^{2^i} and $\text{inverse(QFT}(n))$ (the reversed Quantum Fourier Transform). Both are defined as sub-circuits in a similar way. For the purpose of the current discussion, one should simply note two things. First, the circuit is made of parallel compositions of Hadamard gates and of sequential compositions of controlled U^{2^i} (the controlled operation is depicted with vertical lines and the symbol \bullet). Second, the circuit is *parametrized* by n and by U . This is very common: in general, a quantum algorithm constructs a circuit whose size and shape depend on the parameters of the problem. It therefore describes a *family of quantum circuits*.

2.3 Quantum algorithms

Quantum algorithms intend to solve classical problems in a probabilistic way. For example, the phase estimation algorithm (QPE) [36, 37] takes as parameter a matrix U , an eigenvector $|v\rangle$ and answers the eigenvalue of U corresponding to $|v\rangle$. If it is possible for the algorithm to answer in a deterministic way (we call this case QPE-core in the rest of the paper), in general it only answers it up to some probability. Another example that we study in this paper is Grover’s search algorithm [39]: given a sparse non-zero function $f : \{0 \dots 2^n - 1\} \rightarrow \{0, 1\}$, Grover’s algorithm outputs one value x such that $f(x) = 1$ with a probability high-enough to beat brute-force search. This is true in general: a quantum algorithm is usually designed in such a way that it solves the (classical) problem more efficiently than existing classical algorithms.

A quantum algorithm consists in the generation of a quantum circuit based on the parameters to the problem (e.g. the size of the instance), usually followed by an iteration of the following three steps: memory initialization, run of the quantum circuit, measure of the memory to retrieve a classical piece of data.

The quantum circuit is seen as a predictive tool that probabilistically gives some (classical) information from which one can infer the targeted result. The fact that the probability is high enough is a direct consequence of the mathematical properties of the unitary map described by the quantum circuit. The essence of the quantum algorithm —and the reason for its efficiency— consists in describing an efficient circuit realizing this unitary map.

Obtaining guarantees on the families of circuits realized by quantum programs is therefore of uttermost importance.

2.4 Path-Sum Semantics

Historically the semantics for quantum circuits has been given in term of unitary matrices [16] acting on Hilbert spaces, that is, the canonical mathematical formalism for quantum computation. If this semantics is well-adapted for

$$\begin{aligned}
PS(x) &::= \frac{1}{\sqrt{2}^n} \sum_{k=0}^{2^n-1} e^{\frac{2 \cdot \pi \cdot i \cdot P_k(x)}{2^m}} |\phi_k(x)\rangle \\
P_k(x) &::= x_i \mid n \mid P_1(x) \cdot P_2(x) \mid P_1(x) + P_2(x) \\
|\phi_k(x)\rangle &::= |b_1(x)\rangle \otimes \dots \otimes |b_n(x)\rangle \\
b_i(x) &::= x_i \mid \neg b(x) \mid b_1(x) \wedge b_2(x) \mid b_1(x) \oplus b_2(x) \mid \text{true} \mid \text{false}
\end{aligned}$$

Figure 6: Syntax for regular path-sums [33,34]

representing simple high-level circuit combinators such as the action of control or inversion, it is cumbersome for specifying the semantics of general circuits.

Quantum circuits make great use of complex values of modulus 1, which are values $e^{2\pi i\theta}$ for a given real parameter θ , which is itself a value in $[0, 1[$ (we call it a phase, or *angle*). Angles are invariant through translation: any such angle θ is equal to $\theta + 1$.

A recent semantics is building on this notion of angle and has been shown successful as a medium for proving equivalence of general quantum circuits: the so-called path-sum semantics [33,34]. Its strength is to formalize the notation used in e.g. Figure 2. The path-sum of a unitary matrix U is then written as $U : |x\rangle \mapsto PS(x)$ where x is a list of booleans. $PS(x)$ is defined with the syntax of Figure 6. The $P_k(x)$ are called *phase polynomials* while the $|\phi_k(x)\rangle$ *basis-kets*.

This semantics is closed under functional composition and Kronecker product. For instance, if V sends y to $PS'(y) = \frac{1}{\sqrt{2}^{n'}} \sum_{k=0}^{2^{n'}-1} \exp\left(\frac{2 \cdot \pi \cdot i \cdot P'_k(y)}{2^{m'}}\right) |\phi'_k(y)\rangle$, then $U \otimes V$ sends $|x\rangle \otimes |y\rangle$ to

$$\frac{1}{\sqrt{2}^{n+n'}} \sum_{j=0}^{2^{n+n'}-1} e^{\frac{2 \cdot \pi \cdot i \cdot (2^{m'} \cdot P_{j/2^n}(x) + 2^m \cdot P'_{j\%2^n}(y))}{2^{m+m'}}} |\phi_{j/2^n}(x)\rangle \otimes |\phi'_{j\%2^n}(y)\rangle$$

that is in the form shown in Figure 6.

However, if it has been shown successful to prove the equivalence of large circuit instances [34], its main limitation stands in the fact that each path-sum only address fixed-size circuit. Albeit a useful, compositional tool, as it stands it therefore cannot be used for proving properties of general, parametrized quantum programs.

This paper proposes an extension of path-sum semantics to address the certification of general quantum programs.

3 Our Proposal for Certification

A common way to describe quantum algorithms in the literature is to declare, or describe, either semi-formally or in natural language, a sequence of unitary operations to be implemented, inter-crossed with a sequence of formal assertions describing the evolution of the state of the system along the performance of these functions.

The example in Figure 2 illustrates this case: it corresponds to the exact description of the phase estimation algorithm (QPE) at it is written in [16, p.225]. The formal description of state, left column in Figure 2, is interpretable as specifications for the operations declared in the right column. For example, operation *create superposition* is declared on the right column, line 1. We interpret the formal expression of line 1, left (framed in blue), as its precondition and the one of line 2, left (framed in red), as its post-condition.

The problem solved with QBRICKS is to be able to insert — and prove — the formal assertions given in a program description such as Figure 2 in a concrete implementation. In particular, we target a *scalable* solution, where one can certify quantum algorithms seen as describing *families of circuits of arbitrary size*.

We want to emphasize the scope of the problem: QBRICKS is aimed at certifying families of circuits, representing unitary gates. In particular, QBRICKS is not designed for the classical and probabilistic interaction with the classical computer.

Our key observation here is that standard quantum algorithm descriptions match perfectly with the process of deductive program verification [23, 24], a well established formal method. .

3.1 Deductive verification for quantum programs

Deductive program verification [23–26] is probably the oldest formal method technique, dating back to 1969 [23]. In this approach, programs are *annotated* with *logical assertions*, such as pre- and post-conditions for operations or loop invariants, then so-called *proof obligations* are automatically generated (e.g., by the weakest precondition algorithm) in such a way that proving (a.k.a. discharging) them ensures that the logical assertions hold along any execution of the program. These proof obligations are commonly proven by help of proof assistants or automatic solvers.

In more details, for any function f , annotations by precondition pre and postcondition $post$ are to be understood as a contract for the implementation: the programmer commits to ensuring that the output of f satisfies $post$ for any input satisfying pre , which, in essence, translates into

$$\forall x. pre(x) \rightarrow post(f(x))$$

Suppose that the function f is defined, specified and verified. And suppose one defines and specifies a further function g , using a call $f(a)$ to f . This generates a new proof obligation. To fulfill it, the already verified proof obligation for f is assumed as an hypothesis.

Thanks to the computational structure of circuits, we can define them through functions systematically calling their subcircuits. The specification of a circuit can be derived through the specifications of its sub-circuits.

3.2 Rational for the design of QBRICKS

We adopt the methodology presented in Section 3.1 for the development of QBRICKS: it is equipped with a domain specific language (DSL) for describing circuits, a target datastructure for concretely representing circuits, and a set of specific logical constructs for expressing constraints.

Circuit representation The language QBRICKS is only aimed at implementing algorithms and specifications provided in the form of Figure 2. The circuits used in quantum algorithms act in general on contiguous blocks of memory registers and consist of simple compositions and hierarchical descriptions. Thus, unlike existing quantum programming languages such as Quipper [14] or QWIRE [30], for QBRICKS there is no need for complex wire manipulation.

Following this analysis, the low-level circuit-representation we choose as a target QBRICKS is akin to the one of qPCF [40]: a circuit is a simple compositional structure consisting of base gates and circuit combinators such as sequential and parallel composition, control, inversion, etc. These constructions are packaged withing a domain-specific language (DSL) aimed at describing families of circuits.

Semantics of quantum circuits As mentioned in Section 2.4, if path-sums offer a compositional specification framework, they address the case of fixed-size circuits. In particular, one cannot give specification to general, parametrized programs describing families of circuits.

QBRICKS proposes a solution to this limitation, by unifying what can be done with the matrix and the path-sum semantics. Our proposal is a *higher-order path-sum semantics (HOPS)*. On one hand, we keep the functional view on the action of circuits on quantum registers, making it suitable for deductive verification. On the other hand, we *extend* its syntax to support parametric circuit construction, high-level circuit combinators and reference to QBRICKS-DSL constructs.

Implementation We implement QBRICKS as a domain-specific language embedded in the Why3 deductive verification tool [26, 35], allowing to take advantage of its advanced programming, specification and verification features for classical programs. We add all quantum-related features on top of it.

