A Polytime Quantum Programming Language

Emmanuel Hainry, Romain Péchoux, Mário Silva

▶ **To cite this version:**

Emmanuel Hainry, Romain Péchoux, Mário Silva. A Polytime Quantum Programming Language. ACM Transactions on Quantum Computing, 2025, 7 (1), pp.1 - 32. ⟨10.1145/3769851⟩. ⟨hal-05419254⟩

HAL Id: hal-05419254

https://inria.hal.science/hal-05419254v1

Submitted on 16 Dec 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

A Polytime Quantum Programming Language

EMMANUEL HAINRY, ROMAIN PÉCHOUX, and MÁRIO SILVA, Université de Lorraine, CNRS, Inria, LORIA, France

As quantum computing emerges as a promising computational paradigm, quantum programming languages provide the tools that bridge the distance between abstract programming and its hardware implementation. In some cases, restricted programming languages may even provide an avenue for more efficient circuit compilation strategies.

In this work, we introduce FOQ, a first-order quantum programming language which allows for quantum control and recursion, and where a syntactically restricted subset of programs (PFOQ) is shown to be sound and complete for quantum polytime computation. This is achieved by bounding both the recursion depth and the branching width of programs, which we demonstrate to still be compatible with various interesting applications, such as quantum teleportation and the quantum Fourier transform.

PFOQ constitutes the first programming-language-based characterization of the quantum complexity class FBQP, and we provide a semantics-preserving compilation algorithm such that any PFOQ program can be compiled into a quantum circuit that grows polynomially on its number of input qubits, using an *anchoring-and-merging* technique to solve the problem of branch sequentialization.

CCS Concepts: • **Theory of computation → Quantum complexity theory**; **Complexity theory and logic**.

ACM Reference Format:
Emmanuel Hainry, Romain Péchoux, and Mário Silva. 2025. A Polytime Quantum Programming Language. *ACM Trans. Quantum Comput.* 7, 1, Article 4 (November 2025), 32 pages. https://doi.org/10.1145/3769851

1 Introduction

1.1 Background and Motivation

Quantum computation envisions the development of algorithms which leverage the laws of quantum physics in order to provide an advantage over classical strategies. This interest has lead to a plethora of examples where quantum programs, in theory, outperform what is achievable classically, such as integer factorization [Shor 1994], simulation of physical systems [Childs et al. 2018], search problems [Childs et al. 2003; Grover 1996], cryptography [Bennett and Brassard 2014; Ekert 1991] and teleportation [Bennett et al. 1993]. Similarly, different models of quantum computation have emerged, such as quantum Turing machines (QTMs) [Bernstein and Vazirani 1997; Ozawa and Nishimura 2000; Yamakami 1999], quantum circuits [Feynman 1982; Nielsen and Chuang 2011], measurement-based quantum computers [Briegel et al. 2009; Danos and Kashefi 2006] and circuit description languages [Ross 2015]. This variety of low-level models can be an asset in the creation of new algorithms but it can also be a difficulty for the developer when reasoning about the properties of their program.

The aim of quantum programming languages, starting with the seminal paper [Selinger 2004], is to allow the expressiveness required for creativity in algorithm development, while also restricting the user to instructions that are valid in a quantum computer, namely those that are reversible and, in particular, unitary. One may even aim for more than unitarity, and construct a programming language that ensures even more desirable properties, such as termination or time complexity

Authors' Contact Information: Emmanuel Hainry, hainry@loria.fr; Romain Péchoux, pechoux@loria.fr; Mário Silva, mmachado@loria.fr, Université de Lorraine, CNRS, Inria, LORIA, Nancy, France.

2025. ACM 2643-6817/2025/11-ART4
https://doi.org/10.1145/3769851

bounds. This is related to the field of *implicit computational complexity*, where one seeks to characterize complexity classes by placing syntactical restrictions (e.g. a type system) on programming languages or other paradigms of computation [Bellantoni and Cook 1992; Dal Lago 2011; Péchoux 2020].

In employing a restricted language, the developer is guaranteed that any program they write can be statically (and efficiently) verified to satisfy certain given properties. In this line of work, [Dal Lago et al. 2010] provided the first implicit characterization of the quantum polynomial-time classes for decision problems (EQP, BQP and ZQP). With no explicit mention of polynomial-time functions in its syntax, the authors in [Dal Lago et al. 2010] show that any term in their classically-controlled quantum lambda calculus can be evaluated in polynomial time. Conversely, any problem decidable by a QTM in polynomial time can be encoded into a corresponding term that simulates it.

Our work is greatly inspired by a result of [Yamakami 2020], who described a function algebra characterization of the class FBQP, i.e. functions which can be approximated to bounded-error probability by a QTM in polynomial time. In [Yamakami 2020], the author shows that with relatively simple constructions (base functions, composition, a quantum control case and a restricted recursion scheme) one can approximate precisely all the functions contained in FBQP.

To our knowledge, [Dal Lago et al. 2010; Yamakami 2020] constitute the only results concerning implicit characterizations of quantum polynomial time classes. This work focuses on two main areas of improvement: *implementation* and *expressivity*. While [Dal Lago et al. 2010; Yamakami 2020] both restrict their programs to those simulatable by a polytime QTM, they do not provide any *direct* compilation strategy which ensures polynomial size of the corresponding circuits. Given the constructive equivalence result between QTMs and quantum circuits in [Yao 1993], this is not a theoretical issue but more of a practical one: for a programming language, one would want a compilation strategy that preserves efficiency (i.e. polynomial time) without requiring an intermediate QTM representation. We provide such a compilation strategy in this work, and detail the circuit construction with examples.

From this point-of-view, our work can be seen as a contribution to the fields of *resource-aware* compilation techniques for quantum computation [Häner et al. 2020; Meuli et al. 2020; van de Wetering et al. 2025], and cost analysis of quantum programs [Avanzini et al. 2024, 2022; Colledan and Dal Lago 2024, 2025].

Regarding the second point, a language which is restricted to polynomial time should still aim to be as expressive as possible, to allow for creativity in algorithm design. While FOQ, the language introduced in this paper, and its polytime fragment PFOQ, are inspired by the function algebra in [Yamakami 2020], we claim that it is less constraining in algorithm design. To this end, we provide a number of notable examples which demonstrate FOQ and PFOQ's expressive power.

1.2 Contribution

This paper contains the following work:

- We introduce a quantum programming language, named FOQ, that includes first-order recursive procedures. The input of a FOQ program consists in a sorted set of qubits – a list of pairwise distinct qubit indexes – over which a program can apply its basic operations, corresponding to unary unitary operators, chosen to form a universal set of quantum gates.
- After showing that terminating FOQ programs are reversible (Theorem 1), we restrict programs to a strict subset, named PFOQ, for *polynomial time* FOQ. The restrictions put on PFOQ programs are tractable (i.e., can be decided in polynomial time, see Theorem 2), ensure that programs terminate on any input (Lemma 1), and prevent programs from having any exponential blow up (Lemma 2).

- We show that the class of functions computed by PFOQ programs is *sound* and *complete* for the quantum complexity class FBQP, which is the functional extension of *bounded-error quantum polynomial time*, known as BQP [Bernstein and Vazirani 1997], the class of decision problems solvable by a quantum computer in polynomial time with an error probability of at most $\frac{1}{3}$ for all instances. Hence, PFOQ is, to our knowledge, the first programming language characterizing quantum polynomial time functions. Soundness (Theorem 3) is proved by showing that any PFOQ program can be simulated by a quantum Turing machine running in polynomial time [Bernstein and Vazirani 1997], and completeness (Theorem 6) is demonstrated by showing that PFOQ programs encompass Yamakami's function algebra, known to be FBQP-complete [Yamakami 2020].
- We describe a compilation algorithm, named **compile**, that for any PFOQ program P and input size n, outputs a circuit of polynomial size in n that simulates P on n qubits (Theorem 10). The **compile** algorithm is composed of two subroutines, Algorithms 1 and 2, which perform *anchoring* and *merging* of recursive calls to ensure polynomial time, as described in Section 5. We show that, in our setting, circuits can be efficiently computed and that the **compile** algorithm is tractable (Theorem 9).

This work is an extended version of [Hainry et al. 2023] presented at FoSSaCS 2023. This version contains all the proofs that the previous version could not contain, as well as the following further contributions:

- We prove that the compilation technique is semantics-preserving (Theorem 10) by using the fact that the orthogonality condition is an invariant property of **optimize** (Lemma 5).
- We describe how the **compile** algorithm and its subroutine **optimize** (Algorithm 2) tackle the problem identified in [Yuan and Carbin 2022] as *branch sequentialization*, the fact that the time complexity of a quantum branch in the circuit model is the sum of the branches and not the maximum. We show that this problem can be handled at the compiler level for any PFOQ program, and therefore we increase the space of quantum polytime programs that can be efficiently compiled into circuits.
- In Section 5.4, we detail a compilation example to give a better intuition of the **optimize** subroutine, which constitutes the difficult part of our compilation strategy.

The leading example for demonstrating the syntax and expressivity of PFOQ will be the quantum Fourier transform introduced in Example 1, with the additional example of quantum teleportation in Example 2 (see Section 2).

A software artifact that implements the compilation technique described in this work is available at https://gitlab.inria.fr/mmachado/pfoq-compiler.

2 First-order quantum programming language

2.1 Syntax and Well-Formedness

We consider a quantum programming language, called FOQ for First-Order Quantum programming language, that includes basic data types such as Integers, Booleans, Qubits, Operators, and Sorted Sets of qubits, which are lists of finite length where all elements are different. A FOQ program has the ability to call first-order (recursive) procedures taking a sorted set of qubits as a parameter. Their syntax is provided in Figure 1.

Let x denote an integer variable and $\bar{\mathrm{p}}$, $\bar{\mathrm{q}}$ denote sorted sets variables. The size of the sorted set stored in $\bar{\mathrm{q}}$ will be denoted by $|\bar{\mathrm{q}}|$. We can refer to the i-th qubit in $\bar{\mathrm{q}}$ as $\bar{\mathrm{q}}[i]$, with $1 \leq i \leq |\bar{\mathrm{q}}|$. Hence, each non-empty sorted set variable $\bar{\mathrm{q}}$ can be viewed as a list $[\bar{\mathrm{q}}[1], \dots, \bar{\mathrm{q}}[|\bar{\mathrm{q}}|]]$. The empty sorted set, of size 0, will be denoted by nil and $\bar{\mathrm{q}} \ominus [i]$ will denote the sorted set obtained by removing the

(Integers)	i	$\triangleq$	$n \mid \mathrm{x} \mid \mathrm{i} + n \mid \mathrm{i} - n \mid \lvert\mathrm{s}\rvert$, with $n \in \mathbb{N}$
(Booleans)	b	$\triangleq$	$\mathrm{i} > \mathrm{i} \mid \mathrm{i} \geq \mathrm{i} \mid \mathrm{i} = \mathrm{i} \mid \mathrm{b} \wedge \mathrm{b} \mid \mathrm{b} \vee \mathrm{b} \mid \neg \mathrm{b}$
(Sorted Sets)	s	$\triangleq$	$\mathrm{nil} \mid \bar{\mathrm{q}} \mid \bar{\mathrm{p}} \mid \mathrm{s} \ominus [\mathrm{i}]$
(Qubits)	q	$\triangleq$	$\mathrm{s}[\mathrm{i}]$
(Operators)	$\mathrm{U}^f(\mathrm{i})$	$\triangleq$	$\mathrm{NOT} \mid \mathrm{R}_Y^f(\mathrm{i}) \mid \mathrm{Ph}^f(\mathrm{i})$, with $f \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$
(Statements)	S	$\triangleq$	**skip** $\mid \mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});\ \mid \mathrm{S}\ \mathrm{S} \mid$ **if** b **then** S **else** S
			$\mid$ **qcase** q **of** $\{0 \to \mathrm{S}, 1 \to \mathrm{S}\} \mid$ **call** proc[i](s);
(Procedure declarations)	D	$\triangleq$	$\varepsilon \mid$ **decl** proc[x]($\bar{\mathrm{p}}$){S}, D
(Programs)	$\mathrm{P}(\bar{\mathrm{q}})$	$\triangleq$	D :: S

Fig. 1. Syntax of FOQ programs.

qubit of index i in $\bar{\mathrm{q}}$. For notational convenience, we extend this notation by $\bar{\mathrm{q}} \ominus [i_1, \dots, i_k]$, for the list obtained by removing the qubits of indexes $i_1, \dots, i_k$ in the sorted set $\bar{\mathrm{q}}$.

The language also includes some constructs U^f to represent (unary) unitary operators, for some total function $f \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$. The function f is required to be polynomial-time approximable: its output is restricted to $\tilde{\mathbb{R}}$, the set of real numbers that can be approximated by a Turing machine for any precision 2^{-k} in time polynomial in k [Ko and Friedman 1982].

A FOQ *program* $\mathrm{P}(\bar{\mathrm{q}})$ consists of a sequence of *procedure declarations* D followed by a *program statement* S ∈ Statements, ε denoting the empty sequence. In what follows, we will sometimes refer to program $\mathrm{P}(\bar{\mathrm{q}})$ simply as P. Let *var*(S) be the set of variables appearing in the statement S. Let $|\mathrm{P}|$ be the size of program P, that is the total number of symbols in P.

A procedure declaration **decl** proc[x]($\bar{\mathrm{p}}$){S} takes a sorted set parameter $\bar{\mathrm{p}}$ and some optional integer parameter x as inputs. S is called the *procedure statement*, proc is the *procedure name* and belongs to a countable set Procedures. We will write $\mathrm{S}^{\mathrm{proc}}$ to refer to S and proc ∈ P holds if proc is declared in D.

Besides the no-op instruction **skip**, the most basic statement is the operation of an (unary) operator on a qubit ($\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i})$). The considered operators are NOT, $\mathrm{R}_Y^f(\mathrm{i})$, and $\mathrm{Ph}^f(\mathrm{i})$.

Statements also include sequences, (classical) conditionals, *quantum cases*, and *procedure calls* which take an integer input i and a sorted subset of qubits s (**call** proc[i](s);). A quantum case **qcase** q **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ provides a quantum control feature that will execute statements S_0 and S_1 in superposition. For example, the *CNOT* gate on qubits $\bar{\mathrm{q}}[i]$ and $\bar{\mathrm{q}}[j]$, for $i, j \in \mathbb{N}$, $i \neq j$, can be simulated by the following statement:

$$\mathrm{CNOT}(\bar{\mathrm{q}}[i], \bar{\mathrm{q}}[j]) \triangleq \textbf{qcase } \bar{\mathrm{q}}[i] \textbf{ of } \{0 \to \textbf{skip}, 1 \to \bar{\mathrm{q}}[j] \mathrel{*}= \mathrm{NOT};\}.$$

Throughout the paper, we restrict our study to *well-formed* programs, that is, programs P = D :: S satisfying the following properties: $var(\mathrm{S}) \subseteq \{\bar{\mathrm{q}}\}$; $\forall \mathrm{proc} \in \mathrm{P}$, $var(\mathrm{S}^{\mathrm{proc}}) \subseteq \{\mathrm{x}, \bar{\mathrm{p}}\}$; procedure names declared in D pairwise are distinct; for each procedure call, the procedure name is declared in D.

2.2 Semantics

Let $\mathcal{H}_{2^n}$ be the *Hilbert space* $\mathbb{C}^{2^n}$ of n qubits. We use Dirac notation to denote a quantum state $|\psi\rangle \in \mathcal{H}_{2^n}$. Each $|\psi\rangle \in \mathcal{H}_{2^n}$ can be written as a superposition of bitstrings of size n: $|\psi\rangle = \sum_{w \in \{0,1\}^n} \alpha_w |w\rangle$, with $\alpha_w \in \mathbb{C}$ and $\sum_w |\alpha_w|^2 = 1$. The *length* $\ell(|\psi\rangle)$ of the state $|\psi\rangle$ is n. Given two matrices M, N, we denote by $M^\dagger$ the transpose conjugate of M and by $M \otimes N$ the tensor product of M by N. $\langle\psi|$ is equal to $|\psi\rangle^\dagger$ and $|\psi\rangle\langle\phi|$ and $\langle\psi|\phi\rangle$ are respectively the outer and inner products of $|\psi\rangle$ with $|\phi\rangle$. Let

I_n be the identity matrix in $\mathbb{C}^{n\times n}$. Given $m \leq n$ and $i \in \{0,1\}$, define $|i\rangle\!\rangle_m \triangleq I_{2^{m-1}} \otimes |i\rangle \otimes I_{2^{n-m}}$ and $\langle\!\langle i|_m \triangleq (|i\rangle\!\rangle_m)^\dagger$.

A function $[\![\mathrm{U}^f]\!] \in \mathbb{Z} \to \tilde{\mathbb{C}}^{2\times 2}$ is associated to each operator U^f as follows:

$$[\![\mathrm{NOT}]\!](n) \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad [\![\mathrm{R}^f_Y]\!](n) \triangleq \begin{pmatrix} \cos(f(n)) & -\sin(f(n)) \\ \sin(f(n)) & \cos(f(n)) \end{pmatrix}, \quad [\![\mathrm{Ph}^f]\!](n) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{if(n)} \end{pmatrix},$$

where $\tilde{\mathbb{C}}$ is the set of complex numbers whose real and imaginary parts are both in $\tilde{\mathbb{R}}$. This restriction corresponds to the set of transitions amplitudes allowed by efficient quantum Turing machines [Adleman et al. 1997; Bernstein and Vazirani 1997]. This property is ensured by the description of function f, which links the integer value given as input to the gate with the actual rotation angles. One can easily check that each matrix $M \triangleq [\![\mathrm{U}^f]\!](n) \in \tilde{\mathbb{C}}^{2\times 2}$ is unitary, i.e., it satisfies $M^\dagger M = M M^\dagger = I_2$.

