Branch Sequentialization in Quantum Polytime

Emmanuel Hainry, Romain Péchoux and Mário Silva[1]
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract

Quantum computation leverages the use of quantumly-controlled conditionals in order to achieve computational advantage. However, since the different branches in the conditional may operate on the same qubits, a typical approach to compilation involves performing the branches sequentially, which can easily lead to an exponential blowup of the program complexity. We introduce and study a compilation technique for avoiding branch sequentialization in a language that is sound and complete for quantum polynomial time, improving on previously existing polynomial-size bounds and showing the existence of techniques that preserve the intuitive complexity of the program.

2012 ACM Subject Classification Theory of computation → Quantum complexity theory; Theory of computation → Quantum complexity theory

Keywords and phrases Formal methods, Quantum computation, Implicit Computational Complexity

1 Introduction

Quantum computing is an emerging paradigm of computation where quantum physical phenomena, such as entanglement and superposition, are used to obtain an advantage over classical computation. A testament to the richness of the field is the variety of computational models: quantum Turing machines [3], quantum circuits [18, 16], measurement-based quantum computation [4, 6], linear optical circuits [13], among others. Some of these models have been shown to be equivalent in terms of computational power and complexity. For instance, Yao's equivalency result [18] shows that polynomial-time quantum Turing machine are computationally equivalent to uniform and poly-size quantum circuit families.

A lot of effort has been put on developing high-level quantum programming languages to allow programmers to abstract themselves from the technicalities of these low-level models. Towards that end, several verification techniques such as type systems [9] or categorical approaches for reasoning on programs semantics [2, 11] have been studied and developed to ensure the physical reality of compiled programs, for example, by ensuring that it preserves the main properties of quantum mechanics such as no-cloning theorem [1] or unitarity [8]. An important line of research in this area involves checking polytime termination of quantum programs [5, 17, 10].

By Yao's Theorem, this property implies the feasibility of the corresponding quantum circuit by ensuring that its size is polynomially bounded in the program input size. However, there are still quite a few obstacles to the full use of these techniques. In particular, designing efficient compilation strategies is not trivial [10].

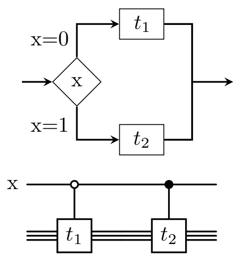


Figure 1 Classical vs. quantum branching.

A prominent example is the time complexity of *quantum branching* in programs, i.e., when the flow in a loop or in a conditional is determined upon the state of a qubit. In the classical setting, the cost of branching is the maximum cost between the two branches (Figure 1). However, this is not necessarily the case in the quantum setting, as a consequence of no-cloning: in a quantum circuit, the two branches may

[1] Mário Silva will present this work if selected.

contain operations on the same qubits, and thus require an implementation in series. This results in a circuit whose total depth is the sum of the depths of the two branches as illustrated by Figure 1. In [19], while trying to encode efficient operations over quantum data structures, the authors encounter this problem which they have coined *branch sequentialization*. While the authors provide a heuristic for avoiding branch sequentialization, it is only applicable in a few precise examples. Given the importance of preserving the time complexity of a program in its circuit implementation, there is an interest in discovering general techniques that avoid the problem of branch sequentialization altogether.

Motivating Example. Consider the program PAIRS defined in Figure 2. The procedure `pairs` takes as input a sorted set $\bar{\mathrm{q}}$ of qubits (i.e., a collection of pairwise distinct qubits) on which it will perform operations. By language design, `pairs` immediately terminates whenever $\bar{\mathrm{q}}$ is empty. First, `pairs` checks that the number of qubits in $\bar{\mathrm{q}}$, given by its size $|\bar{\mathrm{q}}|$, is larger than 1 to enter the recursive case, otherwise it applies a NOT gate to the remaining qubit (line 9). On line 3, the program will branch depending on the state $\bar{\mathrm{q}}[1,2]$ of the first two qubits in $\bar{\mathrm{q}}$. Out of all four cases (lines 4-7), `pairs` only performs an operation when the first two qubits are in state $|00\rangle$ or $|11\rangle$, in which case it performs a recursive call on $\bar{\mathrm{q}} \ominus [1,2]$, the sorted set $\bar{\mathrm{q}}$ where the first and second qubits have been removed.

With $x \in \{0,1\}^*$ and $y \in \{0,1\}$, given the input state $|xy\rangle$, `pairs` will apply a *NOT* gate to y if and only if x is a string consisting only of sequences of 00 and 11. Put another way, `pairs` encodes a unitary transformation that inverts the state of the last qubit of an input when x belongs to the regular language defined by $(00 \mid 11)^*$.