3.3 Key elements behind QBRICKS

QBRICKS is structured as a domain-specific language (DSL), called QBRICKS-DSL, and a domain-specific logical specification language, called QBRICKS-SPEC.

Domain-specific language The DSL QBRICKS-DSL is an ML-style language with an (opaque) datatype `circ` as the medium to build and manipulate circuits. The core of QBRICKS-DSL can be presented as a simply-typed lambda-calculus, presented in Figure 7. On top of `circ`, the type system of QBRICKS-DSL features the type of integers `int` and the arrow-type. This type system is not exhaustive and is meant to be extended with usual constructs such as booleans, pairs, lists, and other user-defined inductive datatypes: its embedding into WhyML makes it easy to use such types.

Types	$A, B ::= \text{circ} \mid \text{int} \mid A \rightarrow B \mid \dots$
Circuit Terms	$C ::= \text{Ph}(\theta) \mid H \mid \text{CNOT} \mid R_z(\theta) \mid \text{parallel}(M, N) \mid \text{sequence}(M, N) \mid \text{invert}(M) \mid \text{control}(M) \mid \text{ancilla}(M) \mid \text{size}(M) \mid$
Terms	$M, N ::= C \mid n \mid \theta \mid x \mid \lambda x.M \mid MN \mid \text{let rec } f x = M \text{ in } N \mid \dots$

Figure 7: Syntax for QBRICKS-DSL

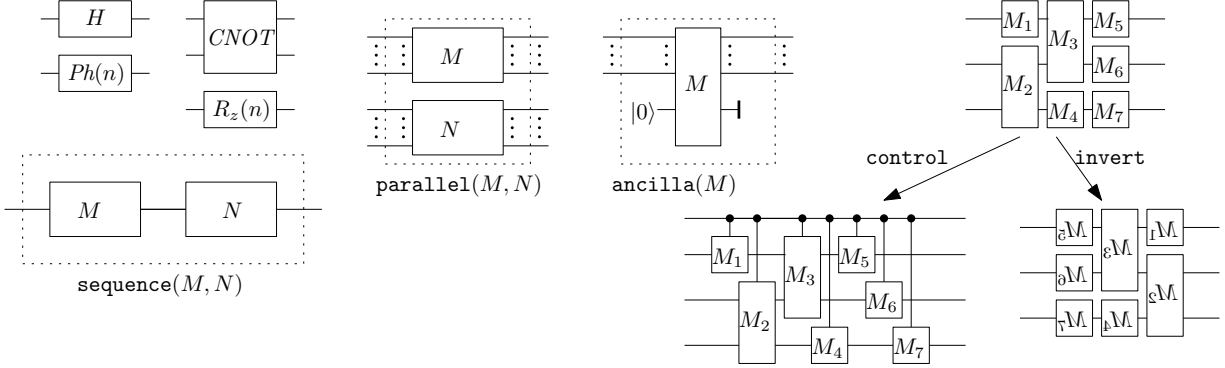


Figure 8: Circuit combinators

The core language constructs of QBRICKS-DSL consists of two main parts

- Circuit Terms, for the construction and manipulation of circuits (in blue). They consists of basic elementary gates and high-level combinators. See Figure 8 for a graphical presentation of the semantics of the combinators in term of circuit.
 - Basic elementary gates. $\text{Ph}(\theta)$ standing for the global phase $e^{2 \cdot i \cdot \theta} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ on one qubit, H for the Hadamard gate $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, CNOT for the control-not gate, and $R_z(\theta)$ for the phase shift around the Z -axis $\begin{pmatrix} e^{-2 \cdot i \cdot \pi \cdot \theta} & 0 \\ 0 & e^{2 \cdot i \cdot \pi \cdot \theta} \end{pmatrix}$. All of these constructs have type `circ`.
 - High-level circuit combinators. `parallel` stands for parallel composition of circuits, `sequence` for sequential composition, `invert` for the inversion, `control` for the controlling of circuits: `control(M)` takes the circuit built by M and control it using one additional line on top of it. Finally, `ancilla(M)` stands for the circuit built by M and where the last wire of M is initialized at $|0\rangle$ before the circuit M and discarded after. Finally, `size(M)` return the number of wires of M . Provided that their input is of type `circ`, the output of all combinators is `circ`, except `size` for which it is `int`.
- Regular ML-like constructs such as: integer constants (n, k, \dots), term variables (x, f, \dots), lambda terms ($\lambda x.M$), application (MN), and let-rec construction to permit recursion. As for types, this is not meant to be exhaustive, and the language can be equipped in the standard way with constructs to manipulate integers, booleans, lists, *etc.*

The typing rules are the usual ones for simply-typed lambda-calculus, and the semantics of QBRICKS-DSL is the usual operational semantics for the lambda-calculus part, and a standard circuit-building semantics for `circ` objects. The target low-level representation for a program of type `circ` is a circuit datastructure. It will be formalized in Section 4: for the purpose of this discussion it is enough to consider a `circ` type with constructors `parallel`, `sequence`, together with the constant gates.

Universality and usability of the chosen circuit constructs In QBRICKS-DSL, we use a restricted, small set of elementary circuit building blocks. For instance, we have not included the NOT-gate $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ nor the SWAP gate sending $|xy\rangle$ to $|yx\rangle$. This is a design choice: the chosen elementary gates are not meant to be convenient but simple to specify yet forming a universal set of gates: A *universal* (resp. *pseudo-universal*) set of elementary gates is such that they can be composed thanks to sequence or parallelism so as to perform (resp. approach arbitrarily close) any quantum unitary matrix.

Other, more convenient gates (such as the swap operation) can then be defined as macros on top of them. If one aims at using QBRICKS inside a verification compilation toolchain, these macros can for instance be the gates of the targeted architecture.

Validity of circuits A circuit is represented as a rigid rectangular shape with a fixed number of input and output wires. In particular, there is a notion of validity: a `circ` object only makes sense provided two constraints:

- in `sequence(C1, C2)`, the two circuits C_1 and C_2 should have the same number of wires (i.e. the same size). This is a simple *syntactic* constraint;
- in `ancilla(C1)`, the circuit C_1 should have $n + 1$ wires. Moreover, if given as input a vector where the last qubit is in state $|0\rangle$, its output should also leave this qubit in state $|0\rangle$. This condition is on the other hand a *semantic* constraint.

Note that even the syntactic constraints cannot be checked by a simple typing procedure, because of the higher-order reasoning involved here: the constraints must hold for any value of the parameters. All these syntactic and semantic constraints are thus expressed in QBRICKS-SPEC, our domain-specific logical specification language, and then sent as proof obligation to the proof engine.

Logical specification language QBRICKS-SPEC consists of a set of dedicated relations and functions, together with a language of algebraic expressions on top of a first-order logic, together with the otherwise standard logical libraries to express constraints coming from e.g. arithmetic, complex or real theories.

The formulas expressing semantical constraints used in QBRICKS-SPEC are of the form

$$\forall \vec{x} : \vec{A} \cdot P(\vec{x}) \longrightarrow (M : Exp_1 \mapsto Exp_2)$$

meaning “For all typed variables $x_1 : A_1, \dots, x_n : A_n$ of QBRICKS-DSL, provided that the property $P(\vec{x})$ is satisfied, then the circuit corresponding to the term M (of type `circ`) maps the algebraic expression Exp_1 to Exp_2 ”.

In particular, M is an open term of QBRICKS-DSL (of type `circ`) with free variables contained in \vec{x} ; $P(\vec{x})$ is a logical property that should be satisfied by a substitution of \vec{x} ; finally, Exp_1 and Exp_2 are algebraic expressions *parametrized with \vec{x}* . This is the main novelty compared with existing approaches: the logical specification language QBRICKS-SPEC is tightly integrated with the domain-specific, circuit description language QBRICKS-DSL, in the sense that QBRICKS-SPEC can natively invoke program parameters. This makes it possible to specify general programs describing families of circuits. The syntax of these algebraic expression is the so-called *higher-order path-sums (HOPS)*, defined below.

Note how \vec{x} has two roles: it is there both for the *parameter* describing the shape of the circuit and the *input* to the circuit.

Higher-order path-sum semantics (HOPS) In term of semantics of quantum programs, the main novelty of QBRICKS is to be able to reason on *open terms* seen as circuit description, parametrized programs. To do so, we build upon the recent proposal of path-sums [33,34]. We define the notion of *higher-order path-sums (HOPS)* in Figure 9. In the table, $P_k(\vec{x})$ is called the *phase polynomial*, $\text{range}(\vec{x})$ the *range* and $|\phi_k(x)\rangle$ the *basis ket*. In QBRICKS-SPEC our syntax for phase polynomials extends the standard definition by allowing general, open terms of type `int` in place of polynomial variable or constant, with free variables ranging in \vec{x} . In the original path-sum semantics, basis kets were defined as simple boolean polynomials. In QBRICKS-SPEC, we extend this syntax with the possibility to introduce an arbitrary open term N with free variables ranging in \vec{x} . This is useful e.g. when using oracles. This integration with QBRICKS-DSL brings two decisive advantages compared to the original path-sums:

- **Parametricity and compositionality.** Because of the sharing of term variables between QBRICKS-DSL terms and HOPS, QBRICKS-SPEC gives the ability to give specification to general programs describing families of circuits instead of fixed-size circuits. QBRICKS-SPEC opens path-sums to *higher-order* specification and verification while retaining the vertical and horizontal compositionality properties;
- **Versatility.** Thanks to the integration within a logical framework, QBRICKS-SPEC gives the ability to define—and reason—upon logical macros asserting useful constraints related to probabilities, eigenvalues, etc (See Section 3.4).