The choice of NOT, $\mathrm{R}^f_Y(\mathrm{i})$, and $\mathrm{Ph}^f(\mathrm{i})$ as basic operators is justified by the fact that, for the constant and polynomial-time approximable function $g \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$ s.t. $g(x) = \pi/4$, these operators simulate the following set of one-qubit unitary gates

$$[\![\mathrm{NOT}]\!](n) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad [\![\mathrm{R}^g_Y]\!](n) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \quad [\![\mathrm{Ph}^g]\!](n) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix},$$

which, combined with a construction for controlled gates, is universal for quantum computing [Boykin et al. 1999]. For instance, the Hadamard gate can be easily derived as

$$H \triangleq \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = [\![\mathrm{NOT}]\!](n) \cdot [\![\mathrm{R}^g_Y]\!](n).$$

Let $\mathbb{B}$ be the set of Boolean values $b \in \{\mathbf{false}, \mathbf{true}\}$. For a given set X, let $\mathcal{L}(X)$ be the set of lists of elements in X. Let $l = [x_1, \ldots, x_m]$, with $x_1, \ldots, x_m \in X$, denote a list of m elements in $\mathcal{L}(X)$ and $[\,]$ be the empty list (when $m = 0$). For $l, l' \in \mathcal{L}(X)$, $l@l'$ denotes the concatenation of l and l'. $hd(l)$ and $tl(l)$ represent the tail and the head of l, respectively. Lists of integers will be used to represent Sorted Sets. They contain pointers to qubits (i.e., indexes) in the global memory.

We interpret each basic data type τ as follows: $[\![\mathrm{Integers}]\!] \triangleq \mathbb{Z}$, $[\![\mathrm{Booleans}]\!] \triangleq \mathbb{B}$, $[\![\mathrm{SortedSets}]\!] \triangleq \mathcal{L}(\mathbb{N})$, $[\![\mathrm{Qubits}]\!] \triangleq \mathbb{N}$, and $[\![\mathrm{Operators}]\!] \triangleq \tilde{\mathbb{C}}^{2\times 2}$. Each basic operation $\mathrm{op} \in \{+, -, >, \geq, =, \wedge, \vee, \neg\}$ of arity n, with $1 \leq n \leq 2$, has a type signature $\tau_1 \times \ldots \times \tau_n \longrightarrow \tau$ fixed by the program syntax. For example, the operation + has signature Integers $\times$ Integers $\longrightarrow$ Integers. A total function $[\![\mathrm{op}]\!] \in [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \longrightarrow [\![\tau]\!]$ is associated to each op.

For each basic type τ, the reduction $\Downarrow_{[\![\tau]\!]}$ is a map in $\tau \times \mathcal{L}(\mathbb{N}) \longrightarrow [\![\tau]\!]$. Intuitively, it maps an expression of type τ and a list of pointers in $\mathcal{L}(\mathbb{N})$ to the value of the expression in $[\![\tau]\!]$. These reductions are defined in Figure 2, where e and d denote either an integer expression i or a boolean expression b.

Note that, in rule ($\mathrm{Rm}_{\notin}$), if we try to delete an undefined index then we return the empty list, and, in rule ($\mathrm{Qu}_{\notin}$), if we try to access an undefined qubit index then we return the value 0 (defined indexes will always be positive).

We can define some syntactic sugar for useful instructions, such as the Hadamard gate H and the swap gate. For instance, we can simulate the application of H on q by the following statement $\mathrm{q} \mathrel{*}= \mathrm{R}^g_Y(0);\ \mathrm{q} \mathrel{*}= \mathrm{NOT}$, with the function g defined by $g \in \mathbb{Z} \longrightarrow [0, 2\pi) \cap \tilde{\mathbb{R}}$ s.t. $g(x) = \pi/4$. By abuse of notation, we will sometimes use q *= H; to denote this statement. Using CNOT, we can also define the SWAP operation swapping the states between two qubits $\bar{\mathrm{q}}[i]$ and $\bar{\mathrm{q}}[j]$, with $i, j \in \mathbb{N},\ i \neq j$:

$$\mathrm{SWAP}(\bar{\mathrm{q}}[i], \bar{\mathrm{q}}[j]) \triangleq \mathrm{CNOT}(\bar{\mathrm{q}}[i], \bar{\mathrm{q}}[j])\ \mathrm{CNOT}(\bar{\mathrm{q}}[j], \bar{\mathrm{q}}[i])\ \mathrm{CNOT}(\bar{\mathrm{q}}[i], \bar{\mathrm{q}}[j]).$$

$$\frac{(\mathrm{e}, l) \Downarrow_{[\![\tau_1]\!]} m \qquad (\mathrm{d}, l) \Downarrow_{[\![\tau_2]\!]} n}{(\mathrm{e}\ \mathrm{op}\ \mathrm{d}, l) \Downarrow_{[\![\mathrm{op}]\!]([\![\tau_1]\!],[\![\tau_2]\!])} [\![\mathrm{op}]\!](m, n)} \text{(Op)} \qquad \frac{(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n}{(\mathrm{U}^f(\mathrm{i}), l) \Downarrow_{\mathbb{C}^{2\times 2}} [\![\mathrm{U}^f]\!](n)} \text{(Unit)}$$

$$\frac{}{(n, l) \Downarrow_{\mathbb{Z}} n} \text{(Cst)} \qquad \frac{(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, l) \Downarrow_{\mathbb{Z}} k \in [1, m]}{(\mathrm{s} \ominus [\mathrm{i}], l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m]} (\mathrm{Rm}_{\in})$$

$$\frac{(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_n]}{(|\mathrm{s}|, l) \Downarrow_{\mathbb{Z}} n} \text{(Size)} \qquad \frac{(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, l) \Downarrow_{\mathbb{Z}} k \notin [1, m]}{(\mathrm{s} \ominus [\mathrm{i}], l) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} (\mathrm{Rm}_{\notin})$$

$$\frac{}{(\mathrm{nil}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} \text{(Nil)} \qquad \frac{(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, l) \Downarrow_{\mathbb{Z}} k \in [1, m]}{(\mathrm{s}[\mathrm{i}], l) \Downarrow_{\mathbb{N}} x_k} (\mathrm{Qu}_{\in})$$

$$\frac{}{(\bar{\mathrm{q}}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l} \text{(Var)} \qquad \frac{(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, l) \Downarrow_{\mathbb{Z}} k \notin [1, m]}{(\mathrm{s}[\mathrm{i}], l) \Downarrow_{\mathbb{N}} 0} (\mathrm{Qu}_{\notin})$$

Fig. 2. Semantics of expressions

Let $\top$ and $\bot$ be two special symbols for termination and error, respectively, and let $\diamond$ stand for a symbol in $\{\top, \bot\}$. The set of *configurations* of dimension 2^n, denoted Conf_n, is defined by

$$\mathrm{Conf}_n \triangleq (\mathrm{Statements} \cup \{\top, \bot\}) \times \mathcal{H}_{2^n} \times \mathcal{P}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}),$$

with $\mathcal{P}(\mathbb{N})$ being the powerset over $\mathbb{N}$. A configuration $c = (\mathrm{S}, |\psi\rangle, A, l) \in \mathrm{Conf}_n$ contains a statement S to be executed (provided that $\mathrm{S} \notin \{\top, \bot\}$), a quantum state $|\psi\rangle$ of length n, a set A containing the indexes of qubits that are allowed to be accessed by statement S, and a list l of qubit pointers.

The program big-step semantics $\longrightarrow$, described in Figure 3, is defined as a relation in $\bigcup_{n\in\mathbb{N}} \mathrm{Conf}_n \times \mathrm{Conf}_n$. In the rules of Figure 3, $\longrightarrow$ is annotated by an integer, called *level*. For example, the level of the conclusion in the $(\mathrm{Call}_{[\,]})$ rule is 1. The level is used to count the total number of procedure calls that are not in superposition (i.e., in distinct branches of a quantum case).

We now give a brief intuition on the rules of Figure 3. Rules $(\mathrm{Asg}_\bot)$ and $(\mathrm{Asg}_\top)$ evaluate the application of a unitary operator, corresponding to $\mathrm{U}^f(\mathrm{j})$, to a qubit s[i]. For that purpose, they evaluate the index n of s[i] in the global memory. Rule $(\mathrm{Asg}_\bot)$ deals with the error case, where the corresponding qubit is not allowed to be accessed. Rule $(\mathrm{Asg}_\top)$ deals with the success case: the new quantum state is obtained by applying the result of tensoring the evaluation of $\mathrm{U}^f(\mathrm{j})$ to the right index. Rules $(\mathrm{Seq}_\diamond)$ and $(\mathrm{Seq}_\bot)$ evaluate the sequence of statements, depending on whether an error occurs or not. The (If) rule deals with classical conditionals in a standard way. The three rules $(\mathrm{Case}_\top)$, $(\mathrm{Case}_\bot)$, and $(\mathrm{Case}_\notin)$ evaluate the qubit index n of the control qubit s[i]. Then they check whether this index belongs to the set of accessible qubits (is n in A?). If so, intuitively, the two statements S_0 and S_1 are evaluated in superposition, on the projected state $\langle 0|_n|\psi\rangle$ and $\langle 1|_n|\psi\rangle$, respectively. During these evaluations, the index n cannot be accessed anymore. The rule $(\mathrm{Call}_{[\,]})$ treats the base case of a procedure call when the sorted set parameter is empty. In the non-empty case, rule $(\mathrm{Call}_\diamond)$ evaluates the sorted set parameter s to l' and the integer parameter x to n. It returns the result of evaluating the procedure statement $\mathrm{S}^{\mathrm{proc}}\{n/\mathrm{x}\}$, where n has been substituted to x, with regards to the updated qubit pointers list l'.

For a given program $\mathrm{P} = \mathrm{D} :: \mathrm{S}$ and a given quantum state $|\psi\rangle \in \mathcal{H}_{2^n}$, the *initial configuration* for input $|\psi\rangle$ is $c_{init}(|\psi\rangle) \triangleq (\mathrm{S}, |\psi\rangle, \{1, \ldots, n\}, [1, \ldots, n]) \in \mathrm{Conf}_n$. A program is *error-free* if there

$$\frac{}{(\textbf{skip}, |\psi\rangle, A, l) \xrightarrow{0} (\top, |\psi\rangle, A, l)} \text{(Skip)}$$

$$\frac{(\text{s[i]}, l) \Downarrow_{\mathbb{N}} n \notin A}{(\text{s[i]} \mathrel{*}= \text{U}^f(\text{j}), |\psi\rangle, A, l) \xrightarrow{0} (\bot, |\psi\rangle, A, l)} \text{(Asg}_\bot)$$

$$\frac{(\text{s[i]}, l) \Downarrow_{\mathbb{N}} n \in A \qquad (\text{U}^f(\text{j}), l) \Downarrow_{\mathbb{C}^{2\times 2}} M}{(\text{s[i]} \mathrel{*}= \text{U}^f(\text{j}), |\psi\rangle, A, l) \xrightarrow{0} (\top, I_{2^{n-1}} \otimes M \otimes I_{2^{l(|\psi\rangle)-n}} |\psi\rangle, A, l)} \text{(Asg}_\top)$$

$$\frac{(\text{S}_1, |\psi\rangle, A, l) \xrightarrow{m_1} (\top, |\psi'\rangle, A, l) \qquad (\text{S}_2, |\psi'\rangle, A, l) \xrightarrow{m_2} (\diamond, |\psi''\rangle, A, l)}{(\text{S}_1\ \text{S}_2, |\psi\rangle, A, l) \xrightarrow{m_1+m_2} (\diamond, |\psi''\rangle, A, l)} \text{(Seq}_\diamond)$$

$$\frac{(\text{S}_1, |\psi\rangle, A, l) \xrightarrow{m} (\bot, |\psi\rangle, A, l)}{(\text{S}_1\ \text{S}_2, |\psi\rangle, A, l) \xrightarrow{m} (\bot, |\psi\rangle, A, l)} \text{(Seq}_\bot)$$

$$\frac{(\text{b}, l) \Downarrow_{\mathbb{B}} b \in \mathbb{B} \qquad (\text{S}_b, |\psi\rangle, A, l) \xrightarrow{m_b} (\diamond, |\psi'\rangle, A, l)}{(\textbf{if}\ \text{b}\ \textbf{then}\ \text{S}_{\textbf{true}}\ \textbf{else}\ \text{S}_{\textbf{false}}, |\psi\rangle, A, l) \xrightarrow{m_b} (\diamond, |\psi'\rangle, A, l)} \text{(If)}$$

$$\frac{(\text{s[i]}, l) \Downarrow_{\mathbb{N}} n \in A \qquad \forall k \in \{0,1\},\ (\text{S}_k, |\psi\rangle, A\backslash\{n\}, l) \xrightarrow{m_k} (\top, |\psi_k\rangle, A\backslash\{n\}, l)}{(\textbf{qcase}\ \text{s[i]}\ \textbf{of}\ \{0 \to \text{S}_0, 1 \to \text{S}_1\}, |\psi\rangle, A, l) \xrightarrow{\max_{k\in\{0,1\}} m_k} (\top, \sum_{k\in\{0,1\}} |k\rangle_n \langle k|_n |\psi_k\rangle, A, l)} \text{(Case}_\top)$$

$$\frac{(\text{s[i]}, l) \Downarrow_{\mathbb{N}} n \in A \qquad \forall k \in \{0,1\},\ (\text{S}_k, |\psi\rangle, A\backslash\{n\}, l) \xrightarrow{m_k} (\diamond_k, |\psi_k\rangle, A\backslash\{n\}, l) \qquad \bot \in \{\diamond_0, \diamond_1\}}{(\textbf{qcase}\ \text{s[i]}\ \textbf{of}\ \{0 \to \text{S}_0, 1 \to \text{S}_1\}, |\psi\rangle, A, l) \xrightarrow{\max_{k\in\{0,1\}} m_k} (\bot, |\psi\rangle, A, l)} \text{(Case}_\bot)$$

$$\frac{(\text{s[i]}, l) \Downarrow_{\mathbb{N}} n \notin A}{(\textbf{qcase}\ \text{s[i]}\ \textbf{of}\ \{0 \to \text{S}_0, 1 \to \text{S}_1\}, |\psi\rangle, A, l) \xrightarrow{0} (\bot, |\psi\rangle, A, l)} \text{(Case}_\notin)$$

$$\frac{(\text{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l' \neq [\,] \qquad (\text{i}, l) \Downarrow_{\mathbb{Z}} n \qquad (\text{S}^{\text{proc}}\{n/\text{x}\}, |\psi\rangle, A, l') \xrightarrow{m} (\diamond, |\psi'\rangle, A, l')}{(\textbf{call}\ \text{proc[i](s)}, |\psi\rangle, A, l) \xrightarrow{m+1} (\diamond, |\psi'\rangle, A, l)} \text{(Call}_\diamond)$$

$$\frac{(\text{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]}{(\textbf{call}\ \text{proc[i](s)}, |\psi\rangle, A, l) \xrightarrow{1} (\top, |\psi\rangle, A, l)} \text{(Call}_{[\,]})$$

Fig. 3. Semantics of statements.

is no initial configuration $c_{init}(|\psi\rangle)$ such that $c_{init}(|\psi\rangle) \longrightarrow (\bot, |\psi'\rangle, A, l)$. We write $[\![\text{P}]\!](|\psi\rangle) = |\psi'\rangle$ whenever $c_{init}(|\psi\rangle) \xrightarrow{m} (\top, |\psi'\rangle, A, l)$ holds for some m. $(\top, |\psi'\rangle, A, l)$ is called a *terminal configuration*. Let $\mathcal{H} \triangleq \bigcup_n \mathcal{H}_{2^n}$, a program *terminates* if $[\![\text{P}]\!]$ is a total function in $\mathcal{H} \to \mathcal{H}$. Note that if a program terminates then it is obviously error-free but the converse property does not hold. Every program P can be efficiently transformed into an error-free program $\text{P}_{\neg\bot}$ such that $\forall|\psi\rangle$, if

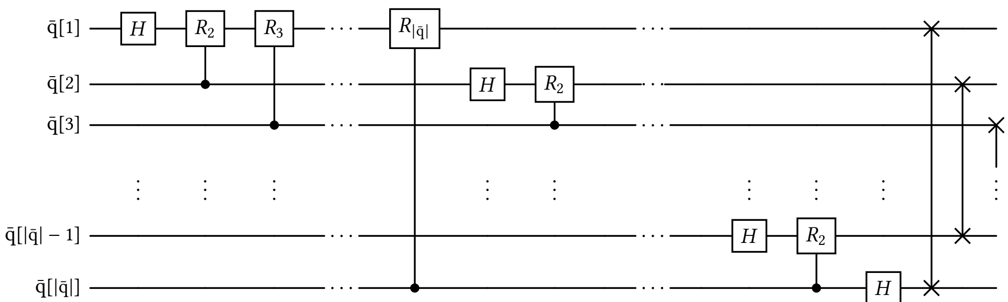


Fig. 4. Circuit for the quantum Fourier transform.

```
decl qft(p̄){
   p̄[1] *= H;
   call rot[2](p̄);
   call qft(p̄ ⊖ [1]); },

decl rot[x](p̄){
 if |p̄| > 1 then
  qcase p̄[2] of {
   0 → skip;
   1 → p̄[1] *= Ph^{λx.π/2^{x−1}}(x);
  }
  call rot[x + 1](p̄ ⊖ [2]);
 else skip; },

decl inv(p̄){
   if |p̄| > 1 then
      SWAP(p̄[1], p̄[|p̄|]);
      call inv(p̄ ⊖ [1, |p̄|]);
   else skip; } ::

call qft(q̄); call inv(q̄);
```

Fig. 5. PFOQ program QFT for the quantum Fourier transform.