Since `pairs` performs at most one call per branch, and consumes 2 qubits from its input while doing so, we conclude that its runtime complexity is bounded linearly.

```
1  decl pairs(q̄){
2    if |q̄| > 1 then
3      qcase q̄[1,2] of {
4        00 → call pairs(q̄ ⊖ [1,2]);
5        01 → skip;
6        10 → skip;
7        11 → call pairs(q̄ ⊖ [1,2]);
8        }
9    else q̄[1] *= NOT; }
10 :: call pairs(q̄);
```

Figure 2 Branching program PAIRS.

Let us now turn to finding a circuit implementation for the recursive case of `pairs`. Consider the two compilation strategies **(a)** and **(b)** shown in Figure 3. While Strategy **(a)** could be considered the more direct approach to building the circuit, at each recursive call the size of the circuit for `pairs` is the sum of the sizes of each branch. On the other hand, while the strategy in **(b)** requires the creation of an ancilla and the use of extra Toffoli gates, it only requires the implementation of one call to `pairs`. As a consequence, the strategies **(a)** and **(b)** result in circuits of depth $\Theta(|\bar{\mathrm{q}}|2^{|\bar{\mathrm{q}}|})$ and $\Theta(|\bar{\mathrm{q}}|)$, respectively, showing how implementing the branches sequentially can result in an exponential blowup in circuit size. It is simple enough to find a compilation strategy that prevents the duplication of `pairs` in the recursive case. However, this becomes much less trivial once we consider programs with more complex recursive calls.

Contribution. In this paper, we study the problem of branch sequentialization and solve it in the case of quantum polynomial time, in the following way:

- We identify a programming language fragment BFOQ (for Basic FOQ) that is sound and complete for quantum polynomial time (Theorem 9). That is, any BFOQ program computes a function in FBQP, the class of functions computable in polynomial time by a quantum Turing machine with bounded error. Conversely, any function in FBQP can be computed by a BFOQ program. BFOQ is a strict but expressive subset of the PFOQ programming language of [10] whose expressive power is illustrated through many

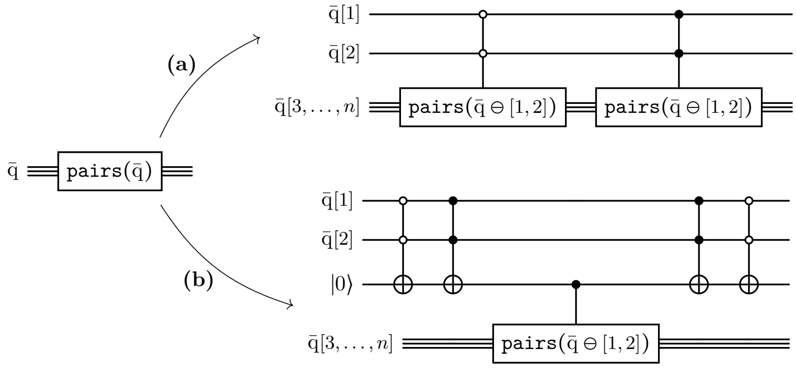


Figure 3 Compilation strategies: branch sequentialization **(a)** vs optimized approach **(b)**.

examples (see Table 1);

- We introduce a compilation strategy **compile**$^+$ from PFOQ to quantum circuits based on two subroutines **compr**$^+$ (Algorithm 1) and **optimize**$^+$ (Algorithm 2): while **compr**$^+$ just generates the compiled circuit by a simple structural induction on program statements, **optimize**$^+$ perform some optimization by merging (recursive) procedure calls in different branches in the program.
- We show that the **compile**$^+$ is sound, i.e., the generated circuit fairly simulates the input program: this correctness result lies on the orthogonality of the control structures used in the **optimize**$^+$ subroutine (Lemma 11).
- On PFOQ programs, we exhibit a direct improvement on size complexity of the generated circuit with respect to the compilation algorithm studied in [10] (Theorem 12).
- We show that, on BFOQ programs, **compile**$^+$ produces circuits whose size is asymptotically bounded by their level (Theorem 13), i.e., by the maximal number of consecutive procedure calls in all branches (including quantum ones) of a program execution, thus avoiding branch sequentialization on a sound and complete language for quantum polynomial time.

Related work. Resource optimization in quantum computing is a well-studied subject for low level computational models such as quantum circuits or ZX-diagrams: in this *constant-depth* scenario, (i.e., taking a specific and fixed circuit of constant size and, thus, constant depth), is it possible to reduce its number of gates [15, 14] (or at least its number of non-Clifford gates [12, 7]), with techniques such as gate substitution, graph-rewriting, among others.