3.4 Specifying quantum programs with QBRICKS

It is possible in QBRICKS to give to functions pre- and post-conditions that are, respectively, requiring conditions on function arguments and ensuring properties for the results of functions. Along composition of functions, the pre- and post-conditions are chained as sketched in Section 3.1 to generate proof-obligations sent to the proof engine. We show the specifications for the native circuit combinators in Table 10 (we omit the one for `size` as it is trivial).

$$\begin{aligned}
 \text{HOPS}(\vec{x}) &::= \frac{1}{\sqrt{2^{\text{range}(\vec{x})}}} \sum_{k=0}^{2^{\text{range}(\vec{x})}-1} \exp\left(\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{n(\vec{x})}}\right) |\phi_k(\vec{x})\rangle \\
 P_k(\vec{x}) &::= N \mid P_1(\vec{x}) \cdot P_2(\vec{x}) \mid P_1(\vec{x}) + P_2(\vec{x}) \mid \\
 |\phi(\vec{x})\rangle &::= |N\rangle \mid |\phi_1(\vec{x})\rangle \otimes |\phi_2(\vec{x})\rangle
 \end{aligned}$$

Figure 9: Syntax for Higher-Order Path-Sum (HOPS)

<p>result = parallel(M_1, M_2)</p> <p>pre: $\text{size}(M_1) = n_1$ $\text{size}(M_2) = n_2$ $\forall \vec{x} \cdot (M_1 \cdot \vec{x}\rangle \mapsto \text{HOPS}_1(\vec{x}))$ $\forall \vec{y} \cdot (M_2 \cdot \vec{x}\rangle \mapsto \text{HOPS}_2(\vec{y}))$</p> <p>post: $\text{size}(\text{result}) = n_1 + n_2$ $\forall \vec{x} \cdot \forall \vec{y} \cdot (\text{result} \cdot \vec{x}\rangle \otimes \vec{y}\rangle \mapsto \text{HOPS}_1(\vec{x}) \otimes \text{HOPS}_2(\vec{y}))$</p>	<p>result = sequence(M_1, M_2)</p> <p>pre: $\text{size}(M_1) = n$ $\text{size}(M_2) = n$ $\forall \vec{x} \cdot (M_1 \cdot \text{HOPS}_1(\vec{x}) \mapsto \text{HOPS}_2(\vec{x}))$ $\forall \vec{x} \cdot (M_2 \cdot \text{HOPS}_2(\vec{x}) \mapsto \text{HOPS}_3(\vec{x}))$</p> <p>post: $\text{size}(\text{result}) = n$ $\forall \vec{x} \cdot (\text{result} \cdot \text{HOPS}_1(\vec{x}) \mapsto \text{HOPS}_3(\vec{x}))$</p>
<p>result = control(M)</p> <p>pre: $\text{size}(M) = n, m = 2^n,$ $\forall \vec{x} \cdot (M \cdot \vec{x}\rangle_m \mapsto \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \exp(\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{m'(\vec{x})}}) \phi_k(\vec{x})\rangle_m)$</p> <p>post: $\text{size}(\text{result}) = n + 1$ $\forall \vec{x} \cdot (\text{result} \cdot b\rangle_1 \otimes \vec{x}\rangle_m \mapsto \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \exp(\frac{2 \cdot \pi \cdot i \cdot P'_k(\vec{x})}{2^{m'(\vec{x})}}) b\rangle_1 \otimes \phi'_k(\vec{x})\rangle_m)$ where $P'_k(\vec{x}) = \text{if } b \text{ then } P_k(\vec{x}) \text{ else } 0$ and $\phi'_k(\vec{x}) = \text{if } b \text{ then } \phi_k(\vec{x}) \text{ else } \vec{x}$</p>	
<p>result = ancilla(M)</p> <p>pre: $\text{size}(M) = n + 1, m = 2^n,$ $\forall \vec{x} \cdot (M \cdot 0\rangle_1 \otimes \vec{x}\rangle_m \mapsto \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \exp(\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{m'(\vec{x})}}) 0\rangle_1 \otimes \phi_k(\vec{x})\rangle_m)$</p> <p>post: $\text{size}(\text{result}) = n$ $\forall \vec{x} \cdot (\text{result} \cdot \vec{x}\rangle_m \mapsto \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \exp(\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{m'(\vec{x})}}) \phi_k(\vec{x})\rangle_m)$</p>	
<p>result = invert(M)</p> <p>pre: $\forall \vec{x} \cdot (M \cdot \text{HOPS}_1(\vec{x}) \mapsto \text{HOPS}_2(\vec{x}))$ post: $\forall \vec{x} \cdot ((M \cdot \text{HOPS}_2(\vec{x}) \mapsto \text{HOPS}_1(\vec{x}))$</p>	

Table 10: Pre- and Post-conditions for the native circuit combinators

Extending the logical specification language The logical framework of QBRICKS-SPEC provided by HOPS gives the possibility to define many useful macros for expressing desired constraints such as probability, eigenvalues, and even algebraic operations on operators.

Eigenvectors and eigenvalues. One of the common need in the specification of quantum programs (and in particular in the case of QPE) is the need for asserting that a particular vector Exp_1 is an eigenvector of the unitary map described by some program M , with eigenvalue Exp_2 . In QBRICKS-SPEC we define a macro $\text{Eigen}(M \cdot Exp_1) = Exp_2$ being valid whenever Exp_2 is a complex number of norm 1 and $(M : Exp_1 \mapsto Exp_2 \cdot Exp_1)$

Probabilities. A quantum program is usually a probabilistic program: it returns the desired result with a probability that depends on the problem parameters (number of iteration, structure of the problem, etc). If one aims at fully specifying such programs, this probability therefore needs to be expressible as post-condition.

As recalled in Section 2.1, the probability of obtaining a result by a measurement is correlated with the amplitudes of the corresponding ket-basis vectors in the quantum state of the memory. In QBRICKS-SPEC we define a macro $\text{ProbMeas}(M \cdot Exp_1 \not\prec |N\rangle_m \otimes |\star\rangle_n) = Exp_2$ meaning that whenever measuring $M \cdot Exp_1$, a basis-ket vector of the form $|N\rangle_m \otimes |\star\rangle_n$ is reached with probability Exp_2 . One can define a similar macro to specify that the probability is *at least* Exp_2 .

<pre> result = QPE(C : circ, k : nat, n : nat) pre: n = k + size(C) 0 ≤ θ < 1 Eigen(C · y>_{n-k}) = e^{$\frac{2 \cdot \pi \cdot i \cdot \theta}{2^n}$} post: dyadic_k(θ) = z → ProbMeas(result · 0>_k ⊗ y>_{n-k} ↗ z>_k ⊗ ∗>) = 1 approx_k(θ) = z → ProbMeas(result · 0>_k ⊗ y>_{n-k} ↗ z>_k ⊗ ∗>) ≥ $\frac{4}{\pi^2}$ </pre>	<pre> result = Grover(f : {0...2ⁿ - 1} → {0,1}, k : nat) pre: ∃x · f(x) = 1 post: size(result) = n ProbMeas(result · 0>_n ↗ x>_k) = sin²(arcsin($\frac{\text{card}(f^{-1}(1))}{\sqrt{2^n}}$)) · (1 + 2 · k) </pre>
--	--

Table 11: Specifications for QPE and Grover

The instance with an equal sign can be defined as the logical formula

$$\left(M : Exp_1 \mapsto \frac{1}{\sqrt{2^{\text{range}(\vec{x})}}} \sum_{k=0}^{2^{\text{range}(\vec{x})}-1} e^{\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{\text{range}(\vec{x})}}} |\phi_k(\vec{x})\rangle_{m+n} \right) \longrightarrow$$

$$\forall K \cdot \text{partition}_{\text{range}(\vec{x})}(K) \longrightarrow Exp_2 = \frac{1}{2^{\text{range}(\vec{x})}} \sum_{I \in K} \left| \sum_{k \in I} e^{\frac{2 \cdot \pi \cdot i \cdot P_k(\vec{x})}{2^{\text{range}(\vec{x})}}} \right|^2$$

is valid, where $\text{partition}_n(K)$ means that K is a partition of $\{0 \dots 2^n - 1\}$ (also expressible as a logical formula in QBRICKS-SPEC).

Note how this formula is completely parametrized by the free variables \vec{x} : these can appear without restriction in the HOPS and in both the input ket-vector Exp_1 and the resulting measure Exp_2 .

Algebraic operations on operators. In the course of the proof of quantum specification, it is sometimes useful to be able to manipulate algebraic expressions containing building blocks such as **rotations** or **projectors**. In QBRICKS-SPEC, with the use of HOPS it is possible to define such objects as macros, and then prove algebraic equalities between them. The use of HOPS gives us the possibility to specify —and prove— equalities parametrized by problem instances. This is used extensively in the proof of the Grover specification.

Using these macros, it is then possible to give the specification of an implementation of QPE as shown in Table 11. The property $\text{dyadic}_k(\theta) = z$ states that z is the exact dyadic representation of θ on k bits, while $\text{approx}_k(\theta) = z$ states that z is *the best* one on k bits. The first post-condition corresponds to QPE-core, when the algorithm is deterministic, while the second post-condition is the general case. Similarly, as shown in Table 11 one can specify an implementation of Grover’s algorithm as follows. Here, k is the number of iterations.