$[\![\mathrm{P}]\!](|\psi\rangle\!\rangle)$ is defined then $[\![\mathrm{P}]\!](|\psi\rangle\!\rangle) = [\![\mathrm{P}_{\neg\perp}]\!](|\psi\rangle\!\rangle)$. For example, an assignment $\mathrm{s[i]} \mathrel{*}= \mathrm{U}^f(\mathrm{j})$; can be transformed into the conditional statement **if** $((0 < \mathrm{i}) \wedge (\mathrm{i} \leq |\mathrm{s}|))$ **then** $\mathrm{s[i]} \mathrel{*}= \mathrm{U}^f(\mathrm{j})$; **else skip**.

Example 1 (Quantum Fourier Transform). *A notable example of a quantum algorithm is the Quantum Fourier Transform (QFT), used as a subroutine in Shor's algorithm [Shor 1994], and whose quantum circuit is provided in Figure 4, with $R_n \triangleq [\![Ph^{\lambda x.\pi/2^{x-1}}]\!](n)$, for $n \geq 2$. After iterating the application of Hadamard and controlled R_n gates, the circuit performs a permutation of qubits using swap gates.*

Note that $\lambda x.\pi/2^{x-1}$ is a total function in $\mathbb{Z} \longrightarrow [0, 2\pi) \cap \tilde{\mathbb{R}}$ that is polynomial-time approximable. The circuit of Figure 4 can be simulated for any number of qubits $|\mathrm{q}|$ by the FOQ *program* QFT *provided in Figure 5.*

Example 2 (Teleportation). *If Alice and Bob share n EPR states, they can teleport any n-length state between their labs [Bennett et al. 1993]. The delayed measurement version of the quantum teleportation circuit can be encoded as a* FOQ *program.*

We consider an input state of length n, and we extend it with 2n qubits in state $|0\rangle^{\otimes 2n}$. We can then create a FOQ *program that transforms each pair $|00\rangle$ into an EPR state and then performs the teleportation for each qubit. An example of such a code is the program of Figure 6. As we will see later, the circuit of Figure 7 is obtained by compiling the* FOQ *program of Figure 6.*

In Figure 7, the ground symbol represents which qubits would be measured in the end, and output wires x_i denote the teleported qubits. For $1 \leq i \leq n$, we have that the i-th qubit of the input appears in position $3n - 2(i - 1)$ of the final state.

```
decl createBell(p̄){                          decl teleport(p̄){
   if |p̄| ≥ 3 :                                  if |p̄| ≥ 3 :
      p̄[|p̄| − 1] ∗= H;                              CNOT(p̄[1], p̄[|p̄| − 1])
      CNOT(p̄[|p̄| − 1], p̄[|p̄|])                     p̄[1] ∗= H;
      call createBell(p̄ ⊖ [1, |p̄| − 1, |p̄|]);       CNOT(p̄[|p̄| − 1], p̄[|p̄|])
   else skip; },                                    qcase p̄[1] of {0 → skip;, 1 → p̄[|p̄|] ∗= Ph^{λx.π}(0); }
                                                    call teleport(p̄ ⊖ [1, |p̄| − 1, |p̄|]);
                                                 else skip; } ::
call createBell(q̄); call teleport(q̄);
```

Fig. 6. PFOQ program for quantum teleportation.

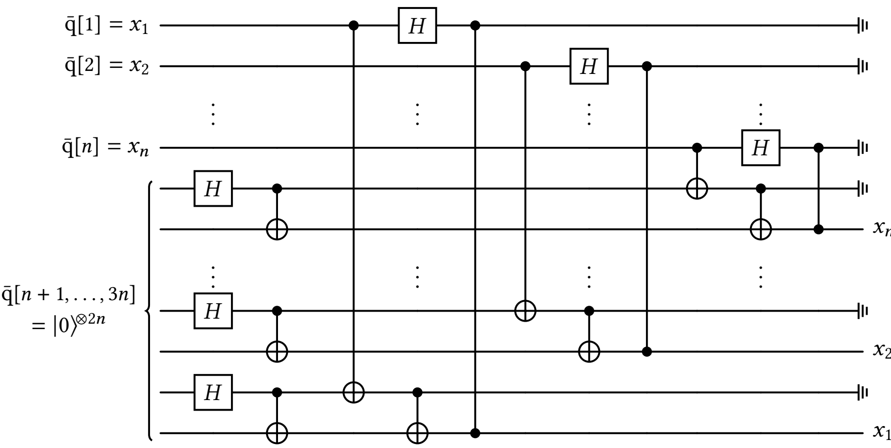


Fig. 7. Circuit for quantum teleportation.

2.3 Derivation Tree and Level

Given a configuration c with regards to a fixed program P, $\pi_{\mathrm{P}} \rhd c \longrightarrow c'$ denotes the *derivation tree* of P that describes the evolution of c into a terminal configuration c'. This tree is only defined if the program terminates, and is obtained by applying the rules of Figures 2 and 3 on configuration c with respect to P. We write π instead of $\pi_{\mathrm{P}} \rhd c \longrightarrow c'$ when P, c and c' are clear from the context. We will write $\pi \trianglelefteq \pi'$ to denote that π is a subtree of π'.

In the case of a terminating computation $\pi \rhd c \longrightarrow c'$, we can define a level $m \in \mathbb{N}$ such that $c \xrightarrow{m} c'$ holds. In this case, the level of π is defined as $\mathrm{lv}_\pi \triangleq m$. Given a terminating FOQ program P, $\mathrm{level}_{\mathrm{P}}$ is a total function in $\mathbb{N} \longrightarrow \mathbb{N}$ defined as $\mathrm{level}_{\mathrm{P}}(n) \triangleq \max_{|\psi\rangle \in \mathcal{H}_{2^n}} \mathrm{lv}_{\pi_{\mathrm{P}} \rhd c_{init}(|\psi\rangle)}$.

Intuitively, $\mathrm{level}_{\mathrm{P}}(n)$ corresponds to the maximal number of non-superposed procedure calls in any program execution on an input of length n.

Example 3. *Consider the program* QFT *of Example 1. Assume temporarily that* QFT *terminates (this will be shown in Example 4). For all* $n \in \mathbb{N}$, $\mathrm{level}_{\mathrm{QFT}}(n) = \frac{(n+1)(n+2)}{2} + \lfloor \frac{n}{2} \rfloor + 1$. *Indeed, on sorted sets of size n, procedure* qft *is called recursively* $n+1$ *times and makes* $n+1$ *calls to procedure* rot

on sorted sets of size n, $n-1, \dots,$ and 1. On sorted sets of size n, rot *performs n recursive calls. Hence the total number of calls to* rot *is equal to $\sum_{i=1}^{n} i$. Finally, on a sorted set of size n, procedure* inv *does* $\lfloor \frac{n}{2} \rfloor + 1$ *recursive call.*

A program P is reversible if it terminates and there exists a program P^{-1} such that $[\![\mathrm{P}^{-1}]\!] \circ [\![\mathrm{P}]\!] = Id$.

Theorem 1. *All terminating* foq *programs are reversible.*

Proof. We show this by induction on the structure of the program. Let $\mathrm{P} = \mathrm{D} :: \mathrm{S}$ be a program that terminates. Define $\mathrm{P}^{-1} \triangleq (\mathrm{D} :: \mathrm{S})^{-1}$ inductively as

$$\begin{array}{rcl}
(\mathrm{D} :: \mathrm{S})^{-1} & \triangleq & \mathrm{D}^{-1} :: \mathrm{S}^{-1} \\
(\mathbf{decl}\ \mathrm{proc}[\mathrm{x}](\bar{\mathrm{p}})\{\mathrm{S}\}, \mathrm{D})^{-1} & \triangleq & \mathbf{decl}\ \mathrm{proc}[\mathrm{x}](\bar{\mathrm{p}})\{\mathrm{S}^{-1}\}, \mathrm{D}^{-1} \\
\varepsilon^{-1} & \triangleq & \varepsilon \\
\mathbf{skip;}^{-1} & \triangleq & \mathbf{skip;} \\
(\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});)^{-1} & \triangleq & \mathrm{q} \mathrel{*}= (\mathrm{U}^f(\mathrm{i}))^\dagger; \\
(\mathrm{S}_1\ \mathrm{S}_2)^{-1} & \triangleq & \mathrm{S}_2^{-1}\ \mathrm{S}_1^{-1} \\
(\mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \mathrm{S}_{\mathbf{true}}\ \mathbf{else}\ \mathrm{S}_{\mathbf{false}})^{-1} & \triangleq & \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \mathrm{S}_{\mathbf{true}}^{-1}\ \mathbf{else}\ \mathrm{S}_{\mathbf{false}}^{-1} \\
(\mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\})^{-1} & \triangleq & \mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0 \to \mathrm{S}_0^{-1}, 1 \to \mathrm{S}_1^{-1}\} \\
(\mathbf{call}\ \mathrm{proc}[\mathrm{i}](\mathrm{s});)^{-1} & \triangleq & \mathbf{call}\ \mathrm{proc}[\mathrm{i}](\mathrm{s});,
\end{array}$$

with $\mathrm{NOT}^\dagger \triangleq \mathrm{NOT}$, $\mathrm{R}_Y^f(\mathrm{i})^\dagger \triangleq \mathrm{R}_Y^{-f}(\mathrm{i})$, and $\mathrm{Ph}^f(\mathrm{i})^\dagger \triangleq \mathrm{Ph}^{-f}(\mathrm{i})$. P^{-1} terminates and is such that for all $|\psi\rangle$, $[\![\mathrm{P}^{-1}]\!]([\![\mathrm{P}]\!](|\psi\rangle)) = |\psi\rangle$. □

3 Polynomial time soundness

In this section, we restrict the set of foq programs to a strict subset, named pfoq, that is sound for the quantum complexity class fbqp. For this, we define two criteria: a criterion ensuring that a program terminates and a criterion preventing a terminating program from having an exponential runtime.

3.1 Polynomial-Time First Order Quantum Programs

Given two statements S, S', we write $\mathrm{S} \in \mathrm{S}'$ to mean that S is a substatement of S' and $\mathrm{proc} \in \mathrm{S}$ holds if there are i and s such that **call** proc[i](s) $\in$ S. Given a program $\mathrm{P} = \mathrm{D} :: \mathrm{S}$, we define the relation $>_{\mathrm{P}} \subseteq \mathrm{Procedures} \times \mathrm{Procedures}$ by $\mathrm{proc}_1 >_{\mathrm{P}} \mathrm{proc}_2$ if $\mathrm{proc}_2 \in \mathrm{S}^{\mathrm{proc}_1}$, for any two procedures $\mathrm{proc}_1, \mathrm{proc}_2 \in \mathrm{S}$. Put simply, the relation $\mathrm{proc}_1 >_{\mathrm{P}} \mathrm{proc}_2$ means that for procedures proc_1 and proc_2 defined in program P, proc_1 performs a call to proc_2.

Using this relation, we can define a partial order on procedures for a given program, by defining $\succeq_{\mathrm{P}}$ as the transitive and reflexive closure of $>_{\mathrm{P}}$. Given a procedure proc_1 in a program P, the procedures proc_i that satisfy $\mathrm{proc}_1 \succeq_{\mathrm{P}} \mathrm{proc}_i$ are precisely those that can called from an original call to proc_1.

Furthermore, we define the equivalence relation $\sim_{\mathrm{P}}$ as $\mathrm{proc}_1 \sim_{\mathrm{P}} \mathrm{proc}_2$ if $\mathrm{proc}_1 \succeq_{\mathrm{P}} \mathrm{proc}_2$ and $\mathrm{proc}_2 \succeq_{\mathrm{P}} \mathrm{proc}_1$ both hold. This gives us a generalized notion of mutual recursion between procedures. We also define the strict order $\succ_{\mathrm{P}}$ as $\mathrm{proc}_1 \succ_{\mathrm{P}} \mathrm{proc}_2$ if $\mathrm{proc}_1 \succeq_{\mathrm{P}} \mathrm{proc}_2$ and $\mathrm{proc}_1 \nsim_{\mathrm{P}} \mathrm{proc}_2$ both hold. Example 4 showcases these relations and the partial order $\succeq_{\mathrm{P}}$ for the QFT program (Example 1).

Definition 1. *Let* wf *be the set of* foq *programs* P *that are error-free and satisfy the well-foundedness constraint:*

$$\forall \mathrm{proc} \in \mathrm{P},\ \forall \mathbf{call}\ \mathrm{proc}'[\mathrm{i}](\mathrm{s}) \in \mathrm{S}^{\mathrm{proc}},\ \mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}' \Rightarrow \exists k > 0, \exists \mathrm{i}_1, \dots, \mathrm{i}_k,\ s = \bar{\mathrm{p}} \ominus [\mathrm{i}_1, \dots, \mathrm{i}_k].$$

Lemma 1. *If* P $\in$ wf, *then* P *terminates.*

Proof. For a given program P, we define a lexicographical partial order between configurations whose statements are procedure calls.

$$(\textbf{call}\ \mathrm{proc}[\mathrm{i}](\mathrm{s}), |\psi\rangle, A, l) \gg_{\mathrm{P}} (\textbf{call}\ \mathrm{proc}'[\mathrm{i}'](\mathrm{s}'), |\psi'\rangle, A', l')$$

if $(\mathrm{proc} \succ_{\mathrm{P}} \mathrm{proc}') \vee (\mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}' \wedge n_{\mathrm{s}} > n_{\mathrm{s}'})$, provided that $(|\mathrm{s}|, l) \Downarrow_{\mathbb{N}} n_{\mathrm{s}}$ and $(|\mathrm{s}'|, l') \Downarrow_{\mathbb{N}} n_{\mathrm{s}'}$.

Given $\pi \rhd (\textbf{call}\ \mathrm{proc}[\mathrm{i}](\mathrm{s}), |\psi\rangle, A, l)$ and $\pi' \rhd (\textbf{call}\ \mathrm{proc}'[\mathrm{i}'](\mathrm{s}'), |\psi'\rangle, A', l')$, we show that if $\pi' \trianglelefteq \pi$ then it holds that

$$(\textbf{call}\ \mathrm{proc}[\mathrm{i}](\mathrm{s}), |\psi\rangle, A, l) \gg_{\mathrm{P}} (\textbf{call}\ \mathrm{proc}'[\mathrm{i}'](\mathrm{s}'), |\psi'\rangle, A', l').$$

Consider a program $\mathrm{P} \in \textsc{wf}$. Assume that $\pi' \trianglelefteq \pi$, then it holds that $\mathrm{proc} \succeq_{\mathrm{P}} \mathrm{proc}'$. Hence, either $\mathrm{proc} \succ_{\mathrm{P}} \mathrm{proc}'$ or $\mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}'$. In this latter case, by transitivity of $\sim_{\mathrm{P}}$, $\mathrm{s}' = \bar{\mathrm{p}} \ominus [\mathrm{i}_1, \ldots, \mathrm{i}_k]$, for some $k > 0$. It implies that the evaluation of $|\mathrm{s}'|$ is strictly smaller than the evaluation of $|\mathrm{s}|$, by transitivity. We conclude by observing that $\gg_{\mathrm{P}}$ is a well-founded order. □

Example 4. *Consider the program* QFT *of Example 1. The statements of the procedure declarations define the following relation:* $\mathrm{qft} >_{\mathrm{QFT}} \mathrm{qft}$, $\mathrm{qft} >_{\mathrm{QFT}} \mathrm{rot}$, $\mathrm{rot} >_{\mathrm{QFT}} \mathrm{rot}$, *and* $\mathrm{inv} >_{\mathrm{QFT}} \mathrm{inv}$. *Consequently,* $\mathrm{qft} \sim_{\mathrm{QFT}} \mathrm{qft}$, $\mathrm{rot} \sim_{\mathrm{QFT}} \mathrm{rot}$, $\mathrm{inv} \sim_{\mathrm{QFT}} \mathrm{inv}$, *and* $\mathrm{qft} \succ_{\mathrm{QFT}} \mathrm{rot}$ *hold. For each call to an equivalent procedure, we check that the argument decreases:* $\bar{\mathrm{p}} \ominus [1]$ *in* qft, $\bar{\mathrm{p}} \ominus [2]$ *in* rot, *and* $\bar{\mathrm{p}} \ominus [1, |\bar{\mathrm{p}}|]$ *in* inv. *Consequently,* $\mathrm{QFT} \in \textsc{wf}$. *We deduce from Lemma 1 that* QFT *terminates.*

We now add a further restriction on mutually recursive procedure calls for guaranteeing polynomial time using a notion of *width*.

Definition 2. *Given a program* P *and a procedure* $\mathrm{proc} \in \mathrm{P}$, *the width of* proc in P, *noted* $\mathrm{width}_{\mathrm{P}}(\mathrm{proc})$, *is defined as* $\mathrm{width}_{\mathrm{P}}(\mathrm{proc}) \triangleq w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}^{\mathrm{proc}})$, *where* $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S})$ *is the* width of the procedure proc in P relative to statement S, *defined inductively as:*

$$\begin{aligned}
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{skip};) &\triangleq 0,\\
w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});) &\triangleq 0,\\
w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1\ \mathrm{S}_2) &\triangleq w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) + w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_2),\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_{\textbf{true}}\ \textbf{else}\ \mathrm{S}_{\textbf{false}}) &\triangleq \max(w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_{\textbf{true}}), w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_{\textbf{false}})),\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}) &\triangleq \max(w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0), w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1)),\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{call}\ \mathrm{proc}'[\mathrm{i}](\mathrm{s});) &\triangleq \begin{cases} 1 & \textit{if}\ \mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}',\\ 0 & \textit{otherwise}. \end{cases}
\end{aligned}$$

Importantly, this notion of width should *not* be mistaken with that of *circuit width*, which refers to the number of qubits in a circuit. In this case, it is analogous to a "branching factor" which accounts for how many recursive calls in sequence can be performed by a single procedure call.