Resource optimization for high-level quantum programs is still a relatively undeveloped research area as it involves the asymptotic consideration of families of circuits. Such an issue has strong connections with programming language-based characterizations of quantum polynomial time classes [17, 5, 10] as, by design, their set of programs is sound and complete for uniform families of quantum circuits of polynomial size, as per Yao's equivalency theorem. While [17, 5] provide non-constructive proofs of the existence of quantum circuits of polynomial size, [10] introduces a programming language that avoids an exponential blowup in the complexity of recursive (quantum) branching with a direct compilation strategy for ensuring polysized circuit representations. However this strategy still performs branch sequentialization and generates polynomial bounds whose degree is not accurate.

(Programs)	$\mathrm{P}(\bar{\mathrm{q}})$	$\triangleq$	$\mathrm{D} :: \mathrm{S}$
(Procedure declarations)	D	$\triangleq$	ε | **decl** $\mathrm{proc}[\mathrm{x}](\bar{\mathrm{q}})\{\mathrm{S}\}, \mathrm{D}$
(Statements)	S	$\triangleq$	**skip**; | $\bar{\mathrm{q}}[\mathrm{i}]$ $\ast\!\!=$ $\mathrm{U}^f(\mathrm{j})$; | S S | **if** b **then** S **else** S | **qcase** $\bar{\mathrm{q}}[\mathrm{i}]$ **of** $\{0 \to \mathrm{S}, 1 \to \mathrm{S}\}$ | **call** $\mathrm{proc}[\mathrm{i}](\mathrm{s})$;

Figure 4 Syntax of FOQ programs.

2 First-Order Quantum Programming Language

We consider the FOQ (First-Order Quantum) programming language with quantum control, introduced in [10] to characterize quantum polynomial time. A complete account of its syntax and its semantics is given in Appendix A.

2.1 Syntax

A FOQ program $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$ is defined in Figure 4 by a list of procedure declarations D and a program statement S . The language include 4 basic datatypes for expressions. *Sorted set* expressions s are either variables $\bar{\mathrm{q}}$, the empty sorted set nil, or $\mathrm{s} \ominus [\mathrm{i}]$, the sorted set s where the i-th element has been removed. Intuitively, a sorted set is a list of unique (i.e., non-duplicable) qubit pointers. *Integer* expressions, noted i, j, are either an integer variable x, a constant n, an addition by a constant $\mathrm{i} \pm n$ or the size of a sorted set $|\mathrm{s}|$. *Boolean* expressions b are defined in a standard way using boolean operators and arithmetic operators, e.g., $\mathrm{i} > \mathrm{j}$. Finally, *qubit* expressions are of the shape $\mathrm{s}[\mathrm{i}]$ which denotes the i-th qubit pointed to in s. $\mathrm{s}[\mathrm{i}_1, \dots, \mathrm{i}_n]$ is a shorthand for $\mathrm{s}[\mathrm{i}_1], \dots, \mathrm{s}[\mathrm{i}_n]$. Finally, we also allow for the syntactic sugar on sorted state of pointing to the n-th *last* qubit in the set, by defining for any $n \geq 1$, $\bar{\mathrm{q}}[-n] \triangleq \bar{\mathrm{q}}[|\bar{\mathrm{q}}| - n + 1]$.

A procedure of name proc is defined by a procedure declaration **decl** $\mathrm{proc}[\mathrm{x}](\bar{\mathrm{q}})\{\mathrm{S}^{\mathrm{proc}}\}$ which takes a sorted set $\bar{\mathrm{q}}$ and an (optional) integer x as input parameters and executes the *procedure statement* $\mathrm{S}^{\mathrm{proc}}$. Let Procedures be an enumerable set of procedure names. We will write S instead of $\mathrm{S}^{\mathrm{proc}}$ when the procedure is clear from context, and we denote by $\mathrm{proc} \in \mathrm{P}$ the fact that proc appears in D. Given two statements S, S', $\mathrm{S} \in \mathrm{S}'$ denotes the fact that S is a substatement of S'. Furthermore, we have that $\mathrm{proc} \in \mathrm{S}$ holds if there are i and s such that **call** $\mathrm{proc}[\mathrm{i}](\mathrm{s}); \in \mathrm{S}$.

Statements include the no-op instruction, unitary operations, sequences, classical and quantum conditionals, and procedures calls. Of these, we highlight the quantum conditional **qcase** $\bar{\mathrm{q}}[\mathrm{i}]$ **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$, which allows branching by executing statements S_0 and S_1 in superposition according to the state of qubit $\bar{\mathrm{q}}[\mathrm{i}]$, and also the procedure call **call** $\mathrm{proc}[\mathrm{i}](\mathrm{s})$;, which runs procedure proc with *integer expression* i and *sorted set* expression s, a list of unique qubit pointers. The quantum conditional can be extended to n qubits **qcase** $\bar{\mathrm{q}}[\mathrm{i}_1, \dots, \mathrm{i}_n]$ **of** $\{0^n \to \mathrm{S}_{0^n}, \dots, 1^n \to \mathrm{S}_{1^n}\}$ in a standard way as used in Figure 2.