3.5 QBRICKS’s proof engine

QBRICKS, through its host language Why3, provides means to validate proof obligations, either by sending them to a herd of automatic SMT-solvers (CVC4, Alt-Ergo, Z3, etc.), or to proof assistants (Coq, Isabelle/HOL). We also benefit from the interactive proof simplification mechanism from Why3.

Statistics about QBRICKS implementation are given in Table 12. Notice the volume of mathematical libraries in the overall development. The line *Generic functions* corresponds to definitions for derivated (families of) circuits that are recurrent patterns in quantum algorithmic (e.g. controlled operations, inverting circuits functions, etc.).

4 Building circuits in QBRICKS

In this Section we focus on the recursive building of quantum circuits and their mathematical interpretation in QBRICKS.

4.1 Conventions for circuit construction

QBRICKS provides various means to build and manipulate circuits and families of circuits. Three approaches can be followed. A first approach consists in progressively assembling *elementary gates* into more complex ones, just as bricks in a wall or a construction game (hence the name QBRICKS). A more generic method is to construct families of circuits: circuit parametrized by a size, a subcircuit, or a combination of both (cf. circuit of Figure 5). Providing such means is crucial in order to fit the (almost always parametrized) description of algorithms from the literature. Finally, QBRICKS allows one to build and use functional circuit combinators, such as controlling, iterating, inverting circuits (see Section 7.2), etc.

	Lines of code	Lemmas	Modules	Definitions
Mathematics libraries	14695	1614	77	328
Sets	532	59	4	14
Algebra	2091	190	10	37
Arithmetics	538	77	4	7
Binary arithmetics	1778	189	8	42
Complex numbers	2226	344	15	57
Quantum data	3335	310	12	68
Exponentiation	843	100	4	4
Iterators	861	72	6	30
Functions	259	33	3	8
Kronecker product	420	41	2	8
Unity circle	1812	199	9	53
Qbricks core	1357	50	5	35
Semantics reasoning	744	55	3	35
Generic functions	517	12	5	34
TOTAL	17313	1731	78	410

Table 12: Statistical data about QBRICKS implementation

4.2 The circuit type

As briefly mentioned in Section 3.3, in QBRICKS a quantum circuit is an object generated by sequential and parallel composition of elementary gates, consisting here of H , $Cnot$, $Ph(\theta)$ and $Rz(\theta)$. Internally, the angles θ are typed with a custom-type to enforce the fact that its value is within $[0, 1[$. A circuit has to respect a constraint: each occurrence of *sequence* in its structural decomposition links two sub-circuits acting on the same number of qubits (i.e. wires). The internal definition of the datastructure for quantum circuits then relies on an algebraic type `preCircuit`, with one constructor per elementary gate and constructors for both sequential and parallel composition:

```

type preCircuit =
  Phase(angle) | Rz(angle) | H | Cnot
  | Sequence(preCircuit, preCircuit)
  | Parallel(preCircuit, preCircuit)
    
```

On these `preCircuits` we recursively define function `size` as the number of qubits a `preCircuit` acts on and predicate `sequenceCorrect`, which assesses that any occurrence of a ‘Sequence’ composition links `preCircuits` of the same size.

The type `circuit` is then introduced as a record type with parameter a `preCircuit` respecting `sequenceCorrect`:

```

type circuit = {Pre : preCircuit}
  in variant {sequenceCorrect(Pre)}
  by{Pre = H}
    
```

It allows to define lifted versions of each quantum circuit constructors. As an example below is the definition for function `sequence`, lifting the constructor `Sequence`.

Listing 1: Definition for the pseudo constructor `sequence`

```

1 let function sequence(d, e : circuit )
2   requires {size(d) = size(e)}
3 = {Pre = Sequence(Pre(d), Pre(e))}
    
```

In this definition the bracketed syntax of line 3 enables to define a record object by its parameters: the result of applying `sequence` to `d` and `e` is the circuit whose `preCircuit` is `Sequence(Pre(d), Pre(e))`. In this paper we shall identify the constructors with their lifting counterpart

Thanks to these lifted constructors, while *automatically guaranteeing the well-formedness of the construction* we can recursively build quantum circuits by composing smaller quantum circuits, starting from elementary gates built by the lifted constructor for their `preCircuits`: this solves most of the syntactic constraints mentioned in Section 3.3. The recursive nature of the construction makes it possible to then define recursive semantics functions and predicates on circuits.

5 Higher-order path-sums and matrix semantics

As discussed in Section 3.2, the Why3 instantiation of QBRICKS implements the HOPS semantics of QBRICKS-SPEC. Nonetheless, in order to ensure the soundness of the presentation (and that, for instance, HOPS expressions indeed corresponds to unitary maps). QBRICKS contains an alternative, internal representation for circuits in terms of matrices. Considering its spread and consensual trust, we use the second one as the ground reference against which the correctness of HOPS is proved.

5.1 Matrix semantics

To any quantum circuit acting on n qubits, the matrix semantics associates a square matrix of size 2^n . The action of this circuit on a quantum register is then given by the matrix product of this matrix to the ket for this register.

Circuit matrices Matrices for circuits are built inductively: Parallel composition is interpreted by Kronecker product and sequential composition by matrix product: for any circuits d and e , the matrix $\mathbf{Mat}(\text{parallel}(d, e))$ is defined as $\mathbf{Mat}(d) \otimes \mathbf{Mat}(e)$, and, if $\text{size}(\text{Pre}(d)) = \text{size}(\text{Pre}(e))$ then the matrix $\mathbf{Mat}(\text{sequence}(d, e))$ is defined as $\mathbf{Mat}(d) \cdot \mathbf{Mat}(e)$.

Discussion We want to interpret quantum circuits as operators upon quantum kets. Matrix semantics provides a way to do so by multiplying the matrix for a circuit with a ket vector. This interpretation is correct but building these matrices and applying matrix product appear as a detour in comparison to directly dealing with ket transformation functions: this was the reason for the introducing of path-sum semantics [33, 34].

HOPS semantics also enables to simplify the management of indices bounds from matrices and to factorize intertwined sums of terms resulting from repeated sequence compositions. These aspects are of great interest when we come to formal proofs (see, e.g., the comparison in complexity data for certified implementation of QFT using each of these semantics, in Table 15).

5.2 Privileged HOPS

Higher-order path-sums (HOPS), as introduced in Figure 9 offers a very versatile specification environment, enabling both specifications and proofs for parametrized families of circuit and oracle manipulation.

Note that from the formalism of HOPS, a quantum circuits may be characterized by many different equivalent HOPS expressions. Quantum algorithm design also contains very low level steps, in which one describes the composition of a quantum circuit by assembling elementary gates. In such development steps (such as, in our case studies, designing the Quantum Fourier Transform), proof support may benefit from the compositionality of circuits and their semantics.

To ease the specification and proof in such cases, in the internal Why3 implementation of QBRICKS we propose a recursive definition assigning, for each circuit c , a unique HOPS expression, written $\mathbf{Ps}(c)$ called the privileged HOPS, that is proved correct and is determined by the algebraic structure of c . To do so, we separately implemented recursive definitions for three functions, which assemble together as a HOPS and fully characterize the action of a given circuit c over any quantum register:

- \mathbf{sR}_c (the sum range) is an integer parameter,
- $\mathbf{sC}_c(j, k)$ is the angle, determining a scalar value,
- $\mathbf{bK}_c(j, k)$ is an integer in $\llbracket 0, 2^n \rrbracket$ determining a basis ket of size n

Then, for any For any basis ket vector $|j\rangle_{s_c}$ and circuit c ,

$$(c : |j\rangle_{s_c} \mapsto \mathbf{Ps}(c, |j\rangle_{s_c}) = \frac{1}{\sqrt{2^{\mathbf{sR}_c}}} \sum_{k=0}^{2^{\mathbf{sR}_c}-1} e^{2\pi i(\mathbf{sC}_c(j,k))} |\mathbf{bK}_c(j, k)\rangle_{s_c}) \quad (1)$$

Specified circuit building functions Functions \mathbf{sR} , \mathbf{sC} and \mathbf{bK} being defined recursively, we use them decorate our preCircuit lifted constructor with adequate postconditions referring to each of these functions. As example below is the definition for sequence from Listing 1 enriched with these postconditions. For the sake of place and readability, in the listing we did not figure postconditions for the basis ket (\mathbf{bK}) and scalar (\mathbf{sC}) parameters. They depend only on the specifications for the privileged HOPS of the sub-circuits d and e , so that the specifications for the semantics of a sequence is completely determined by the semantics of its parameters. The same observation holds for parallel composition.

Listing 2: Specified definition for function sequence

```

1 let function sequence(d, e : circuit ) : circuit
2   requires { sd = sd }
3   ensures { sresult = sd }
4   ensures { sRresult = sRd + sRe }
5   ensures { (* a specification for bKresult *) }
6   ensures { (* a specification for sCresult *) }
7   = {pre = Sequence (pre d) (pre e) }

```

Once pseudo constructors are decorated with semantic postconditions, this decoration can be extended to circuit building functions using these primitives.

Compositionality and proof support One of the major interests in using such a compositional semantics for quantum circuits is that the proof for the specification of a circuit can rely on a recursive structure isomorphic to that of the specified circuit.