Definition 3 (PFOQ). pfoq *is the set of programs defined as:*

$$\textsc{pfoq} \triangleq \{\mathrm{P} \mid \mathrm{P} \in \textsc{wf} \wedge \forall \mathrm{proc} \in \mathrm{P}, \mathrm{width}_{\mathrm{P}}(\mathrm{proc}) \leq 1\}.$$

If $\mathrm{P} \in \textsc{pfoq}$, then we call P a pfoq program.

Example 5. *In the program of Example 1,* $\mathrm{width}_{\mathrm{QFT}}(\mathrm{qft}) = \mathrm{width}_{\mathrm{QFT}}(\mathrm{rot}) = \mathrm{width}_{\mathrm{QFT}}(\mathrm{inv}) = 1$, *since* $\mathrm{qft} \succ_{\mathrm{QFT}} \mathrm{rot}$ *holds. Since* $\mathrm{QFT} \in \textsc{wf}$, *by Example 4, we conclude that* QFT *is a* pfoq *program.*

We now show that the level of a pfoq program is bounded by a polynomial in the length of its input. We will start by defining the notion of *rank of a procedure* and *rank of a program*.

Definition 4 (Rank of a procedure). *Given a* FOQ *program* P, *the rank of a procedure in* P *is defined as:*

$$rk(\text{proc}) \triangleq 0 \qquad \textit{if} \nexists\, \text{proc}',\ \text{proc} \succ_{\text{P}} \text{proc}',$$
$$rk(\text{proc}) \triangleq \max\{rk(\text{proc}') + 1 \mid \text{proc} \succ_{\text{P}} \text{proc}'\} \qquad \textit{otherwise.}$$

Definition 5 (Rank of a program). *The rank of program* P *is defined by*

$$rk(\text{P}) \triangleq \max_{\text{proc} \in \text{P}} rk(\text{proc}).$$

Lemma 2. *For any* PFOQ *program* P, *we have that* $\text{level}_{\text{P}}(n) = O(n^{rk(\text{P})+1})$.

Proof. Consider a PFOQ program P = D :: S. We show the result by induction on the rank of procedure calls in S. Take **call** proc[i](s) ∈ S. If $rk(\text{proc}) = 0$ and the procedure is not recursive then there is only 1 call to a procedure. If the procedure is recursive, it can be called at most once in each branch of a quantum case statement. Hence there can be at most $n + 1$ such calls in the full quantum case branch of the derivation tree and it holds that $\text{level}_{\text{D::}\mathbf{call}\ \text{proc}[\text{i}](\text{s})}(n) = O(n)$.

Induction hypothesis: assume that any procedure proc′ such that $rk(\text{proc}') \leq k$ satisfies

$$\text{level}_{\text{D::}\mathbf{call}\ \text{proc}'[\text{i}](\text{s})}(n) = O(n^{k+1}).$$

Consider a procedure proc such that $rk(\text{proc}) = k + 1$. If the procedure is not recursive then it can call a constant number (bounded by the size of the program) of procedures of strictly smaller rank. By induction hypothesis,

$$\text{level}_{\text{D::}\mathbf{call}\ \text{proc}[\text{i}](\text{s});}(n) = \sum_{\text{proc}' <_{\text{P}} \text{proc}} O(n^{rk(\text{proc}')+1}) = O(n^{rk(\text{proc})}).$$

If the procedure is recursive, it can be called at most once in each branch of a quantum case statement. Hence there can be at most $n + 1$ such calls in the full quantum case branch of the derivation tree. Moreover, each of these calls can perform a constant number of calls to procedures of strictly smaller rank. Consequently,

$$\text{level}_{\text{D::}\mathbf{call}\ \text{proc}[\text{i}](\text{s});}(n) = O(n) + \sum_{i=0}^{n} \sum_{\text{proc}' <_{\text{P}} \text{proc}} O(n^{rk(\text{proc}')+1}) = O(n^{rk(\text{proc})+1}).$$

We observe that for a program P = D :: $\text{S}_1 \ \ldots \ \text{S}_k$, it holds that

$$\text{level}_{\text{P}}(n) = O(\sum_{i=1}^{k} \text{level}_{\text{D::S}_i}(n)) = O(n^{rk(\text{P})+1}).$$

This concludes the proof. □

Moreover, checking whether a program is in PFOQ is tractable.

Theorem 2. *For each* FOQ *program* P, *it can be decided in time* $O(|\text{P}|^2)$ *whether* $\text{P}_{\neg\perp} \in$ PFOQ.

Proof. Any FOQ program P can be transformed in time $O(|\text{P}|)$ into an equivalent error-free program $\text{P}_{\neg\perp}$ of size $O(|\text{P}|)$. Then, checking that $\text{P}_{\neg\perp} \in$ PFOQ amounts to checking that $\text{P}_{\neg\perp} \in$ WF and that $\forall \text{proc} \in \text{P}_{\neg\perp}$, $\text{width}_{\text{P}_{\neg\perp}}(\text{proc}) \leq 1$.

First, we can compute the recursion relations $\sim_{\text{P}}$, $\succeq_{\text{P}}$, and $\succ_{\text{P}}$, in time $O(|\text{P}|^2)$. Then, checking that all recursive calls reduce the size of the input qubit list and computing the width of each procedure can be done at the same time, by traversing the structure of each procedure statement in linear time. Consequently, both conditions can be checked in time $O(|\text{P}|^2)$. □

3.2 Quantum Turing Machines and the Class of Polytime Quantum Functions

Following [Bernstein and Vazirani 1997], a k-tape *Quantum Turing Machine* (QTM), with $k \geq 1$, is defined by a triplet (Σ, Q, δ) where

- Σ is a finite alphabet including a blank symbol #,
- Q is a finite set of states with an initial state s_0 and a final state $s_\top \neq s_0$,
- δ is the quantum transition function in $Q \times \Sigma^k \to (Q \times \Sigma^k \times \{L, N, R\}^k \to \tilde{\mathbb{C}})$, with $\{L, N, R\}$ being the set of possible movements of a head on a tape.

Each tape of the QTM is two-way infinite and contains cells indexed by $\mathbb{Z}$. A QTM successfully terminates if it reaches a superposition of only the final state $s_\top$. A QTM is said to be *well-formed* if the transition function δ preserves the norm of the superposition (or, equivalently, if the time evolution of the machine is unitary). The starting position of the tape heads is the *start cell*, the cell indexed by 0. If the machine terminates with all of its tape heads back on the start cells, it is usually called *stationary*. We will use *stationary* in the case where the machine terminates with its input tape head in the first cell, and all other tape heads in the rightmost non-blank cell. This notion differs from the definition of stationary QTM in [Yamakami 2020] but it can be shown to be equivalent. We will further refer to a QTM as being *in normal form* if the only transitions from the final state $s_\top$ are towards the initial state s_0. These will be important conditions for the composition and branching constructions of QTMs. If a QTM is well-formed, stationary, and in normal form, we will call it *conservative* [Yamakami 2020].

A configuration γ of a k-tape QTM is a tuple $(s, \overline{w}, \overline{n})$, where s is a state in Q, $\overline{w}$ is a k-tuple of words in Σ^*, and $\overline{n}$ is a k-tuple of indexes (head positions) in $\mathbb{Z}$. An initial (resp. final) configuration γ_{init} (resp. γ_{fin}) is a configuration of the shape $(s_0, \overline{w}, \overline{0})$ (resp. $(s_\top, \overline{w}, \overline{0})$). We use $\gamma(w)$ to denote a configuration γ where the word w is written on the input/output tape. Following [Bernstein and Vazirani 1997], we write $\mathcal{S}$ to represent the inner-product space of finite complex linear combinations of configurations of the QTM M with the Euclidean norm. A QTM M defines a linear time operator $U_M : \mathcal{S} \to \mathcal{S}$ that outputs a superposition of configurations $\sum_i \alpha_i |\gamma_i\rangle$ obtained by applying a single-step transition of M to a configuration $|\gamma\rangle$ (i.e., $U_M|\gamma\rangle = \sum_i \alpha_i |\gamma_i\rangle$). Let U_M^t, for $t \geq 1$, be the t-steps transition obtained from U_M as follows: $U_M^1 \triangleq U_M$ and $U_M^{t+1} \triangleq U_M \circ U_M^t$. Given a quantum state $|\psi\rangle = \sum_{w \in \{0,1\}^n} \alpha_w |w\rangle$ and a configuration γ, let $\gamma(|\psi\rangle) \in \mathcal{S}$ be the quantum configuration defined by $\gamma(|\psi\rangle) \triangleq \sum_{w \in \{0,1\}^n} \alpha_w |\gamma(w)\rangle$.

A quantum function $f : \mathcal{H} \to \mathcal{H}$ is computed by the QTM M in time t for an input $|\psi\rangle \in \mathcal{H}$ if $U_M^t(\gamma_{init}(|\psi\rangle)) = \gamma_{fin}(f(|\psi\rangle))$. Given $T : \mathbb{N} \to \mathbb{N}$ and a quantum function f, we say that the QTM M computes f in time T if for inputs of length n, M computes f in time $T(n)$.

Definition 6. *Given two functions $f : \{0,1\}^\star \to \{0,1\}^\star$, $F : \mathcal{H} \to \mathcal{H}$, and a value $p \in [0,1]$, we say that f is computed by F with probability p if $\forall x \in \{0,1\}^\star,\ |\langle f(x)|F(|x\rangle)|^2 \geq p$.*

Definition 7 (fbqp, [Bernstein and Vazirani 1997]). *A function $f \in \{0,1\}^\star \to \{0,1\}^\star$ is in* fbqp*iff there exist a QTM M and a polynomial $P \in \mathbb{N}[X]$ such that M computes f in time P with probability $\frac{2}{3}$.*

The class fbqp is the functional extension of the complexity class bqp, where we only consider functions $f \in \{0,1\}^\star \to \{0,1\}$.

A function $f \in \{0,1\}^\star \to \{0,1\}^\star$ has a *polynomial bound* $P \in \mathbb{N}[X]$ if $\forall n \in \mathbb{N}, \forall x \in \{0,1\}^n, \exists k \leq P(n),\ f(x) \in \{0,1\}^k$. Functions in fbqp have a polynomial bound as the size of their output is smaller than the polynomial time bound.

3.3 Proof of Soundness

We show that QTMs can simulate the function computed by any terminating FOQ program. The time complexity of this simulation depends on the length of the input quantum state and on the level of the considered program.

LEMMA 3. *For any terminating* FOQ *program* P*, there exists a conservative QTM M that computes* $[\![\mathrm{P}]\!]$ *in time* $O(n + n \times \mathrm{level}_{\mathrm{P}}(n))$.

PROOF. Consider a terminating FOQ program P = D :: S. We build a 4-tape QTM M computing $[\![\mathrm{P}]\!]$ inductively on the statement S. Fix $\Sigma \triangleq \{0, 1, \#, \|, \&\}$, where # is the blank symbol and where $\|$ and & are special separation symbols for encoding stacks. The input tape t_{in} of M contains a word in $\{0, 1, \#\}^n$ encoding the quantum state. The 3 working tapes are t_{call}, t_l, and $t_{\mathbb{K}}$ for storing the integer values of a procedure call, the list of qubit pointers, and intermediate classical computations, respectively, as words in Σ^*. The configurations of M will be in $Q \times (\Sigma^*)^4 \times \mathbb{Z}^4$, for some finite set of states Q. In particular, the initial configuration is $(s_0, w, \varepsilon, \varepsilon, \varepsilon, 0, 0, 0, 0)$, with $w \in \{0, 1\}^n$ encoding a quantum state of length $n \in \mathbb{N}$; the tapes t_{call}, t_l, and $t_{\mathbb{K}}$ are initially empty (ε). The tape heads all start on the first cells indexed by 0. For $m \in \mathbb{Z}$, let $t(m)$ denote the symbol at position m on tape t. Given a word $w \in \Sigma^*$ and a tape t, tw denotes that the content of t ends with the word w.

By abuse of notation, let $[\![\mathrm{e}]\!]$ denote the result of evaluating the expression e with respect to the machine current configuration. Also, we will assume that deterministic computations, such as taking tape $t||[\![\mathrm{i}]\!]$ and appending $f([\![\mathrm{i}]\!])$, for any function f, are done by a reversible Turing machine [Bennett 1973], as reversible TMs are well-formed QTMs [Bernstein and Vazirani 1997, Theorem 4.2].

We now describe a QTM M simulating P inductively on the statement S.

- The **skip** statement is trivial.
- If S = q $*=$ U^f(j);, M appends $[\![\mathrm{q}]\!]$ to $t_{\mathbb{K}}$. As the program terminates, $t_{in}([\![\mathrm{q}]\!]) \neq \#$ and the transition function is set to:

$$\forall a \in \{0, 1\},\ \delta(s_{\mathrm{S}}, t_{in}([\![\mathrm{q}]\!]), s_{next(\mathrm{S})}, a, N) \triangleq \langle t_{in}([\![\mathrm{q}]\!])|[\![\mathrm{U}^f]\!]([\![\mathrm{j}]\!])|a\rangle$$

  where s_{S} is the state before executing the assignment when the head of the input tape has been moved to position $[\![\mathrm{q}]\!]$, and $s_{next(\mathrm{S})}$ is the state just after executing the assignment. Finally, the machine erases $\|[\![\mathrm{q}]\!]$ at the end of $t_{\mathbb{K}}$, leaves its head in the last non-blank cell of $t_{\mathbb{K}}$, and moves the head in t_{in} back to the initial cell. Program P has $\mathrm{level}_{\mathrm{P}}(n) = 0$ and the simulating machine runs in time $O(n)$.

For the remaining statements, assume by induction hypothesis the existence of two conservative QTMs M_1 and M_2 that compute functions $[\![\mathrm{P}_1]\!]$ and $[\![\mathrm{P}_2]\!]$, respectively, with $\mathrm{P}_1 \triangleq \mathrm{D} :: \mathrm{S}_1$ and $\mathrm{P}_2 \triangleq \mathrm{D} :: \mathrm{S}_2$. By induction hypothesis M_1 and M_2 run in time $O(n + n \times \mathrm{level}_{\mathrm{P}_i}(n))$. States of M_i will be denoted by s^i, for $i \in \{1, 2\}$. By using internal clocks, we can assume without loss of generality that machines M_i halt in exactly the same time for any quantum input of length n.

- Consider the case $\mathrm{S} = \mathrm{S}_1\,\mathrm{S}_2$. Machine M is defined as in [Bernstein and Vazirani 1997, Dovetailing Lemma], with the initial state $s_0 \triangleq s_0^1$, its final state $s_\top \triangleq s_\top^2$, and the two machines are composed by setting $s_\top^1 = s_0^2$. The machine M is stationary, well-formed and it is well-behaved since the running time of M_2 only depends on n and the output of M_1 contains a superposition of equally sized quantum states. M computes $[\![\mathrm{P}]\!]$ in time $O(n + n \times \mathrm{level}_{\mathrm{P}_1}(n))) + O(n + n \times \mathrm{level}_{\mathrm{P}_2}(n)) = O(n + n \times \mathrm{level}_{\mathrm{P}}(n))$.
- For the conditional S = **if** b **then** S_1 **else** S_2, we build a machine M that concatenates $\|[\![\mathrm{b}]\!]$ on the working tape $t_{\mathbb{K}}$ and runs M_1 or M_2 deterministically depending on the value of $[\![\mathrm{b}]\!]$,

using the [Bernstein and Vazirani 1997, Branching Lemma]. Then we erase $[\![\mathrm{b}]\!]$ from the end of tape $t_{\mathbb{K}}$. M computes $[\![\mathrm{P}]\!]$ in time $\max_i(O(n + n \times \mathrm{level}_{\mathrm{P}_i}(n))) = O(n + n \times \mathrm{level}_{\mathrm{P}}(n))$.
- For the quantum case $\mathrm{S} =$ **qcase** q **of** $\{0 \to \mathrm{S}_1, 1 \to \mathrm{S}_2\}$, the machine appends $\|[\![\mathrm{q}]\!]$ on tape $t_{\mathbb{K}}$. It reads $t_{in}([\![\mathrm{q}]\!])$, sets it to #, and if it reads 0 runs M_1, if it reads 1, runs M_2. Finally, from state $s_{\top}^1$, it writes 0 in $t_{in}([\![\mathrm{q}]\!])$, moves the head to index 0, and transitions to $s_{\top}$; similarly, from state $s_{\top}^2$, the machines writes 1 in $t_{in}([\![\mathrm{q}]\!])$ before moving the head and transitioning to $s_{\top}$. We have that M computes $[\![\mathrm{P}]\!]$ in time $\max_i(O(n+n\times \mathrm{level}_{\mathrm{P}_i}(n))) = O(n+n\times \mathrm{level}_{\mathrm{P}}(n))$.
- For the procedure call $\mathrm{S} =$ **call** proc[i](s);, inductively define machine M as follows: update t_{call} by appending $[\![\mathrm{i}]\!]$, and update t_l by adding the qubit pointer indices excluded in $[\![\mathrm{s}]\!]$, separating them using the symbol &. We then run machine M_{proc} that computes the function $[\![\mathrm{P}_{\mathrm{proc}}]\!]$, for $\mathrm{P}_{\mathrm{proc}} \triangleq \mathrm{D} :: \mathrm{S}^{\mathrm{proc}}\{[\![\mathrm{i}]\!]/\mathrm{x}, \mathrm{s}/\bar{\mathrm{q}}\}$, in time $O(m+m\times \mathrm{level}_{\mathrm{P}_{\mathrm{proc}}}(m))$, with $m \triangleq [\![|\mathrm{s}|]\!] \leq n$, afterwards erasing $[\![\mathrm{i}]\!]$ and the new indices of t_{in}. As $\mathrm{level}_{\mathrm{P}_{\mathrm{proc}}}(n) = O(\mathrm{level}_{\mathrm{P}}(n))$ and the complexity of M is $O(n + n \times \mathrm{level}_{\mathrm{P}}(n))$. □

Now we show that any PFOQ program computes a FBQPfunction.

THEOREM 3 (SOUNDNESS). *Given a* PFOQ *program* P*, a function* $f : \{0,1\}^\star \to \{0,1\}^\star$*, and a value* $p \in (\frac{1}{2}, 1]$*. If* f *is computed by* $[\![\mathrm{P}]\!]$ *with probability* p *then* $f \in$ FBQP*.*

PROOF. Given a PFOQ program P, by Lemma 3, there exists a conservative QTM M_{P} that computes $[\![\mathrm{P}]\!]$ in time $O(n + n \times \mathrm{level}_{\mathrm{P}}(n))$. By Lemma 2, there exists a polynomial $Q \in \mathbb{N}[X]$ such that $\forall n,\ \mathrm{level}_{\mathrm{P}}(n) \leq Q(n)$. Consequently, the result holds by Definition 7. □

4 Polytime completeness

In this section we show that any function in FBQP can be faithfully approximated by a PFOQ program. Toward this end, we show that [Yamakami 2020]'s FBQP-complete function algebra can be exactly simulated in PFOQ.

A characterization of FBQP was provided in [Yamakami 2020] using a function algebra, named $\widehat{\square_1^{\mathrm{QP}}}$. Given a quantum state $|\psi\rangle$ and a word $w \in \{0,1\}^n$, with $n \leq \ell(|\psi\rangle)$, $|\psi\rangle$ can be written as $|\psi\rangle = \sum_i \alpha_i |w_i z_i\rangle$, with $w_i \in \{0,1\}^n$ and $z_i \in \{0,1\}^{\ell(|\psi\rangle)-n}$. We write $\langle w|\psi\rangle$ as an abuse of notation for the quantum state defined by $\langle w|\psi\rangle \triangleq \sum_i \alpha_i \langle w|w_i\rangle\, |z_i\rangle$, which consists of projecting the state $|\psi\rangle$ onto the subspace where the first n qubits are in state $|w\rangle$. For instance,

$$\langle 00|\left(\frac{1}{\sqrt{2}}|000\rangle + \frac{1}{2}|001\rangle + \frac{1}{2}|010\rangle\right) = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{2}|1\rangle.$$

DEFINITION 8 (YAMAKAMI'S FUNCTION ALGEBRA). $\widehat{\square_1^{\mathrm{QP}}}$ *is the smallest class of functions including the following basic functions:*

- $I(|\psi\rangle) \triangleq |\psi\rangle$,
- $Ph_\theta(|\psi\rangle) \triangleq |0\rangle\langle 0|\psi\rangle + e^{\mathrm{i}\theta}|1\rangle\langle 1|\psi\rangle$, *with* $\theta \in [0, 2\pi) \cap \tilde{\mathbb{C}}$,
- $Rot_\theta(|\psi\rangle) \triangleq \cos\theta|\psi\rangle + \sin\theta(|1\rangle\langle 0|\psi\rangle - |0\rangle\langle 1|\psi\rangle)$, *with* $\theta \in [0, 2\pi) \cap \tilde{\mathbb{C}}$,
- $NOT(|\psi\rangle) \triangleq |0\rangle\langle 1|\psi\rangle + |1\rangle\langle 0|\psi\rangle$
- $SWAP(|\psi\rangle) \triangleq \begin{cases} |\psi\rangle & \text{if } \ell(|\psi\rangle) \leq 1 \\ \sum_{a,b\in\{0,1\}} |ba\rangle\langle ab|\psi\rangle & \text{otherwise,} \end{cases}$

and closed under schemes Comp*,* Branch*, and* $kQRec_t$*, for* $k, t \in \mathbb{N}$*,*

- $Comp[F,G](|\psi\rangle) \triangleq F(G(|\psi\rangle))$
- $Branch[F,G](|\psi\rangle) \triangleq \begin{cases} |\psi\rangle & \text{if } \ell(|\psi\rangle) \leq 1 \\ |0\rangle\otimes F(\langle 0|\psi\rangle) + |1\rangle\otimes G(\langle 1|\psi\rangle) & \text{otherwise,} \end{cases}$

- $kQRec_t[F,G,H](|\psi\rangle) \triangleq \begin{cases} F(|\psi\rangle) & \text{if } \ell(|\psi\rangle) \leq t \\ G\left(\sum_{w\in\{0,1\}^k} |w\rangle \otimes F_w(\langle w|H(|\psi\rangle))\right) & \text{otherwise,} \end{cases}$
  where each $F_w \in \{kQRec_t[F,G,H], I\}$.

To handle general FBQPfunctions, [Yamakami 2020] defines the extended encoding of an input $x \in \{0,1\}^\star$ as $\phi_P(|x\rangle) \triangleq |0^{\ell(|x\rangle)}1\rangle|0^{P(\ell(|x\rangle))}10^{11P(\ell(|x\rangle))+6}1\rangle|x\rangle$, for some polynomial $P \in \mathbb{N}[X]$ that is an upper bound on the output size of the desired FBQP function. ϕ_P simply consists in the quantum state $|x\rangle$ preceded by a polynomial number of ancilla qubits (the constants appearing in the expression are a result of the precise simulation of the QTM used in [Yamakami 2020]). These ancilla provide space for internal computations and account for the polynomial bound associated with polynomial time QTMs.

Theorem 4 ([Yamakami 2020]). *Given* $f : \{0,1\}^\star \to \{0,1\}^\star$ *with polynomial bound* $P \in \mathbb{N}[X]$, *the following statements are equivalent.*

1. *The function* f *is in* FBQP.
2. *There exists* $F \in \widehat{\square_1^{\mathrm{QP}}}$ *such that* $F \circ \phi_P$ *computes* f *with probability* $\frac{2}{3}$.

We now show that any function in $\widehat{\square_1^{\mathrm{QP}}}$ can be simulated by a PFOQ program.

Theorem 5. *Let* F *be a function in* $\widehat{\square_1^{\mathrm{QP}}}$. *Then there exists a* PFOQ *program* P *such that* $[\![\mathrm{P}]\!] = F$.

Proof. We prove this result by structural induction on a function in the $\widehat{\square_1^{\mathrm{QP}}}$ algebra. The basic function I can be simulated by $\mathrm{P}(\bar{\mathrm{q}}) = \varepsilon :: \mathbf{skip}$. $F \in \{Ph_\theta, ROT_\theta, NOT\}$ can be simulated using an assignment. In these cases $\mathrm{P}(\bar{\mathrm{q}}) = \varepsilon :: \bar{\mathrm{q}}[1] \mathrel{*}= \mathrm{U}^f(0)$; with f such that $[\![\mathrm{U}^f]\!](0) = F$. The basic initial function $SWAP$ can be simulated by the program $\mathrm{P}(\bar{\mathrm{q}}) = \varepsilon :: \mathrm{SWAP}(\bar{\mathrm{q}}[1], \bar{\mathrm{q}}[2])$, with the SWAP statement defined in Section 2.2.

We now simulate the $Comp$, $Branch$ and $kQRec_t$ schemes. For that purpose, assume the existence of PFOQ programs $\mathrm{P}_F(\bar{\mathrm{q}}) = \mathrm{D}_F :: \mathrm{S}_F$, $\mathrm{P}_G(\bar{\mathrm{q}}) = \mathrm{D}_G :: \mathrm{S}_G$, and $\mathrm{P}_H(\bar{\mathrm{q}}) = \mathrm{D}_H :: \mathrm{S}_H$ simulating the $\widehat{\square_1^{\mathrm{QP}}}$ functions F, G, and H, respectively. For simplicity, we assume that there are no name clashes between the procedures declared in D_F, D_G, and D_H.

$Comp[F,G]$ can be simulated by $\mathrm{P}(\bar{\mathrm{q}}) = \mathrm{D}_F, \mathrm{D}_G :: \mathrm{S}_G\ \mathrm{S}_F$. Moreover $\mathrm{P} \in$ PFOQ by construction.

$Branch[F,G]$ can be simulated by

$$\begin{aligned}\mathrm{P}(\bar{\mathrm{q}}) =& \mathrm{D}_F, \mathrm{D}_G, \mathbf{decl}\ \mathrm{proc}_F(\bar{\mathrm{q}})\{\mathrm{S}_F\},\ \mathbf{decl}\ \mathrm{proc}_G(\bar{\mathrm{q}})\{\mathrm{S}_G\} :: \\ & \mathbf{qcase}\ \bar{\mathrm{q}}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathrm{proc}_F(\bar{\mathrm{q}} \ominus [1]);, 1 \to \mathbf{call}\ \mathrm{proc}_G(\bar{\mathrm{q}} \ominus [1]);\}.\end{aligned}$$

Procedures proc_F and proc_G are not recursive and therefore $\mathrm{P} \in$ PFOQ.

We introduce the following syntactical sugar for an extended quantum case for $k \geq 2$, $\forall a \in \{0,1\}$, and $\forall w \in \{0,1\}^{k-1}$:

$$\begin{aligned}&\mathbf{qcase}\ \mathrm{s}[\mathrm{i}_1,\dots,\mathrm{i}_k] && \mathbf{of}\ \{aw \to \mathrm{S}_{aw}\} \\ &\triangleq \mathbf{qcase}\ \mathrm{s}[\mathrm{i}_1]\ \mathbf{of}\ \{ && 0 \to \mathbf{qcase}\ \mathrm{s}[\mathrm{i}_2\dots\mathrm{i}_k]\ \mathbf{of}\ \{w \to \mathrm{S}_{0w}\}, \\ &&& 1 \to \mathbf{qcase}\ \mathrm{s}[\mathrm{i}_2\dots\mathrm{i}_k]\ \mathbf{of}\ \{w \to \mathrm{S}_{1w}\}\}.\end{aligned}$$

$kQRec_t[F, G, H]$ can be simulated by

```
P(q̄) =D_F, D_G, D_H,
        decl proc_F(q̄){S_F}, decl proc_G(q̄){S_G}, decl proc_H(q̄){S_H},
        decl kQRec_t(p̄){
          if |p̄| > t then
            call proc_H(p̄);
            qcase p̄[1...k] of {w → S_w}
            call proc_G(p̄);
          else
            call proc_F(p̄); } ::
        call kQRec_t(q̄);
```

with $\mathrm{S}_w \triangleq \begin{cases} \textbf{call } \mathrm{kQRec}_t(\bar{\mathrm{p}} \ominus [1\ldots k]); & \text{if } F_w = kQRec_t, \\ \textbf{skip}; & \text{if } F_w = I. \end{cases}$

The recursive calls to procedure kQRec_t are of the shape **call** $\mathrm{kQRec}_t(\bar{\mathrm{p}} \ominus [1, \ldots, k])$. Moreover, $\mathrm{kQRec}_t \succ_{\mathrm{P}} \mathrm{proc}_F, \mathrm{proc}_G, \mathrm{proc}_H$, and $\mathrm{P}_F,\ \mathrm{P}_G,\ \mathrm{P}_H \in$ WF. Consequently, P $\in$ WF. Finally, the procedure kQRec_t is called recursively at most once per branch of a quantum case. Consequently, $width_{\mathrm{P}}(\mathrm{kQRec}_t) \leq 1$ and P is a PFOQ program. □

We are now ready to state the completeness result.

THEOREM 6. *For every function f in* FBQP *with polynomial bound $Q \in \mathbb{N}[X]$, there is a* PFOQ *program* P *such that $[\![\mathrm{P}]\!] \circ \phi_Q$ computes f with probability $\frac{2}{3}$.*

PROOF. Consider a function f in FBQP with polynomial bound Q and an input $|\psi\rangle \in \mathcal{H}$. By Theorem 4 there exists a function F in $\widehat{\square_1^{\mathrm{QP}}}$ such that $F \circ \phi_Q$ computes f with probability $\frac{2}{3}$. By Theorem 5, there exists a program P_F such that $[\![\mathrm{P}_F]\!] = F$. Finally, $[\![\mathrm{P}_F]\!] \circ \phi_Q$ computes f with probability $\frac{2}{3}$. □

5 Compilation to polynomial-size quantum circuits

In this section, we provide an algorithm that compiles a PFOQ program on a given input length $n \in \mathbb{N}$ into a quantum circuit of size polynomial in n.

5.1 Quantum Circuits

Quantum circuits [Deutsch 1989] are a well-known graphical computational model for describing quantum computations. Qubits are represented by wires. Each unitary transformation U acting on n qubits can be represented as a gate U with n inputs and n outputs. A circuit C is an element of a PROP category ([MacLane 1965], a symmetric strict monoidal category) whose morphisms are generated by gates in G and wires. Let $\mathbb{1}$ be the identity circuit (for any length) and $\circ$ and $\otimes$ be the composition and product, respectively. By abuse of notation, given k circuits $C^1, \ldots, C^k$, $\circ_{i=1}^k C^i$ will denote the circuit $\tilde{C}^1 \circ \cdots \circ \tilde{C}^k$, where each circuit $\tilde{C}^i$ is obtained by tensoring C^i appropriately with identities so that the output of C^i matches the input of C^{i+1}. By construction, a circuit is acyclic. Each circuit C_n can be indexed by its number $n \in \mathbb{N}$ of input wires (i.e., non ancilla qubits) and computes a function $[\![C_n]\!] \in \mathcal{H}_{2^n} \to \mathcal{H}_{2^n}$. To deal with functions in $\mathcal{H} \to \mathcal{H}$, we consider families of circuits $(C_n)_{n \in \mathbb{N}}$, that are sequences of circuits such that each C_n encodes computation on quantum states of length n. Hence each circuit has n input qubits plus some extra ancilla qubits.

These ancillas can be used to perform intermediate computations but also to represent functions whose output size is strictly greater than their input size. To avoid the consideration of families encoding undecidable properties, we put a uniformity restriction.

Definition 9. *A family of circuits $(C_n)_{n\in\mathbb{N}}$ is said to be* uniform *if there exists a polynomial time Turing machine that takes n as input and outputs a representation of C_n, for all $n \in \mathbb{N}$.*

In quantifying the complexity of a circuit, it is necessary to specify the considered elementary gates, and define the complexity of an operation as the number of elementary gates needed to perform it. In our setting, this will be of the same order as the number of elementary instructions corresponding to a unitary operation $\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i})$. Notice that, as exemplified in Figure 8, the controlled version of a unitary may comprise more than a single elementary gate, and in fact the chosen amplitude may have to be approximated up to some desired value; but these can all be considered to add a constant overhead on the number of instructions. The size $\#C$ of a circuit C is equal to the number of its gates and wires.

Definition 10. *A family of circuits $(C_n)_{n\in\mathbb{N}}$ is said to be* polynomial size *with $\alpha \in \mathbb{N} \to \mathbb{N}$ ancilla qubits if there exists a polynomial $P \in \mathbb{N}[X]$ such that, for each $n \in \mathbb{N}$, $\#C_n \leq P(n)$ and the number of ancilla qubits in C_n is exactly $\alpha(n)$.*

Let $\chi_m : \mathcal{H}_{2^n} \to \mathcal{H}_{2^{n+m}}$ be defined by $\chi_m(|\psi\rangle) \triangleq |\psi\rangle \otimes |0^m\rangle$, for a state $|\psi\rangle$ of size n. This functions pads the input with ancillas in state $|0\rangle$ to match the circuit dimension. Furthermore, we use $\| |\psi\rangle \|$ to denote the norm of state $|\psi\rangle$. Finally, let $|w|$, for $w \in \{0,1\}^\star$, be the size of the word w.

Theorem 7 (Adapted from [Yao 1993] and [Nielsen and Chuang 2011]). *A function $f : \{0,1\}^\star \to \{0,1\}^\star$ is in* FBQP *iff there exists a uniform polynomial size family of circuits $(C_n)_{n\in\mathbb{N}}$ with α ancilla qubits s.t. $\forall x \in \{0,1\}^\star$,*

$$\left\| \left\langle f(x) \middle| [\![C_{|x|}]\!] \circ \chi_{\alpha(|x|)}(|x\rangle) \right\rangle \right\| \geq \frac{2}{3}.$$

5.2 Compilation Algorithm

For each PFOQ program P, the existence of a polynomial size uniform family of circuits $(C_n)_{n\in\mathbb{N}}$ that computes $[\![\mathrm{P}]\!]$ is entailed by the combination of Lemma 2 and Theorem 7. However, compilation of a program into its equivalent circuit via its QTM representation is far from desirable in a practical scenario. In this section, we exhibit an algorithm that compiles any PFOQ program directly into a polynomial size circuit. The remainder of this section will be devoted to presenting an algorithm, named **compile**, which, for a given PFOQ program P and a given integer n, produces a circuit C_n such that $\forall |\psi\rangle \in \mathcal{H}_{2^n}$, $[\![\mathrm{P}]\!](|\psi\rangle) = \xi_n \circ [\![C_n]\!] \circ \chi_{\alpha(n)}(|\psi\rangle)$, where ξ_m removes the ancilla qubits from the output state by performing the projection $\xi_m(|\psi\rangle) \triangleq \sum_{w\in\{0,1\}^m} \langle w0^{n-m}|\psi\rangle$, where $n = \ell(|\psi\rangle)$.