Indeed, at compilation within Why3, a QBRICKS function specification *spec* generates a proof obligations *PO* formalizing that *spec* is true in the current logical context. This logical context is made of both results established in dedicated modules (for QBRICKS in our *Mathematical libraries* : algebra, complex number theory, angle theory, binary arithmetic, etc.) and specifications for called functions, that are supposed fulfilled.

For example, in Listing 3, let us consider two circuits *d* and *e*. They are specified such as to have respective sum ranges *n* and *m*. Consider also circuit *f*, defined as sequence(*d, e*) and specified so as to have sum range *l*.

Listing 3: Illustration for compositional semantics

```

1 let function d : circuit
2   ensures { sRresult = n } (* further spec. *) = (* function d *)
3
4 let function e : circuit
5   ensures { sRresult = m } (* further spec. *) = (* function e *)
6
7 let function f : circuit
8   ensures { sRresult = l }
9   (* further spec. *) = sequence(d, e)

```

At compilation, the postconditions for *d, e* and function sequence are assumed in the current context, so that the proof obligation for postcondition in Line 9 of Listing 3 is

$$PO : T, \mathbf{sR}_d = n, \mathbf{sR}_e = m, \mathbf{sR}_{\text{sequence}(d,e)} = \mathbf{sR}_d + \mathbf{sR}_e, \text{result} = \text{sequence}(d, e) \models \mathbf{sR}_{\text{result}} = l$$

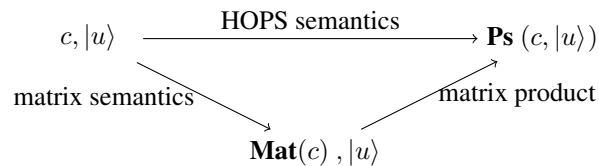
where *T* designates the rest of the logical context. More generally, the recursive composition of circuits brings, by the specifications of their components, the hypothesis that are required for proving the satisfaction of their specifications.

5.3 Matrix and HOPS semantics

In QBRICKS both matrix and HOPS semantics are proved equivalent, in the sense that for any circuit *c* and ket $|u\rangle$ of length \mathbf{s}_c , building matrix $\mathbf{Mat}(c)$ and multiplying it with $|u\rangle$ is the same as straightly computing the privileged HOPS $\mathbf{Ps}(c, |u\rangle)$.

Theorem 1. For any circuit *c* and ket $|u\rangle$ of length \mathbf{s}_c , we have $\mathbf{Mat}(c) \cdot |u\rangle = \mathbf{Ps}(c, |u\rangle)$. □

In other words, the following diagram commutes:



This equivalence is mainly used for assessing, in the universally known matrix semantics formalism, results that are proved in the better proof fitted HOPS semantics.

6 Simplifying semantic reasoning

We present now the different proof automation mechanisms we have developed in order to ease formal proofs and avoid interactive proof finalization as much as possible.

These mechanisms are of two kinds:

Composition rules. The relation between a circuit c and two kets $|u\rangle_{l_u}$ and $|v\rangle$ such that $s_c = l_u$ and $(c : |u\rangle \mapsto |v\rangle)$ satisfies nice composition properties, enabling easy handling specifications (Section 6.1);

Efficient circuit subclasses. Some subclasses of circuits (structural constraints) admit simplified HOPS expressions. We give the examples of *diagonal* and *flat* circuits (Section 6.2).

6.1 Composition rules

It translates sequence into function composition: for any circuits d, e having the same size and for any kets $|u\rangle, |v\rangle$ and $|w\rangle$, if $(d : |u\rangle \mapsto |v\rangle)$ and $(e : |v\rangle \mapsto |w\rangle)$, then

$$(\text{sequence}(d, e) : |u\rangle \mapsto |w\rangle)$$

HOPS furthermore commute with parallel composition and preserves linearity, enabling reasoning with ket basis decompositions.

These properties are particularly useful for dealing with semantic specifications that are stable through composition. In the following paragraphs we illustrate its use with eigenvalue reasoning and schemes to plug circuits within larger quantum registers.

Reasoning on eigenvalues and eigenvectors As discussed in Section 3.4, it is common in quantum circuit designing to reason about eigenvectors and eigenvalues for unitary operators (see, e.g., Section 7).

Specifications on eigenvalues enable to reason with circuits that are not entirely known. As an example, QPE (Section 7) takes as input a circuit c and a ket $|u\rangle$, such that there exists a value v such that $\text{eigen}(c \cdot |u\rangle) = v$. Then the algorithm outputs v . The circuit for this algorithm is built by specifying its output over $|0\rangle_n \otimes |u\rangle$, based on the assumption that $\text{eigen}(c \cdot |u\rangle) = v$ is satisfied. It satisfies the following composition lemma:

Lemma 1 (Composition of eigenvalues). *Let c, c' be quantum circuits, let $|u\rangle$ be a ket and let v, v' be complex values. Then:*

- *If $\text{eigen}(c \cdot |u\rangle) = v$ and $\text{eigen}(c' \cdot |u\rangle) = v'$ then $\text{eigen}(\text{sequence}(c, c') \cdot |u\rangle) = v * v'$*
- *If $\text{eigen}(c \cdot |u\rangle) = v$ then for any $k \geq 0$, $\text{eigen}(c^k \cdot |u\rangle) = v^k$.*

Placing circuits So far, we built QBRICKS circuits by composition of sub-circuits. Languages such as Quipper, Q#, LIQui], etc, have wire identifiers and build circuits by applying sub-circuits on these wires. In QBRICKS, such operation as *applying circuit c on wire x* is writable as a derived function, defined by help of parallel composition and identity operator $Id^{\otimes k}$ (which is itself defined, for any $k \geq 0$ by k parallel iterations of $\text{ph}(0)$):

Definition 1 (Function place.). *Let c be a quantum circuit and let k and n be positive integers such that $k + s_c < n$. Then $\text{place}(c, k, n) =_{\text{def}} \text{parallel}(\text{parallel}(Id^{\otimes k}, c), Id^{\otimes(n-k-(s_c))})$*

Note that this function requires the overall size n of the circuit as a parameter. In QBRICKS, *place* is defined with postcondition ensuring the satisfaction of the following lemma:

Lemma 2 (Place specification). *For all quantum circuit c and for all positive integers n, k such that $k + s_c < n$, for all kets $|x\rangle_k, |y\rangle_{n-k-s_c}, |u\rangle, |v\rangle$, if $(c : |u\rangle \mapsto |v\rangle)$ then*

$$(\text{place}(c, k, n) : |x\rangle_k \otimes |u\rangle \otimes |y\rangle_{n-k-s_c} \mapsto |x\rangle_k \otimes |u\rangle \otimes |y\rangle_{n-k-s_c})$$

6.2 Efficient circuit subclasses

We present now some sub-classes of quantum circuits with simplified HOPS semantics. These classes comes with a set of dedicated lemmas easing automation.

Diagonal circuits A diagonal circuit c of size n transforms any basis ket $|j\rangle_n$ into $e^{2\pi i \mathbf{dSc}_c(j)} |j\rangle_n$, where $\mathbf{dSc}_c(j)$ is an angle. In other words it admit a correct HOPS without sum and with identity basis ket:

$$\forall j. j \in \llbracket 0, 2^n \rrbracket \rightarrow (C : |j\rangle_n \mapsto \exp\left(\frac{2 \cdot \pi \cdot i \cdot P(j)}{2^n}\right) |j\rangle_n)$$

A diagonal circuit is completely specified by its size and function \mathbf{dSc} . Diagonally it composes with sequence and parallel by a simple addition of parameter \mathbf{dSc} . For example, if two diagonal circuits have the same size c , then for all $j \in \llbracket 0, 2^{s_d+s_e} \rrbracket$,

$$\mathbf{dSc}_{\text{sequence}(d,e)}(j) = \mathbf{dSc}_d(j) +_a \mathbf{dSc}_e(j)$$

Since rz and ph gates are diagonal, they generate a fragment of QBRICKS that is entirely diagonal. We also formally proved that property `diag` is preserved through application of functions `place` and controlled operations. Therefore it enables local specifications and verifications taking goods of these facilities: using diagonal HOPS, one does not have to care for sum range and basis kets in the specifications and proofs, which eases the practice (see, for example, the implementation data comparison for QFT in Table 15).

Flat circuits QBRICKS contains another similar simplified fragment, made of circuits with sum range equals to 0 (called *flat circuits*). This fragment is syntactically generated by gates rz, ph and cnot. Being less restrictive than `diag`, this condition offers an intermediate HOPS treatment (there is still no sum to deal with but there is a ket basis function) on a strictly larger fragment (since it includes gate cnot). The general HOPS for flat circuit is as follows:

$$\forall j. j \in \llbracket 0, 2^n \rrbracket \rightarrow (C : |j\rangle_n \mapsto \exp\left(\frac{2 \cdot \pi \cdot i \cdot P(j)}{2^n}\right) |\phi(j)\rangle_n)$$

Note that, unlike for diagonal circuit, flat circuit specifications requires a basis ket function ϕ . Still, the expression is simplified from the general case since there is no sum and therefore functions P and ϕ do not depend on the summed parameter k .

Integration in QBRICKS Being flat or diagonal are predicates over circuits expressed in the HOPS language. Flatness and diagonality can then be explicitly specified by the developer, allowing to take advantage of more powerful deduction rules.

7 Certifying QPE

In this section we illustrate QBRICKS usage together with its specification framework. To do so, we detail the writing and certification for our case study, the QPE algorithm. Experimental evaluation *per se* is presented in Section 8.

7.1 Introduction to QPE

All the eigenvalues of a unitary operator U are equal to $e^{2\pi i \Phi}$ for a given real Φ such that $0 \leq \Phi < 1$. Phase estimation (QPE) [36,37] is a procedure that, given a unitary operator U and an eigenvector $|v\rangle$ of U , finds the eigenvalue $e^{2\pi i \Phi_v}$ associated with $|v\rangle$. Since for any unitary operator U , any ket vector $|v\rangle$ is a weighted sum of eigenvectors for U , QPE enables to derives U 's spectral decomposition. It is a central piece in many emblematic algorithms, such as quantum simulation [38] or HHL algorithm [6] – resolution of linear systems of equations in time `POLYLOG`. QPE is also the central procedure in Shor's integer prime factor decomposition algorithm [2]. It consists in: (1) a reduction of the prime factor decomposition problem for integer N to the order-finding problem for exponentiation, modulo N , of an integer x co-prime to N , and (2) a solution to this problem, given by applying QPE to the unitary operator $U : |y\rangle \rightarrow |x \cdot y \bmod N\rangle$.

We implemented two different versions of QPE:

- In the first case (*core case*) we assume that Φ_v admits a binary writing with n bits. Then there is $\varphi_v \in \llbracket 0, 2^n \rrbracket$ such that $\Phi_v = \frac{\varphi_v}{2^n}$, and the eigenvalue associated with $|v\rangle$ is $e^{2\pi i \Phi_v}$ (also written $\omega_n^{\varphi_v}$). The goal is to seek this value φ_v and the algorithm deterministically outputs this value.
- In the second version (*general case*), no assumption is made over Φ_v which can take any real value such that $0 \leq \Phi < 1$. The goal is to seek the value $k \in \llbracket 0, 2^n \rrbracket$ that minimizes the distance $\Phi - \frac{k}{2^n}$ (modulo 1). The output of the algorithm is non deterministic. The proved specification is that it outputs \tilde{k} with probability at least $\frac{4}{\pi^2}$.

7.2 Walk-through

An overall view on the circuit for QPE was given on Figure 5. In Figure 13 we annotate the circuit and discuss its construction in the remainder of the section. The circuit uses two registers, one of size n and initialized to $|0\rangle_n$ and

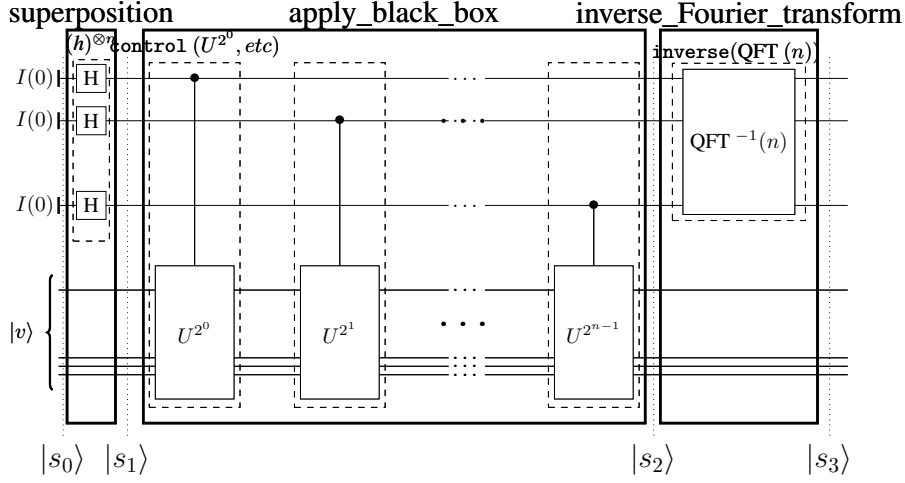


Figure 13: Modular structure of the circuit for QPE

one of size s and initially in state $|v\rangle_s$: $s_0 = |0\rangle_n \otimes |v\rangle_s$. We read this circuit from left to right, as follows.

Create superposition First, the superposition transformation is applied to the first register. It results in state

$$s_1 = \left(\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle_n \right) \otimes |v\rangle_s$$

The corresponding circuit is built as $\text{place}((h)^{\otimes n}, 0, n + s)$ where $(h)^{\otimes n}$ is the result of iterating parallel composition of the (lifted) Hadamard gate n times and place is the function from Definition 1, positioning the sub-circuit at the right vertical place.

Apply black box Then we perform, on the second register, a sequence of U gates elevated to the successive powers of 2 and controlled by qubits from the first register, written $\text{control}(U^{2^c}, c, n, n + s)$. Each of them transforms any ket $|j\rangle_n \otimes |v\rangle_s$ into $|j\rangle_n \otimes |v\rangle_s$ if $\bar{j}_c = 0$ and into $|j\rangle_n \otimes (e^{2\pi i \frac{1}{2^c} \Phi_v} \cdot |v\rangle_s)$ otherwise. By linearity of Kronecker product, in both cases;

$$(\text{control}(U^{2^c}, c, n, n + s) : |j\rangle_n \otimes |v\rangle_s \mapsto \omega_n^{\bar{j}_c \frac{1}{2^c} \Phi_v} |j\rangle_n \otimes |v\rangle) \quad (2)$$

Thus the whole series of controlled U^{2^k} operations (called circuit `apply_black_box`) applied to $|j\rangle_n \otimes |v\rangle_s$ results in state

$$\prod_{c=0}^{n-1} \omega_n^{\bar{j}_c \frac{1}{2^c} \Phi_v} |j\rangle_n \otimes |v\rangle \quad (3)$$

which rewrites into $\omega_n^{\overleftrightarrow{j} \Phi_v} |j\rangle_n \otimes |v\rangle$. By linearity, apply `apply_black_box` to s_1 outputs state $s_2 = s'_2 \otimes |v\rangle$, where

$$s'_2 = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} \omega_n^{\overleftrightarrow{j} \Phi_v} |j\rangle_n \quad (4)$$

Equation 2 comes by applying the second item of Lemma 1 and specifications of function `control`. Circuit `apply_black_box` is then built recursively over n by sequentially iterating circuits $\text{control}(U^{2^k}, n + s, n, k)$ composition iterations, with invariant

$$(\text{result} : |j\rangle_n \otimes |v\rangle_s \mapsto \omega_n^{\sum_{c=0}^k \bar{j}_c \frac{1}{2^c} \Phi_v} |j\rangle_n \otimes |v\rangle) \quad (5)$$

for k ranging from 0 to $n - 1$. Rewriting the ket expression in (5) as expression (2) is enabled by our mathematical libraries. Then, equality 4 is a straight application of the semantic linearity.

Apply inverse Fourier transform For any $n > 0$, QFT transforms any basis ket $|i\rangle_n$ into $\sum_{j=0}^{2^n-1} \omega_n^{j*} \omega_n^{\leftarrow *k}$. In particular, if applied to the first register of state $|\varphi_v\rangle_n \otimes |v\rangle$, it outputs state s_2 .

The circuit for QFT (n) is drawn in Figure 14, with the structure we used for our implementation: it is a recursive sequential composition of QFT_line sub-circuits. Each QFT_line is a sequence of an Hadamard gate and a recursive sequential composition $S - CR_n^k$ of gates $CR_n^{k,t} = cont(R_t, (k + t - 1), k, n)$, t ranging from 2 to $n - k$. For the sake of place we do not detail the specifications for this construction. For each positive integer t , R_t is a diagonal gate obtained by sequencing $ph(\frac{1}{2^{k+1}})$ and $rz(\frac{1}{2^{k+1}})$. Hence, gates $CR_n^{k,t}$ are specified using diagonal specifications from Section 6.2.

Function `invert` transforms any quantum circuit c into a circuit uncomputing any computation of c . It is introduced as a derived constructor in QBRICKS, defined by induction: it transforms, respectively, gates $ph(\theta)$ and $ry(\theta)$ into $ph(-_a(\theta))$ and $ry(-_a(\theta))$, it leaves gates h and `cnot` unchanged and it commutes with both sequential and parallel compositions. We proved that this construction ensures the correction of the semantics given in Table 10, that for a circuit c and for any kets $|u\rangle$ and $|v\rangle$, we have

$$(c : |u\rangle \mapsto |v\rangle) \leftrightarrow (\text{inverse}(c) : |v\rangle \mapsto |u\rangle)$$

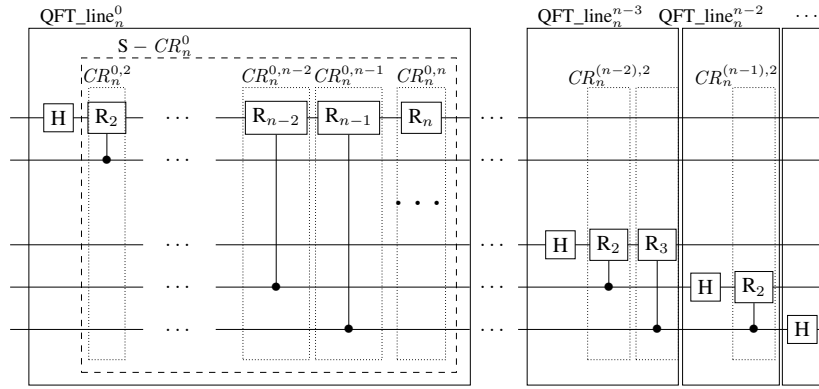


Figure 14: Quantum Fourier Transform

The final post-condition Applying `inverse(QFT n)` on the first register in s_2 results:

- In the core case, in state $|\varphi_v\rangle_n \otimes |v\rangle$, which enable to measures state $|\varphi_v\rangle_n$ on the first register with probability one,
- In the general case, in state

$$s_3 = \left(\frac{1}{2^n} \sum_{j=0}^{2^n-1} \sum_{i=0}^{2^n-1} \omega_n^{v*j*(\varphi-\bar{i})} |i\rangle_n \right) \otimes |v\rangle$$

Some numerical calculus and rewritings then derive the postcondition from Table 11 from this expression.