The subroutine **compr** (Algorithm 1) generates the circuit inductively on the program statement. It takes as inputs: a program P, a list of qubit pointers l, and a *control structure* cs. A control structure cs is a partial function in $\mathbb{N} \to \{0,1\}$, mapping a qubit pointer to a control value (of a quantum case). Let $\cdot$ be the control structure of empty domain. For $n \in \mathbb{N}$ and $k \in \{0,1\}$, $cs[n := k]$ is the control structure obtained from cs by setting $cs(n) \triangleq k$. Given a partial function $f : A \to B$, let $dom(f)$ be the domain of f.

For a given $x \in \{0,1\}^\star$, we say that state $|x\rangle$ *satisfies* cs if, $\forall n \in dom(cs), cs(n) = k \Rightarrow |\langle k|_n |x\rangle|^2 = 1$. Put differently, $|x\rangle$ satisfies cs if it for every $n \in dom(cs)$, qubit n in $|x\rangle$ is in state $cs(n)$. In the context of Figure 8, this can be interpreted as $|x\rangle$ being a basis state where U will be applied since $|x\rangle$ satisfies U's control structure.

Two control structures cs and cs' are said to be *orthogonal* if there does not exist a state $|x\rangle$ that satisfies cs and cs'. This occurs when two control structures differ on a value in both their domains.

Algorithm 1 (**compr**)
Input: $(\mathrm{D} :: \mathrm{S}, l, cs) \in$ Programs $\times \mathcal{L}(\mathbb{N}) \times (\mathbb{N} \to \{0,1\})$

if S = **skip then**
  $C \leftarrow \mathbb{1}$ ▹ Identity circuit

else if S = s[i] *= U^f(j); **and** $(\mathrm{s[i]}, l) \Downarrow_{\mathbb{N}} n$ **and** $(\mathrm{U}^f(\mathrm{j}), l) \Downarrow_{\mathbb{C}^{2\times 2}} M$ **then**
  $C \leftarrow M(cs, [n])$ ▹ Controlled gate

else if S = $\mathrm{S}_1\ \mathrm{S}_2$ **then**
  $C \leftarrow$ **compr**$(\mathrm{D} :: \mathrm{S}_1, l, cs) \circ$ **compr**$(\mathrm{D} :: \mathrm{S}_2, l, cs)$ ▹ Composition

else if S = **if** b **then** $\mathrm{S}_{\mathbf{true}}$ **else** $\mathrm{S}_{\mathbf{false}}$ **and** $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} b$ **then**
  $C \leftarrow$ **compr**$(\mathrm{D} :: \mathrm{S}_b, l, cs)$ ▹ Conditional

else if S = **qcase** s[i] **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ **and** $(\mathrm{s[i]}, l) \Downarrow_{\mathbb{N}} n$ **then**
  $C \leftarrow$ **compr**$(\mathrm{D} :: \mathrm{S}_0, l, cs[n := 0]) \circ$ **compr**$(\mathrm{D} :: \mathrm{S}_1, l, cs[n := 1])$ ▹ Quantum case

else if S = **call** proc[i](s); **and** $(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]$ **then**
  $C \leftarrow \mathbb{1}$ ▹ Nil call

else if S = **call** proc[i](s) **and** $(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l' \neq [\,]$ **and** $(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n$ **then**
  if $\mathrm{width_P}(\mathrm{proc}) = 0$ **then**
    $C \leftarrow$ **compr**$(\mathrm{D} :: \mathrm{S}^{\mathrm{proc}}\{n/\mathrm{x}\}, l', cs)$ ▹ Non-recursive call
  else if $\mathrm{width_P}(\mathrm{proc}) = 1$ **then**
    $C \leftarrow$ **optimize**$(\mathrm{D}, [(cs, \mathrm{S}^{\mathrm{proc}}\{n/\mathrm{x}\})], \mathrm{proc}, l', \{\})$ ▹ Recursive call
  end if
end if
return C

Intuitively, two controlled gates with orthogonal control structures will act on disjoint sets of basis states. An important consequence of this is that the two controlled gates will commute.

Given a control structure cs and a statement S, a *controlled statement* is a pair $(cs, \mathrm{S}) \in \mathrm{Cst} \triangleq (\mathbb{N} \to \{0,1\}) \times$ Statements. Intuitively, a controlled statement (cs, S) denotes a statement controlled by the qubits whose indices are in $dom(cs)$. For a unitary gate $U \in \mathcal{H}_{2^n} \to \mathcal{H}_{2^n}$, a control structure cs, and a list of pointers $l = [x_1, \ldots, x_n] \in \mathcal{L}(\mathbb{N})$ such that $\{x_1, \ldots, x_n\} \cap dom(cs) = \emptyset$, $U(cs, l)$ denotes the circuit applying gate U on qubits $\bar{\mathrm{q}}[x_1], \ldots, \bar{\mathrm{q}}[x_n]$, whenever $\forall m \in dom(cs)$, $\bar{\mathrm{q}}[m]$ is in state $|cs(m)\rangle$. Given a set $\mathcal{S}$, let $card(\mathcal{S})$ be the cardinality (i.e., number of elements) of $\mathcal{S}$. As demonstrated in [Nielsen and Chuang 2011], this circuit can be built with $O(card(dom(cs)))$ elementary gates and ancillas, and a single controlled-U gate.

Example 6. *As an illustrative example, consider a binary gate U and a control structure cs such that $dom(cs) = \{1,2,3\}$, $cs(1) = cs(2) = 1$, and $cs(3) = 0$. Also consider a list $l = [4,5] \in \mathcal{L}(\mathbb{N})$. The circuit $U(cs, l)$ is provided in Figure 8.*

Similarly, we can define a generalized Toffoli gate as a circuit of the shape $NOT(cs, n)$. Since $card(dom(cs))$ will not scale with the size of the input, such a circuit has a constant cost in gates and ancillas and can thus be considered as an elementary gate. We will also be interested in rearranging wires under a given control structure. For two lists of qubit pointers $l_1 = [x_1, \ldots, x_n]$, $l_2 =$

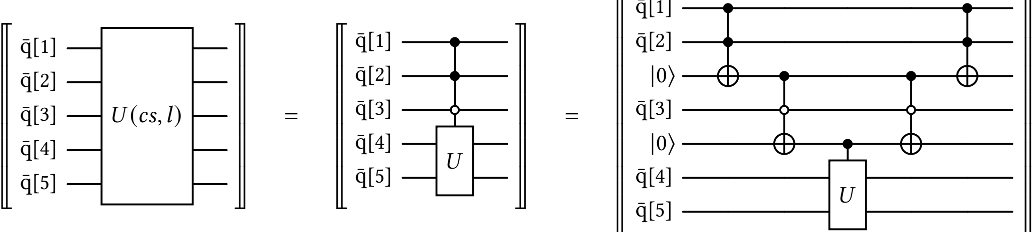


Fig. 8. Example of circuit $U(cs, l)$

$[x'_1, \ldots, x'_n] \in \mathcal{L}(\mathbb{N})$, define $SWAP(cs, l_1, l_2)$ as the circuit that swaps the wires in l_1 with wires in l_2, controlled on cs. This circuit needs in the worst case one ancilla and $O(n)$ controlled $SWAP$ gates (also known as Fredkin gates).

Let $\mathcal{D} \triangleq \text{Dict}(\text{Procedures} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{N} \times \mathcal{L}(\mathbb{N}))$ denote the set of dictionaries mapping keys of the shape (proc, i, j) to pairs of the shape (a, l), where i is the value of a classical parameter, j is the size of a sorted set, and a is a qubit index. We will denote the empty dictionary by $\{\}$. Let also $a \leftarrow$ **new** $ancilla()$ be an instruction that sets a to a fresh qubit index.

The subroutine **optimize** (Algorithm 2) treats the complex cases where circuit optimization (merging) is needed, that is, for recursive procedure calls. It takes as input a sequence of procedure declarations D, a list of controlled statements l_{Cst}, a procedure name proc, a list of qubit pointers l, and a dictionary Anc. The subroutine iterates on list l_{Cst} of controlled statements, indicating the statements left to be treated together with their control qubits. When recursive procedure calls appear in distinct branches of a quantum case, the algorithm merges these calls together. For that purpose, it uses new ancilla qubits as control qubits. Given procedure calls of shape **call** proc[i](s), with respect to a given list $l \in \mathcal{L}(\mathbb{N})$, we evaluate $(\text{i}, l) \Downarrow_{\mathbb{Z}} i$, $(\text{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l'$, and $(|\text{s}|, l) \Downarrow_{\mathbb{N}} j$. If the key (proc, i, j) already exists in the dictionary Anc, the associated ancilla is re-used, otherwise, $\text{Anc}[\text{proc}, i, j]$ is set to (a, l'). We include a few comments in the code which help identify the different compilation cases, where we use the shorthand phrase “S is recursive” to say that statement S contains a recursive procedure call.

We describe the technique of Algorithm 2 as *anchoring and merging*, highlighting the two types of operations performed to reduce the circuit complexity:

(1) *Anchoring*: each function call on a specific input size (and specific integer input) is assigned a controlled ancilla that will carry all its subsequent calls and operations;
(2) *Merging*: whenever a procedure call is repeated (same procedure, same input size, same integer input value), its realization is done simply by routing the control into the correct ancilla.

Some extra ancillas e are also created for swapping wires and are not explicitly indexed since they are not revisited by the subroutine, and are just considered unique. Ancillas a and e are indexed and treated as input qubits, therefore they can be part of the domain of control structures.

The **compile** algorithm employs two subroutines, named **compr** and **optimize**, and is defined as $\textbf{compile}(\text{P}, n) \triangleq \textbf{compr}(\text{P}, [1, \ldots, n], \cdot)$.

Theorem 8. *For each* $\text{P} \in$ PFOQ, *we have that, for all* $n \in \mathbb{N}$ *and* $|\psi\rangle \in \mathcal{H}_{2^n}$, *the following hold:*

1. $[\![\text{P}]\!](|\psi\rangle) = \xi_n \circ [\![\textbf{compile}(\text{P}, n)]\!] \circ \chi_{\alpha(n)}(|\psi\rangle)$ *and*
2. $\#\textbf{compile}(\text{P}, n) = O(n^{2rk(\text{P})+1})$.

Algorithm 2 (optimize) Build circuit for recursive procedure proc
Inputs: $(\mathrm{D}, l_{\mathrm{Cst}}, \mathrm{proc}, l, \mathrm{Anc}) \in \mathrm{Decl} \times \mathcal{L}(\mathrm{Cst}) \times \mathrm{Procedures} \times \mathcal{L}(\mathbb{N}) \times \mathcal{D}$

$C_{\mathrm{L}} \leftarrow \mathbb{1}; C_{\mathrm{R}} \leftarrow \mathbb{1}; \mathrm{P} \leftarrow \mathrm{D} :: \mathbf{skip};$
while $l_{\mathrm{Cst}} \neq [\,]$ **do**
  $(cs, \mathrm{S}) \leftarrow hd(l_{\mathrm{Cst}});\ l_{\mathrm{Cst}} \leftarrow tl(l_{\mathrm{Cst}})$

  if $\mathrm{S} = \mathrm{S}_1\,\mathrm{S}_2$ **then** ▹ Composition (Figure 10a)
    if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 1$ **then** ▹ branch S_1 is recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_1)];\ C_{\mathrm{R}} \leftarrow \mathbf{compr}(\mathrm{D} :: \mathrm{S}_2, l, cs) \circ C_{\mathrm{R}}$
    else ▹ branch S_2 is recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_2)];\ C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \mathbf{compr}(\mathrm{D} :: \mathrm{S}_1, l, cs)$
    end if
  end if

  if $\mathrm{S} =$ **if** b **then** $\mathrm{S}_{\mathbf{true}}$ **else** $\mathrm{S}_{\mathbf{false}}$ **and** $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} b$ **then** ▹ Conditional (Figure 10b)
    if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_b) = 1$ **then** ▹ if branch S_b is recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_b)]$
    else ▹ if branch S_b is not recursive
      $C_L \leftarrow C_L \circ \mathbf{compr}(\mathrm{D} :: \mathrm{S}_b, l, cs)$
    end if
  end if

  if $\mathrm{S} =$ **qcase** $s[\mathrm{i}]$ **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ **and** $(s[\mathrm{i}], l) \Downarrow_{\mathbb{N}} n$ **then** ▹ Quantum case (Figure 10c)
    if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 1$ **and** $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 1$ **then** ▹ both branches are recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0), (cs[n := 1], \mathrm{S}_1)]$
    else if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 0$ **then** ▹ only branch S_0 is recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0)];$
      $C_{\mathrm{R}} \leftarrow \mathbf{compr}(\mathrm{D} :: \mathrm{S}_1, l, cs[n := 1]) \circ C_{\mathrm{R}}$
    else if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 0$ **then** ▹ only branch S_1 is recursive
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 1], \mathrm{S}_1)];$
      $C_{\mathrm{R}} \leftarrow \mathbf{compr}(\mathrm{D} :: \mathrm{S}_0, l, cs[n := 0]) \circ C_{\mathrm{R}}$
    end if
  end if

  if $\mathrm{S} =$ **call** $\mathrm{proc}'[\mathrm{i}](\mathrm{s})$ **and** $(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l' \neq [\,]$ **and** $(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n$ **then**
    if $(\mathrm{proc}', n, |l'|) \in \mathrm{Anc}$ **then** ▹ Merging (Figure 10d)
      Let $(a, l'') = \mathrm{Anc}[\mathrm{proc}', n, |l'|]$ **in**
      $e \leftarrow$ **new** $ancilla();$
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ NOT(cs, e) \circ NOT(\cdot[e = 1], a) \circ SWAP(\cdot[e = 1], l', l'');$
      $C_{\mathrm{R}} \leftarrow SWAP(\cdot[e = 1], l'', l') \circ NOT(\cdot[e = 1], a) \circ NOT(cs, e) \circ C_{\mathrm{R}}$
    else ▹ Anchoring (Figure 10e)
      $a \leftarrow$ **new** $ancilla()$
      $\mathrm{Anc}[\mathrm{proc}', n, |l'|] \leftarrow (a, l');$
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ NOT(cs, a);\ C_{\mathrm{R}} \leftarrow NOT(cs, a) \circ C_{\mathrm{R}};$
      $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(\cdot[a = 1], \mathrm{S}^{\mathrm{proc}'}\{n/\mathrm{i}\})]$
    end if
  end if
end while
return $C_L \circ C_R$

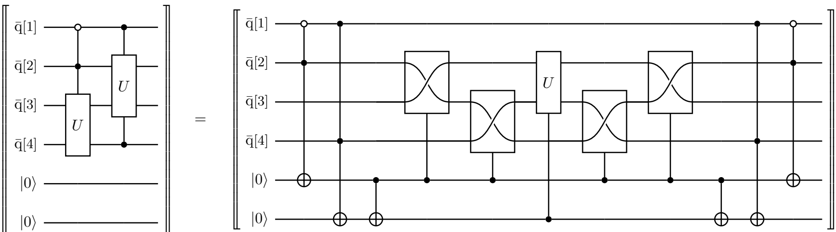


Fig. 9. Example of merging two instances of the unitary U.

Item 1 means that **compile** *preserves the semantics of its input program, whereas item 2 means that the obtained circuit has a polynomial upper bound on the number of its gates and wires.*

Proof. From Corollary 1 and Theorem 10. ☐

Example 7. **compile**(QFT, n) *outputs the circuit provided in Figure 4 corresponding to the* foq *program* QFT *of Example 1. Notice that there is no extra ancilla as no procedure call appears in the branch of a quantum case.*

5.3 Polynomial-Size Circuits

We show Theorem 8 by exhibiting that any exponential growth of the circuit can be avoided by the **compile** algorithm using an argument based on orthogonal control structures. With a linear number of gates and a constant number of extra ancillas, we can merge calls referring to the same procedure, on different branches of a quantum case, when they are applied to sorted sets of equal size. An example of the construction is given in Figure 9, where two instances of a gate U are merged into one using *SWAP* gates and gates controlled by orthogonal control structures.

The following proposition shows that multiple uses of a gate can be merged in one provided they are applied to orthogonal control structures.

Lemma 4. *For any circuit $C_n \triangleq \circ_{i=1}^{k} U(cs_i, l_i)$, with a unitary gate U, pairwise orthogonal $cs_1, \ldots, cs_k \in$ Cst, and $l_1, \ldots l_k \in \mathcal{L}(\mathbb{N})$, there exists a circuit C using one controlled gate U, $O(kn)$ gates, and $O(k)$ ancillas, and such that $[\![C]\!] = [\![C_n]\!]$.*

Proof. We describe the circuit that achieves the result and then prove its correctness. Let a_i, for $i \in \{1, \ldots, k\}$, be ancillas in the zero state. Define $C \triangleq C^1 \circ G(\cdot[a_k = 1], l_k) \circ C^2$ with

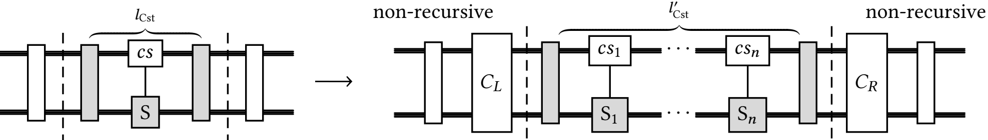
$$C^1 \triangleq \circ_{i=1}^{k} NOT(cs_i, a_i) \circ (\circ_{i=1}^{k-1} NOT(\cdot[a_i = 1], a_k)) \circ (\circ_{i=1}^{k-1} SWAP(\cdot[a_i = 1], l_i, l_k))$$

and C^2 defined from C^1 by reversing the order of the gates. The total number of ancillas is k. Circuits C^1 and C^2 contain a constant number of controlled *NOT* gates and, in the worst case, $O(kn)$ controlled *SWAP* gates (depending on the overlap between l_k and each l_i).

Consider a computational basis state $|x\rangle$, for $x \in \{0,1\}^n$. Since all control structures cs_i are pairwise orthogonal, after the two compositions of *NOT* circuits in C^1, $|x\rangle$ satisfies $\cdot[a_i := 1]$ iff it satisfies cs_i, for $i \in \{1, \ldots, k-1\}$. Moreover, $|x\rangle$ satisfies $\cdot[a_k := 1]$ iff there exists i such that it satisfies cs_i. Therefore, each *SWAP* sends wires in l_i into l_k iff $|x\rangle$ satisfies cs_i and then U is applied on l_k if it satisfies one of the control structure cs_i. As circuit C^2 reverses the actions of C^1, afterwards, the qubits are in the right positions, all ancillas are set to zero, and the result on non-ancillary qubits is the same as in C_n. ☐

The properties of **compile** shown thus far will allow us to argue the validity of the resulting circuit, by showing an invariant property of the list of controlled statements l_{Cst}. At each step of

optimize, we are handling one of the controlled statements in l_{Cst}, say (cs, S), such that we obtain a new list l'_{Cst} in the following way:



where C_L and C_R on the left and right, respectively, represent circuits not containing recursive calls. We let empty gates (either white or shaded) denote arbitrary parts of the circuit that can appear in the context of a rule. We use the vertical dashed lines to represent to separate the controlled statements in l_{Cst} from the rest of the circuit. The fact that l'_{Cst} will still only contain mutually orthogonal controlled statements will allow us to continually unpack each element (cs, S) in a way that preserves circuit semantics.

The case for each possible statement S for a controlled statement (cs, S) where S contains a recursive call, is given in Figure 10. To simplify notation, we represent using $(a|$ the addition of a *fresh ancilla* in state $|0\rangle$ with label a to the circuit. In Figure 10e (and and Figure 11), gates noted $\leftrightarrow$ represent the swapping of wires (see Figure 9).

We now show that orthogonality is an invariant property of **optimize**.

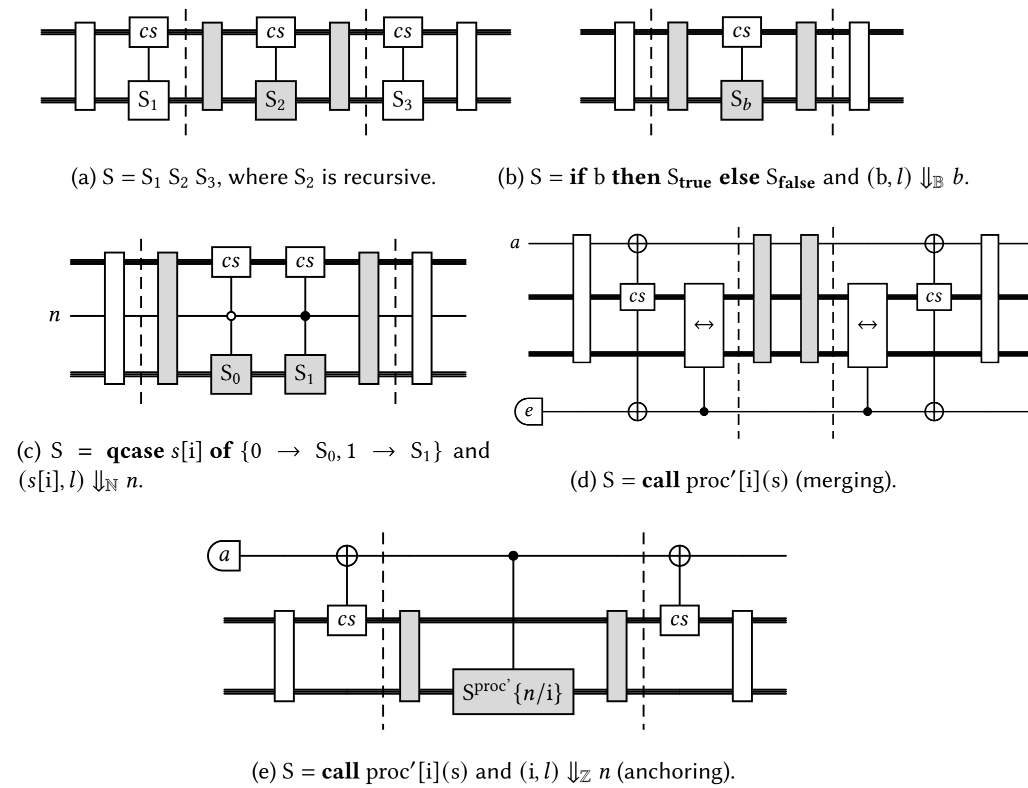
Lemma 5. *Orthogonality is an invariant property of the control structures in l_{Cst} of the subroutine* **optimize**. *In other words, for any two distinct pairs (cs, S), (cs', S') in l_{Cst}, cs and cs' are orthogonal.*

Proof. The proof is done by induction on a procedure statement, doing a case analysis on the rules of **optimize**. The initial value of l_{Cst} is of the form $[(cs, \mathrm{S}^{\mathrm{proc}})]$, which trivially satisfies the condition. Consider the possible cases for a pair (cs, S) in l_{Cst}. By induction hypothesis, assume that cs is orthogonal to all other control structures. For Composition, quantum case and classical **if**, the replacing pair is either of the shape (cs, S'), or it is two pairs with $cs[n := 0]$ and $cs[n := 1]$, therefore we conclude that the resulting l_{Cst} has pairwise orthogonal controls.

For a procedure call, we consider two cases. The first case occurs when a control statement $(\cdot[a := 1], \mathrm{S}^{\mathrm{proc}})$ is added to the list l_{Cst} and a is an ancilla controlled by cs. Since the original pair is removed, this construction also satisfies the invariant property. In the second case, there already exists an ancilla a assigned to the triple (proc, i, j). Let us argue by induction on the number of previous procedure call instances for this ancilla. Let $cs_1, \ldots, cs_k$ be control structures such that a computational basis state $|x\rangle$ satisfies $\cdot[a := 1]$ iff it satisfies one of the cs_i, which we assume by induction hypothesis are pairwise orthogonal. By the definition of PFOQ, cs will be orthogonal to each of the cs_i, and so after gate $NOT(cs, a)$, a state $|x\rangle$ will satisfy $\cdot[a := 1]$ iff it satisfies any of $cs, cs_1, \ldots, cs_k$. Since cs was orthogonal to all other control structures appearing in l_{Cst}, and the original pair is removed, the result l_{Cst} will have pairwise orthogonal controls. □

Theorem 9. *For any program* P *in* PFOQ, **compile**(P, n) *runs in time* $O(n^{2rk(\mathrm{P})+1})$.

Proof. We first show that an execution of the **optimize** subroutine performs $O(n^2)$ calls to **compr**. We use a simple counting argument. The dictionary Anc ensures that ancillas related to the same $(\mathrm{proc}', \mathrm{i}, \mathrm{j}) \in \mathrm{Procedures} \times \mathbb{Z} \times \{1, \ldots, n\}$ will only be created once. The classical parameter can either be updated to a new constant or by adding (or subtracting) a constant and this can be done $O(n)$ times as procedure calls also remove at least an element from the sorted set parameter. So the range of values for i is $O(n)$, therefore the space of possible values has size $O(n^2)$. Hence $O(n^2)$ ancillas can be created, which also bounds the number of calls to **compr**.



(a) $\mathrm{S} = \mathrm{S}_1\ \mathrm{S}_2\ \mathrm{S}_3$, where S_2 is recursive.

(b) $\mathrm{S} = \textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_{\textbf{true}}\ \textbf{else}\ \mathrm{S}_{\textbf{false}}$ and $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} b$.

(c) $\mathrm{S} = \textbf{qcase}\ s[\mathrm{i}]\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ and $(s[\mathrm{i}], l) \Downarrow_{\mathbb{N}} n$.

(d) $\mathrm{S} = \textbf{call}\ \mathrm{proc}'[\mathrm{i}](\mathrm{s})$ (merging).

(e) $\mathrm{S} = \textbf{call}\ \mathrm{proc}'[\mathrm{i}](\mathrm{s})$ and $(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n$ (anchoring).

Fig. 10. A step of the **optimize** subroutine.

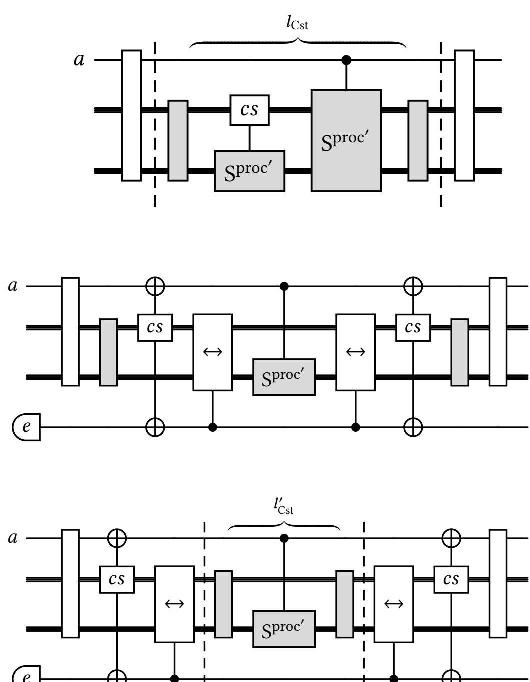
Second, consider an execution of **compr**. Subroutine **compr** calls itself directly only outside of recursive procedure calls, meaning that these cases consist of constant time circuit constructions. On the case of a call to a recursive procedure, it does a single call to **optimize** which creates $O(n^2)$ calls back to **compr**. Each of these instances of **compr** either does a non-recursive circuit construction of constant size, or does a call to **optimize** for a procedure of strictly lower rank, as defined in the proof of Lemma 2. Since there are $O(n^2)$ such calls, the maximum total number of procedure calls is $O(n^{2\,rk(\mathrm{P})})$, where $rk(\mathrm{P})$ is the rank of P (Definition 5). Each of these recursive procedure calls can have a constant number of merge constructions with linear complexity, therefore the total number of gates is $O(n^{2rk(\mathrm{P})+1})$. ◻

The final size of the generated circuit is bounded by the number of steps of the **compile** algorithm, as each step adds at most a constant number of gates and ancillas.

Corollary 1. *For any program* P *in* PFOQ, *if* $C_n \triangleq \textbf{compile}(\mathrm{P}, n)$ *we have that* $\#C_n = O(n^{2rk(\mathrm{P})+1})$.

We now show that the **compile** algorithm is semantics-preserving.

Theorem 10. *The circuit obtained in* $\textbf{compile}(\mathrm{P}, n)$ *implements the* PFOQ *program* P *applied to an input of size* n.



We assume two identical procedure statements, $S^{\text{proc}'}$, for same input size and same integer input value, where one instance has already been anchored on ancilla a. From the orthogonality property in Lemma 5, the elements in l_{Cst} commute and we may make the two repeated procedure calls adjacent.

We merge the two instances as in Lemma 4. Since $\cdot[a = 1]$ and cs are orthogonal by hypothesis, we may add cs to the ancilla a. A new ancilla e is created to control the set of swap gates, represented as $\leftrightarrow$.

By orthogonality, merging is adjacent to $S^{\text{proc}'}$ and does not contain any recursive calls, and we thus obtain a new list of recursive controlled statements, l'_{Cst}.

Fig. 11. Circuit representation illustrating the correctness of merging.

Proof. This can be shown by checking that each step of **compile** and **optimize** preserves the semantics of the program. This is easy to check in **compile**, so we will focus on the steps performed in **optimize**.

The **optimize** subroutine consists of simplification steps performed on the list of controlled statements, until the list is empty and the subroutine terminates. We can therefore simply show that each simplification preserves the semantics of the treated statement. This can be done by inspecting the rules in Figure 10 and checking their correctness.

For instance, in case 10a, we obtain the circuit by replacing (cs, S) by $(cs, \text{S}_1) \circ (cs, \text{S}_2) \circ (cs, \text{S}_3)$, where S_2 is recursive. Since cs is orthogonal to all other control structures in l_{Cst}, we have that (cs, S_1) and (cs, S_3) commute with all these controlled statements (but not each other) and can be moved to the left and right, respectively, into the non-recursive parts of the circuit. The cases of quantum (10c) and boolean (10b) control are trivial.

In the case where the procedure call is repeated (same procedure, same input length, and same classical input value), we obtain the circuit in Figure 10d. The validity of merging is based on the reasoning in Figure 11.

For a procedure call $\text{S} = \textbf{call}\ \text{proc}'[\text{i}](\text{s})$, where it is the first instance of proc' for a given input size and classical input (Figure 10e), we may replace (cs, S) with $(cs, \text{S}^{\text{proc}'})$ applied on the appropriate qubits. To obtain the final circuit, we simply direct the control structure cs into a fresh ancilla and proceed inductively on the procedure body. ◻

```
decl compex(p̄){
  if |p̄| > 2 then
    qcase p̄[1] of
    { 0 → call compex(p̄ ⊖ [1]);,
      1 → qcase p̄[2] of
        { 0 → skip;,
          1 → call compex(p̄ ⊖ [1, 2]); } }
  else p̄[1] *= U; } ::
call compex(q̄);
```

Fig. 12. Illustrative example for the compilation procedure.

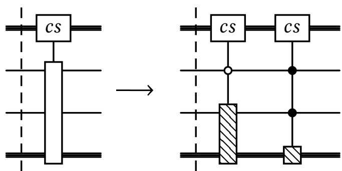


Fig. 13. One step of **optimize** on the controlled statement (cs, S^{compex}).

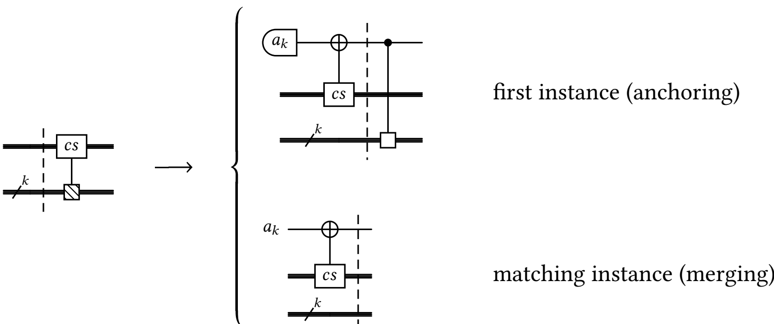


Fig. 14. One step of **optimize** on the controlled statement $(cs, \textbf{call}\ \text{compex}[\text{i}](\text{s}))$.

5.4 Compilation Example

We will now consider the example of the PFOQ program provided in Figure 12 and show how the **compile** algorithm works to generate a polynomial-size circuit that simulates it for a given input size. This program has a single procedure compex with two recursive calls on inputs of different sizes and makes use of the arbitrary operator U.

In our compilation example, we will consider the simple case where the input size is large enough so that we remain in the recursive case. Therefore, each of compilation will be one step of the **optimize** subroutine (Algorithm 2). Our compilation will consist of three operations: implementing the recursive call of compex, anchoring, and merging. The circuit simulating the procedure body S^{compex} and the circuit simulating a call "**call** compex" will be denoted:

$$\mathrm{S}^{\text{compex}} : \square \qquad \textbf{call}\ \text{compex} : ▧$$

By the definition of compex, its procedure body consists of two recursive calls on two distinct branches, which we expand directly, as in Figure 13, where *cs* represents a given control structure, and the vertical dashed line represents the separation of recursive and non-recursive statements (as in Figure 10).

Notice that while we have at most one recursive call per branch, in Figure 13 the branches must appear in sequence and so the complexity is given by the sum over the branches. This is the problem identified in [Yuan and Carbin 2022] as *branch sequentialization*, which arises in problems that involve branching over quantum data: "Branch sequentialization denotes the fact that the time complexity of a conditionally branching quantum program may be larger than its classical equivalent and the developer may need to structure the program differently to avoid this slowdown."

In [Yuan and Carbin 2022], the authors address this problem in the context of a binary tree traversal by restructuring their program to create an ancilla which controls the branching case. We will see in this example that the restrictions of PFOQ are strong enough that we can solve this problem at the compiler level, even with the added difficulty that the quantum branching occurs with procedure calls on different input sizes.

In Figure 14, we identify the two cases that arise in simplifying procedure calls: in anchoring, we direct the control structure *cs* to a fresh ancilla a_k, which we then use to control the procedure statement. This ancilla is indexed by k as it depends on the number k of qubits on which the procedure is called; in merging, since the ancilla has already been created, we simply redirect *cs*, which makes it so that we only need to expand one procedure for each size (notice that in this case the merging does not require any controlled swapping).

To graphically represent the creation of ancillas during the circuit compilation, we use an *ancilla diagram*, where each node a_k represents the ancilla anchoring the body of procedure compex on input size k. Edges in the diagram denote Toffoli gates connecting ancillas – in this case, each node a_k will have two outgoing edges, ∘ and ••, directed to nodes a_{k-1} and a_{k-2}, respectively. When the node represents a procedure call that has not yet been treated, it is denoted by an asterisk. The ancilla diagram serves mainly to aid the reader in visualizing the state of each ancilla, and the number of such ancillas that are created during the compilation.