8 Experimental evaluation

We want to assess the practical relevance of our methodology and QBRICKS. More precisely, we consider the following Research Questions:

- (RQ1) Efficiency:** Does our approach work in practice on significant quantum programs, especially is it feasible to implement and verify (in a scale-invariant manner) QPE, and what is the cost in terms of annotations and proofs?
- (RQ2) Impact of automation support:** Is the technique automated enough (cf. Section 6), and how does it compare to other state of the art approaches from the automation point of view?
- (RQ3) Genericity:** Does the approach enable easy extensions? What is the price to pay for introducing new features, further developments?

Experiments were run on Linux, on a PC equipped with an Intel(R) Core(TM) i7-7820HQ 2.90GHz and 15 GB RAM. We used Why3 version 1.2.0 with solvers Alt-Ergo-2.2.0, CVC 3-2.4.1, CVC4-1.0, Z3-4.4.1.

8.1 Verification of QPE (RQ1)

Following the methodology described in Section 7, we indeed manage to implement and verify QPE with QBRICKS. Statistics are presented in Table 15, where lines figure the main steps of the implementation. We report several complexity criteria: numbers of lines of code, required definitions and lemmas, proofs obligations (POs) generated at compilation, POs automatically discharged by the underlying automated solvers (within time limit 5 seconds) and the number of interactive commands we entered to discharge the remaining POs and finalize the proof.

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
<i>create_superposition</i>	42	2	1	11	6	36
<i>apply_black_box</i>	57	3	1	50	44	46
QFT	75	3	0	57	51	30
QPE	63	4	0	72	65	51
Total	237	12	2	190	166	163

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

Table 15: Implementation & verification of QPE (Core case)

Conclusion (RQ1) QBRICKS did allow us to implement and verify in a scale-invariant manner the QPE algorithm, at a rather smooth cost. Tables 15 and give implementation for the core case: 237 lines of code, only 24 POs left for manual proof – we discharged them with 2 lemmas and 163 interactive commands (mainly calls for deduction rules – 79 – or hypotheses – 47–). Interestingly, around 87% of proof obligations were automatically discharged. In Table 16 we give corresponding data for the general case. Apart from dealing with angles having real numbers measures instead of dyadic fractions, it adds a series of rewritings for the output of the circuit (line *Measure and output rewritings*). Note that the rate of automatized proof obligation comes above 93% in this part.

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
<i>Inverse QFT</i>	75	3	0	57	51	30
<i>QPE circuit</i>	171	8	2	131	111	107
<i>Measure and output rewritings</i>	202	6	6	249	235	64
Total	448	18	8	437	397	201

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

Table 16: Implementation & verification of QPE (General case)

8.2 Impact of automation features (RQ2)

We implement the QFT module in three different manners: (1) with full QBRICKS abilities (especially, the automation features such as flat and diagonal HOPS), (2) only with basic HOPS and (3) only with the standard matrix semantics [30–32]. Note that we restrict ourselves to QFT because specifying the whole algorithm with matrix or basic HOPS semantics only would have been extremely painful. Results are given in Table 17.

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
QFT (full QBRICKS)	75	3	0	57	51	30
QFT (HOPS only)	87	3	0	73	64	49
QFT (matrix only)	200	8	15	306	285	106

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

Table 17: Comparison of several approaches, QFT algorithm

Conclusion (RQ2) The impact of our simplification mechanisms is clear: our method allows much smaller specifications and more automated proofs than the standard matrix semantic (lines: 75 vs. 200, remaining POs: 6 vs. 21). Moreover, our automation abilities allows a clear improvement against HPOS only (lines: 75 vs. 87, remaining POs: 6 vs. 9).

8.3 Genericity: verification of Grover algorithm (RQ3)

We also verified an implementation of the Grover algorithm, whose data are given in Table 18. Concretely, given a predicate p over integers, it takes as oracle an unitary operator outputting

- $|i\rangle_n$ if $p(i)$ is true,
- $-|i\rangle_n$ otherwise.

Then the specification shows that the circuit outputs, with high probability, an i such that $p(i)$ is true after a number of iterations that is linear in $\sqrt{2^n}$ (see the formal specifications in Table 11). The proof mainly consists in algebraic interpretation of HOPS: embeddings of HOPS into sub-vectorial space, projection, composition of reflections as rotation, composition of rotations, etc. This development highly relies on HOPS versatility and the possibility to manipulate all the main algebraic constructs in this formalism. In Table 11), line *Diffusion operator* corresponds to building the circuit for Grover diffusion operator, line *Rotations* concerns the algebraic interpretation of both the oracle and the diffusion operator, and line *Grover iterations* builds the sequential iteration of i occurrences of the oracle followed by the diffusion operator. Note that the rate of automation is higher than in our QPE implementation (96 % – 660 out of 696 – of proof obligations are automatically discharged). This is mainly due to the proof being largely made of algebraic reasoning.

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
<i>Diffusion operator</i>	149	8	2	157	144	63
<i>Rotations</i>	208	10	7	248	241	69
<i>Grover iterations</i>	97	4	0	281	275	37
Total	454	22	9	686	660	169

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

Table 18: Implementation & verification of Grover algorithm

Interestingly, it should be noted that while Grover relies on mechanisms and arguments significantly different from those of QPE, we were able to implement, specify and prove it without adding anything new to Qbricks – demonstrating the versatility of the platform and the deep interest of HOPS, for both specification and reasoning automation.

8.4 Efforts (RQ3)

The implementation of the core case of QPE was led along with the design and development of QBRICKS itself. Altogether it took us over 1.5 person.year. Interestingly, once this initial effort done, implementing both the general case of QPE and the Grover case study have been done within ten person.days each – with no modification of the framework. We thus have good reasons to believe that QBRICKS provides a generic and convenient environment for specifying, developing and proving quantum programs in a reasonably fast and easy way.

9 Related works

Formal verification of quantum circuits In the last couple of years, efforts have been led for introducing formal methods in quantum programming. Prior efforts regarding quantum circuit verification [27–34] have been described throughout the paper, especially in Section 1.

We build on top of these seminal works and propose the first certified development environment for quantum programs featuring clear separation between code & proof, scale-invariance specification and proof and high degree of proof automation.

Quantum Hoare logic Another explored direction tackles the formalization of *quantum programs with classical control*, where the input/outputs from the quantum co-processor are taken as oracles. It uses an extension of Hoare Logic, called Quantum Hoare Logic [41] and designed for the specification of quantum programs with classical control. Recently, the authors focused on the definition, automatic generation [42] and proof support [43, 44] for loop invariants in quantum programs invariants. In these works, quantum circuits are used as oracles, so that their verification is not addressed.

This approach may be complementary with the one presented here, for further developments of QBRICKS including probabilistic measures and the control flow of classical data.

Optimized compilation of circuits Formal methods and other program analysis techniques are also used in quantum compilation, in order to build highly optimized circuits [45–51]. This is a crucial current research area. Indeed, the quantum hardware available in the near future is expected to be highly constrained in terms of qubits, connectivity and quality: the so-called NISQ era [52–54].

ZX-calculus [55] represents quantum circuits by diagrams, to which one can apply a number of rewriting rules. This framework leads to a graphical proof assistant [56], certifying the semantic equivalence between circuit diagrams, with application to circuit equivalence checking and certified circuit compilation and optimization [57–59].

Other quantum applications of formal methods Huang *et al.* [60, 61] proposes a “runtime-monitoring like” verification method for quantum circuits, with an annotation language restricted to structural properties of interest (e.g., superposition or entanglement). Verification of these assertions is led by statistical testing instead of formal proofs.

Another more fundamental line of research is concerned with the development of specialized type systems for quantum programming languages. In particular, frameworks based on linear logic [62–64] and dependent types [30, 40] have been developed to tackle the non-duplicability of qubits and the constraints on the structure of circuits.

Finally, formal methods are also at stake for the verification of protocols using quantum information, such as cryptographic protocols [65–69].

10 Conclusion

In this article we presented QBRICKS, the first development environment for certified quantum programs featuring clear separation between code and proof, scale-invariance specification and proof, high degree of proof automation and allowing to encode quantum programs in a natural way. QBRICKS builds on best practice of formal verification for the classic case and tailor them to the quantum case. Especially, it introduces and intensively builds upon *higher-order path sums*, for both specification (parametrized, versatile) and automation (closure properties).

We implement the first scale-invariant verified implementations of non-trivial quantum algorithms, namely QPE – the purely quantum part of Shor algorithm – and Grover search, proving by fact that applying formal verification to real quantum programs is possible and should be further developed.