Figures 15 and 16 illustrate the first steps of the compilation process, for an input $\bar{\mathrm{q}}$ of size n. The dashed line separates the non-recursive (left) and recursive (right) parts of the circuit, with the right part corresponding to the list of controlled recursive statements l_{Cst} changed in each step. One can verify that the semantics is conserved with each operation (anchoring, merging, and application of the procedure definition). During the compilation we have considered that n is large enough that we were always in the recursive case. Since compex does not use an integer input, procedure calls will only differ on input size, and since the merging cost is $O(1)$ (wire swapping is not required) we obtain a circuit that grows linearly on the input size n.

6 Conclusion

Sound implementations of quantum programs require programming languages that allow for reasoning about desirable properties, one of which being their compilation into efficient quantum circuit families. In this work, we presented FOQ, a quantum programming language with a fragment PFOQ which represents the first programming-language-based characterization of quantum polynomial time transformations. We showed the expressivity of PFOQ with quite a few examples of efficient quantum programs, and we demonstrated that all PFOQ programs can be efficiently compiled into quantum circuits of polynomial size. In the latter objective, we addressed the problem of branch sequentialization using an anchoring-and-merging technique when building the circuit.

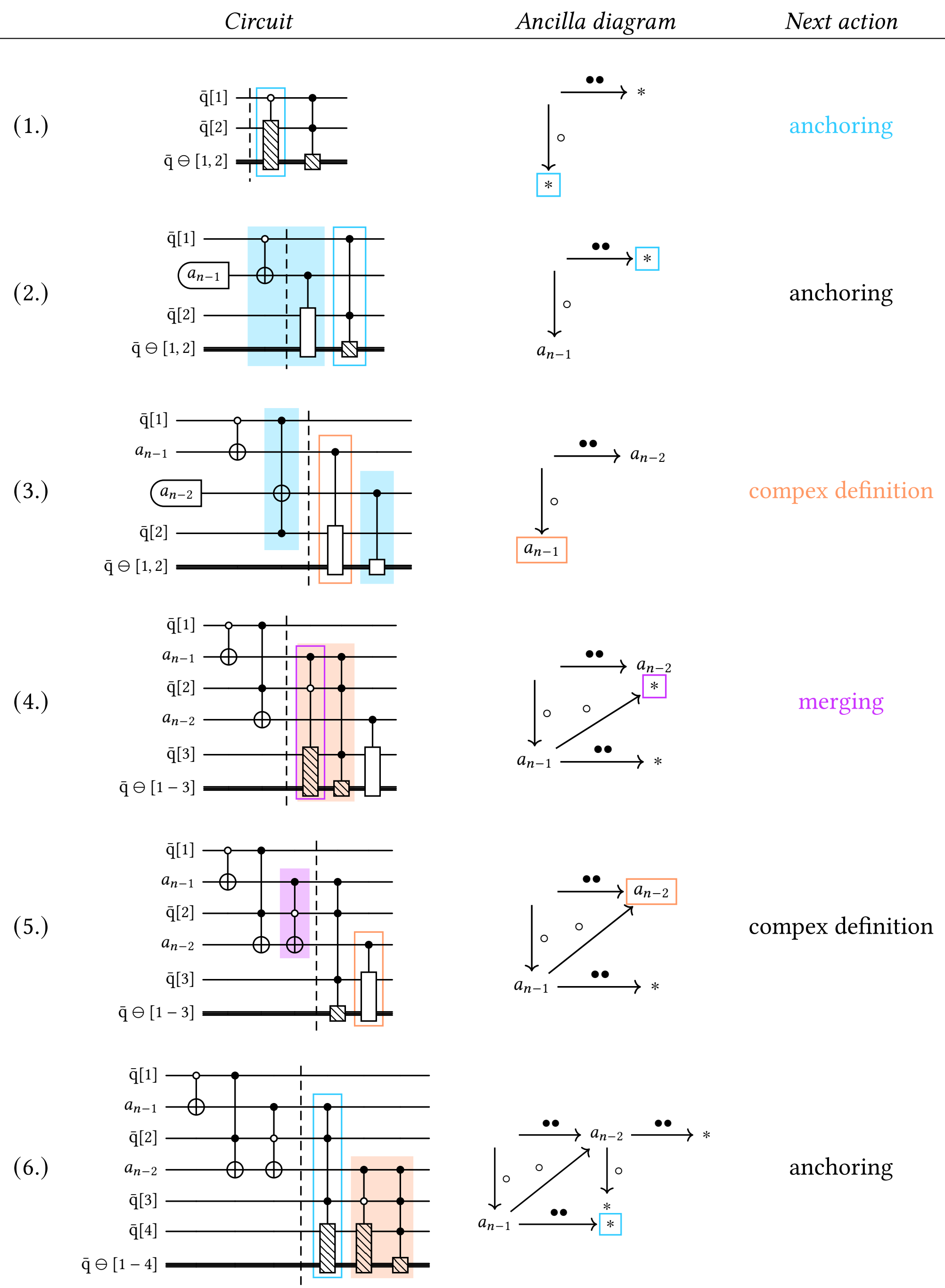


Fig. 15. Circuit construction (part 1/2).

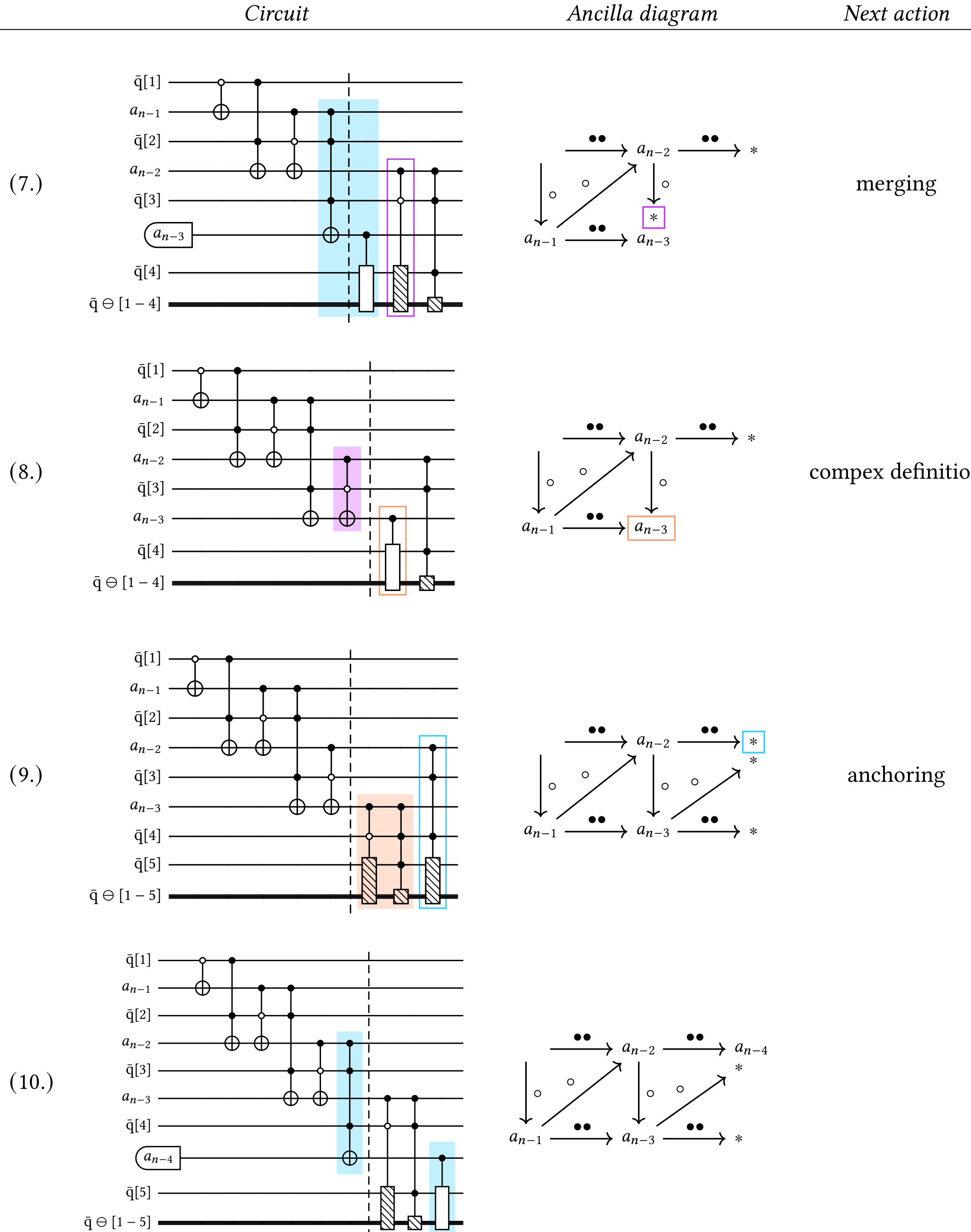


Fig. 16. Circuit construction (part 2/2).

We believe the simplicity of FOQ syntax lends itself well to the development of interesting quantum compilation strategies related to specific complexity classes. More recently, a variant of FOQ including divide-and-conquer techniques has been shown to be sound and complete for quantum poly-logarithmic time [Ferrari et al. 2025]. Furthermore, a fragment of PFOQ that is still complete for quantum polynomial time has been shown to avoid the problem of branch sequentialization altogether [Hainry et al. 2025].

Open problems in resource estimation of quantum programming languages remain, such as the investigation of techniques related to *spatial parallelization* in quantum circuits. While the characterization of quantum poly-logarithmic time in [Ferrari et al. 2025] admits a compilation strategy into uniform quantum circuits of polynomial size and poly-logarithmic depth, the complete class of such circuits, named QNC [Moore and Nilsson 2001], has not yet been implicitly characterized. This contrasts with the fact that its classical counterpart, NC, admits quite a few implicit characterizations, e.g. [Bonfante et al. 2006; Leivant 1998; Marion and Péchoux 2008].

Acknowledgments

This work is supported by the the Plan France 2030 through the PEPR integrated project EPiQ ANR-22- PETQ-0007, the HQI platform ANR-22-PNCQ-0002 and by the European projects NEASQC and HPCQS. The authors would like to thank Alexandre Guernut for his help on the compiler implementation and the anonymous reviewers for their constructive and useful comments.

References

Leonard M. Adleman, Jonathan DeMarrais, and Ming-Deh A. Huang. 1997. Quantum computability. *SIAM J. Comput.* 26, 5 (1997), 1524–1540. doi:10.1137/S0097539795293639

Martin Avanzini, Georg Moser, Romain Péchoux, and Simon Perdrix. 2024. On the hardness of analyzing quantum programs quantitatively. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 31–58. doi:10.1007/978-3-031-57267-8_2

Martin Avanzini, Georg Moser, Romain Péchoux, Simon Perdrix, and Vladimir Zamdzhiev. 2022. Quantum expectation transformers for cost analysis. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22)*. ACM, 1–13. doi:10.1145/3531130.3533332

Stephen Bellantoni and Stephen Cook. 1992. A new recursion-theoretic characterization of the polytime functions. *computational complexity* 2, 2 (01 Jun 1992), 97–110. doi:10.1007/BF01201998

Charles H. Bennett. 1973. Logical reversibility of computation. *IBM Journal of Research and Development* 17, 6 (1973), 525–532. doi:10.1147/rd.176.0525

Charles H. Bennett and Gilles Brassard. 2014. Quantum cryptography: public key distribution and coin tossing. *Theoretical Computer Science* 560 (dec 2014), 7–11. doi:10.1016/j.tcs.2014.05.025

Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.* 70 (Mar 1993), 1895–1899. Issue 13. doi:10.1103/PhysRevLett.70.1895

Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM J. Comput.* 26, 5 (1997), 1411–1473. doi:10.1137/S0097539796300921

G. Bonfante, R. Kahle, J. Y. Marion, and I. Oitavem. 2006. Towards an implicit characterization of NC^k. In *Computer Science Logic*, Zoltán Ésik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–224. doi:10.1007/11874683_14

P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. 1999. On universal and fault-tolerant quantum computing. doi:10.48550/ARXIV.QUANT-PH/9906054

Hans J. Briegel, David E. Browne, Wolfgang Dür, Robert Raussendorf, and Maarten Van den Nest. 2009. Measurement-based quantum computation. *Nature Physics* 5, 1 (2009), 19–26. doi:10.1038/nphys1157

Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. 2003. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM. doi:10.1145/780542.780552

Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (sep 2018), 9456–9461. doi:10.1073/pnas.1801723115

Andrea Colledan and Ugo Dal Lago. 2024. Circuit width estimation via effect typing and linear dependency. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 3–30. doi:10.1007/978-3-031-57267-8_1

Andrea Colledan and Ugo Dal Lago. 2025. Flexible type-based resource estimation in quantum circuit description languages. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 1386–1416. doi:10.1145/370488

Ugo Dal Lago. 2011. A short introduction to implicit computational complexity. In *ESSLLI 2010*. 89–109. doi:10.1007/978-3-642-31485-8_3

Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. 2010. Quantum implicit computational complexity. *Theoretical Computer Science* 411, 2 (2010), 377–409. doi:10.1016/j.tcs.2009.07.045

Vincent Danos and Elham Kashefi. 2006. Determinism in the one-way model. *Physical Review A* 74, 5 (2006), 052310. doi:10.1103/PhysRevA.74.052310

David E. Deutsch. 1989. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 425, 1868 (1989), 73–90. doi:10.1098/rspa.1989.0099

Artur K. Ekert. 1991. Quantum cryptography based on Bell's theorem. *Phys. Rev. Lett.* 67 (Aug 1991), 661–663. Issue 6. doi:10.1103/PhysRevLett.67.661

Florent Ferrari, Emmanuel Hainry, Romain Péchoux, and Mário Silva. 2025. Quantum programming in polylogarithmic time. (July 2025). https://hal.science/hal-05171674 working paper or preprint.

Richard P. Feynman. 1982. Simulating physics with computers. *International Journal of Theoretical Physics* 21, 6 (01 Jun 1982), 467–488. doi:10.1007/BF02650179

Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. doi:10.1145/237814.237866

Emmanuel Hainry, Romain Péchoux, and Mário Silva. 2023. A programming language characterizing quantum polynomial time. In *Foundations of Software Science and Computation Structures*, Orna Kupferman and Pawel Sobocinski (Eds.). Springer Nature Switzerland, Cham, 156–175. doi:10.1007/978-3-031-30829-1_8

Emmanuel Hainry, Romain Péchoux, and Mário Silva. 2025. Branch sequentialization in quantum polytime. In *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:22. doi:10.4230/LIPIcs.FSCD.2025.22

Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-based optimization of quantum programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–20. doi:10.1145/3428201

Ker-I. Ko and Harvey Friedman. 1982. Computational complexity of real functions. *Theoretical Computer Science* 20, 3 (1982), 323–352. doi:10.1016/S0304-3975(82)80003-0

D. Leivant. 1998. A characterization of NC by tree recurrence. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*. 716–724. doi:10.1109/SFCS.1998.743522

Saunders MacLane. 1965. Categorical algebra. *Bull. Amer. Math. Soc.* 71, 1 (1965), 40–106. doi:10.1090/S0002-9904-1965-11234-4

Jean-Yves Marion and Romain Péchoux. 2008. A characterization of $\textsc{nc}^k$ by first order functional programs. In *Theory and Applications of Models of Computation*, Manindra Agrawal, Dingzhu Du, Zhenhua Duan, and Angsheng Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–147. doi:10.1007/978-3-540-79228-4_12

Giulia Meuli, Mathias Soeken, Martin Roetteler, and Thomas Häner. 2020. Enabling accuracy-aware quantum compilers using symbolic resource estimation. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–26. doi:10.1145/3428198

Cristopher Moore and Martin Nilsson. 2001. Parallel quantum computation and quantum codes. *SIAM J. Comput.* 31, 3 (2001), 799–815. doi:10.1137/S0097539799355053

Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum computation and quantum information: 10th anniversary edition*. Cambridge University Press. doi:10.1017/CBO9780511976667

Masanao Ozawa and Harumichi Nishimura. 2000. Local transition functions of quantum Turing machines. *RAIRO-Theoretical Informatics and Applications* 34, 5 (2000), 379–402. doi:10.1051/ita:2000123

Romain Péchoux. 2020. Implicit computational complexity: past and future. Mémoire d'Habilitation à Diriger des Recherches. https://tel.archives-ouvertes.fr/tel-02978986 Université de Lorraine.

Neil J. Ross. 2015. Algebraic and logical methods in quantum computation. PhD thesis. doi:10.48550/ARXIV.1510.02198

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586. doi:10.1017/S0960129504004256

Peter W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. doi:10.1109/SFCS.1994.365700

John van de Wetering, Richie Yeung, Tuomas Laakkonen, and Aleks Kissinger. 2025. Optimal compilation of parametrised quantum circuits. arXiv:2401.12877

Tomoyuki Yamakami. 1999. A foundation of programming a multi-tape quantum Turing machine. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 430–441. doi:10.1007/3-540-48340-3_39

Tomoyuki Yamakami. 2020. A schematic definition of quantum polynomial time computability. *J. Symb. Log.* 85, 4 (2020), 1546–1587. doi:10.1017/jsl.2020.45

Andrew Chi-Chih Yao. 1993. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*. 352–361. doi:10.1109/SFCS.1993.366852

Charles Yuan and Michael Carbin. 2022. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct 2022), 259–288. doi:10.1145/3563297