Acknowledgments This work was supported in part by the French National Research Agency (ANR) under the research project SoftQPRO ANR17-CE25-0009-02, and by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746- 484124.

References

- [1] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6–7):467–488, 1982.
- [2] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS’94)*, pages 124–134, Santa Fe, New Mexico, US., 1994. IEEE, IEEE Computer Society Press.
- [3] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195, 2017.
- [4] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, Joshua Lapan, Andrew Lundgren, and Daniel Preda. A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem. *Science*, 292(5516):472–475, 2001.
- [5] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. Available online as [arXiv:1411.4028](https://arxiv.org/abs/1411.4028), 2014.
- [6] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103:150502, Oct 2009.
- [7] Isaac L Chuang, Neil Gershenfeld, and Mark Kubinec. Experimental implementation of fast quantum searching. *Physical review letters*, 80(15):3408, 1998.
- [8] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

- [9] IBM Blog. On quantum supremacy. Blog Article¹, 2019.
- [10] Emmanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 1996.
- [11] Dave Wecker and Krysta M Svore. LIQUI|): A software design architecture and domain-specific language for quantum computing. Available online as [arXiv:1402.4467](https://arxiv.org/abs/1402.4467), 2014.
- [12] 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. Available online as [arXiv:1803.00652](https://arxiv.org/abs/1803.00652), 2018.
- [13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In Gerhard W. Dueck and D. Michael Miller, editors, *Proceedings of the 5th International Conference on Reversible Computation (RC'13)*, volume 7948 of *Lecture Notes in Computer Science*, pages 110–124, Victoria, BC, Canada, 2013. Springer.
- [14] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'13)*, pages 333–342, Seattle, WA, USA, 2013. ACM.
- [15] Quantum Computing Report. List of tools. Available online², 2019.
- [16] Michael A. Nielsen and Isaac Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, United Kingdom, 2002.
- [17] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.
- [18] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
- [19] Xavier Leroy et al. *The CompCert verified compiler. Documentation and user's manual*. INRIA Paris-Rocquencourt, 2012.
- [20] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Communication of the ACM*, 53(6):107–115, 2010.
- [21] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - a software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247, Thessaloniki, Greece, 2012. Springer.
- [22] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387, Toulouse, France, 1999. Springer.
- [23] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [24] Jean-Christophe Filliâtre. Deductive software verification. *STTT*, 13(5):397–403, 2011.
- [25] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
- [26] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, 2007. Springer.
- [27] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. QMC: a model checker for quantum systems. In Aarti Gupta and Sharad Malik, editors, *Proceeding of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 543–547, Princeton, NJ, USA, 2008. Springer.
- [28] Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. Model-checking linear-time properties of quantum systems. *ACM Transactions on Computational Logic*, 15(3):22:1–22:31, 2014.

¹<https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/>

²<https://quantumcomputingreport.com/resources/tools/>

- [29] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL 2015)*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–83, Oxford, UK, 2015. EPTCS.
- [30] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: a core language for quantum circuits. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, pages 846–858, Paris, France, 2017. ACM.
- [31] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in coq. In Bob Coecke and Aleks Kissinger, editors, *Proceedings 14th International Conference on Quantum Physics and Logic (QPL 2017)*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 119–132, Nijmegen, The Netherlands, 2017. EPTCS.
- [32] Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.
- [33] Matthew Amy. *Formal Methods in Quantum Circuit Design*. PhD thesis, University of Waterloo, Ontario, Canada, 2019.
- [34] Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In Peter Selinger and Giulio Chiribella, editors, *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018*, volume 287 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–21, Halifax, Canada, 2019. EPTCS.
- [35] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Proceedings of Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wroclaw, Poland, 53–64, 2011. Available online as [ha1-00790310](https://arxiv.org/abs/ha1-00790310).
- [36] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. Available online as [arXiv:quant-ph/9511026](https://arxiv.org/abs/quant-ph/9511026), 1995.
- [37] 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.
- [38] Iulia M Georgescu, Sahel Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153, 2014.
- [39] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 212–219, Philadelphia, Pennsylvania, USA, 1996. ACM.
- [40] Luca Paolini, Mauro Piccolo, and Margherita Zorzi. qPCF: Higher-order languages and quantum circuits. *Journal of Automated Reasoning*, 63(4):941–966, Dec 2019.
- [41] Mingsheng Ying. Floyd-hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19:1–19:49, 2011.
- [42] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. Invariants of quantum programs: characterisations and generation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, pages 818–832, Paris, France, 2017. ACM.
- [43] Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. Available as [arXiv:1601.03835](https://arxiv.org/abs/1601.03835), 2016.
- [44] Mingsheng Ying. Toward automatic verification of quantum programs. *Formal Aspects of Computing*, 31(1):3–25, 2019.
- [45] Alex Parent, Martin Roetteler, and Krysta M. Svore. REVS: a tool for space-optimized reversible circuit synthesis. In Iain Phillips and Hafizur Rahaman, editors, *Proceedings of the 9th International Conference on Reversible Computation (RC 2017)*, volume 10301 of *Lecture Notes in Computer Science*, pages 90–101, Kolkata, India, 2017. Springer.
- [46] Debjyoti Bhattacharjee, Mathias Soeken, Srijit Dutta, Anupam Chattopadhyay, and Giovanni De Micheli. Reversible pebble games for reducing qubits in hierarchical quantum circuit synthesis. In *Proceedings of the 49th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2019)*, pages 102–107, Fredericton, NB, Canada, 2019. IEEE.
- [47] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli. SAT-based exact synthesis: Encodings, topology families, and parallelism. To appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, <https://doi.org/10.1109/TCAD.2019.2897703>, 2019.

- [48] Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified compilation of space-efficient reversible circuits. In Rupak Majumdar and Viktor Kuncak, editors, *Proceedings of the 29th International Conference on Computer Aided Verification (CAV 2017), Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 3–21, Heidelberg, Germany, 2017. Springer.
- [49] Mathias Soeken, Thomas Häner, and Martin Roetteler. Programming quantum computers using design automation. Available online as arXiv:1803.01022, 2018.
- [50] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning about reversible quantum circuits. In Peter Selinger and Giulio Chiribella, editors, *Proceedings 15th International Conference on Quantum Physics and Logic (QPL 2018)*, volume 287 of *Electronic Proceedings in Theoretical Computer Science*, pages 299–312, Halifax, Canada, 2018. EPTCS.
- [51] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. Available as arXiv:1912.02250v1, 2019.
- [52] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.
- [53] Swamit S. Tannu and Moinuddin K. Qureshi. Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2019)*, pages 987–999, Providence, RI, USA, 2019. ACM.
- [54] Alexandru Paler. On the influence of initial qubit placement during NISQ circuit compilation. In Sebastian Feld and Claudia Linnhoff-Popien, editors, *Proceedings of the First International Workshop on Quantum Technology and Optimization Problems (QTOPNetSys 2019)*, volume 11413 of *Lecture Notes in Computer Science*, pages 207–217, Munich, Germany, 2017. Springer.
- [55] Bob Coecke and Aleks Kissinger. *Picturing quantum processes*. Cambridge University Press, Cambridge, United Kingdom, 2017.
- [56] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings for the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 326–336, Berlin, Germany, 2015. Springer.
- [57] Andrew Fagan and Ross Duncan. Optimising Clifford circuits with Quantomatic. In Peter Selinger and Giulio Chiribella, editors, *Proceedings of the 15th International Conference on Quantum Physics and Logic (QPL 2018)*, volume 287 of *Electronic Notes In Theoretical Computer Science*, pages 85–105, Halifax, Canada, 2018. EPTCS.
- [58] Niel de Beaudrap, Ross Duncan, Dominic Horsman, and Simon Perdrix. Pauli fusion: a computational model to realise quantum transformations from ZX terms. Available online as arXiv:1904.12817, 2019.
- [59] Aleks Kissinger and John van de Wetering. Reducing t-count with the ZX-calculus. Available online as arXiv:1903.10477, 2019.
- [60] Yipeng Huang and Margaret Martonosi. QDB: from quantum algorithms towards correct quantum programs. In Titus Barik, Joshua Sunshine, and Sarah Chasins, editors, *Proceedings of the 9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU@SPLASH 2018)*, volume 67 of *OpenAccess Series in Informatics (OASICs)*, pages 4:1–4:14, Boston, Massachusetts, USA, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
- [61] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture (ISCA 2019)*, pages 541–553, Phoenix, AZ, USA, 2019. ACM.
- [62] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16:527–552, 2006.
- [63] Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.
- [64] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.
- [65] Rajagopal Nagarajan and Simon Gay. Formal verification of quantum protocols. Available online as arXiv:quant-ph/0203086, 2002.
- [66] Alexandru Gheorghiu, Theodoros Kapourniotis, and Elham Kashefi. Verification of quantum computation: An overview of existing approaches. *Theory of Computing Systems*, 63(4):715–808, 2019.

-
- [67] Anne Broadbent. How to verify a quantum computation. *Theory of Computing*, 14(1):1–37, 2018.
 - [68] Urmila Mahadev. Classical verification of quantum computations. In Mikkel Thorup, editor, *Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2018)*, pages 259–267, Paris, France, 2018. IEEE Computer Society.
 - [69] Timothy AS Davidson. *Formal verification techniques using quantum process calculus*. PhD thesis, University of Warwick, 2012.