In a statement $\bar{\mathrm{q}}[\mathrm{i}]$ $\ast\!\!=$ $\mathrm{U}^f(\mathrm{j})$;, if the integer expression j evaluates to n, then the unitary operator $[\![\mathrm{U}^f]\!](n)$ corresponding to the unary construct $\mathrm{U}^f(\mathrm{j})$ is applied to qubit $\bar{\mathrm{q}}[\mathrm{i}]$. For expressivity purposes, these constructs are parameterized by some polynomial-time approximable total function $f \in \mathbb{Z} \to [0, 2\pi)$ and some integer expression j. For example, the gates of the quantum Fourier transform can be defined by $R_n \triangleq [\![\mathrm{Ph}^{\lambda x.\pi/2^{x-1}}]\!](n)$ with $[\![\mathrm{Ph}^f]\!](n) \triangleq \left(\begin{smallmatrix} 1 & 0 \\ 0 & e^{if(n)} \end{smallmatrix}\right)$. Other basic unary gates are the NOT and the R_Y gate (see [10]). We also make use of some syntactic sugar to describe statements encoding constant-time quantum operations. For instance, the $CNOT$, $SWAP$, and Toffoli gates can be defined by:

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

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

$$\mathrm{TOF}(\bar{q}[i],\bar{q}[j],\bar{q}[k]) \triangleq \textbf{qcase } \bar{q}[i] \textbf{ of } \{0 \to \textbf{skip};, 1 \to \mathrm{CNOT}(\bar{q}[i],\bar{q}[j])\}$$

We define notions of rank that provide quantitative information on the recursion level of a given program or procedure.

▶ **Definition 1** (Rank). *Given a* FOQ *program* P*, the rank of a procedure* proc *in* P*, denoted* $rk_{\mathrm{P}}(\mathrm{proc})$*, is defined as follows:*

$$rk_{\mathrm{P}}(\mathrm{proc}) \triangleq \begin{cases} 0, & \textit{if } \neg(\exists \mathrm{proc}',\ \mathrm{proc} \geq_{\mathrm{P}} \mathrm{proc}'), \\ \max_{\mathrm{proc} \geq_{\mathrm{P}} \mathrm{proc}'}\{rk_{\mathrm{P}}(\mathrm{proc}')\}, & \textit{if } \exists \mathrm{proc}',\ \mathrm{proc} \geq_{\mathrm{P}} \mathrm{proc}' \wedge \neg(\mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}), \\ 1 + \max_{\mathrm{proc} >_{\mathrm{P}} \mathrm{proc}'}\{rk_{\mathrm{P}}(\mathrm{proc}')\}, & \textit{if } \mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}, \end{cases}$$

where $\max(\varnothing) \triangleq 0$*. The* rank of a program *is defined as the maximum rank among all procedures, i.e., for a program* $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$*, we have that* $rk(\mathrm{P}) \triangleq \max_{\mathrm{proc} \in \mathrm{D}} rk_{\mathrm{P}}(\mathrm{proc})$*.*

▶ **Example 2.** The program PAIRS given in Figure 2 has rank 1, since $rk(\mathrm{PAIRS}) \triangleq \max_{\mathrm{proc} \in \mathrm{PAIRS}} rk_{\mathrm{P}}(\mathrm{proc}) = rk(\texttt{pairs}) = 1$.

2.2 Semantics

Let $\mathcal{H}_{2^n}$ denote the Hilbert space of n qubits $\mathbb{C}^{2^n}$, $\mathcal{L}(\mathbb{N})$ denote the set of lists of natural numbers, and $\mathcal{P}(\mathbb{N})$ denote the powerset of natural numbers.

Expressions. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{C}^{2\times 2}, \mathcal{L}(\mathbb{N})\}$, we write $(\mathrm{e}, l) \Downarrow_{\mathbb{K}} v$ when the expression e evaluates to the value $v \in \mathbb{K}$ under the context $l \in \mathcal{L}(\mathbb{N})$. The context l is just the sorted set of qubit pointers into consideration when evaluating e. For example, we have that $(\bar{q}[2], [1,4,5]) \Downarrow_{\mathbb{N}} 4$ (the second qubit is of index 4), $(\bar{q}[4], [1,4,5]) \Downarrow_{\mathbb{N}} 0$ (index 0 is used for error), and $(\bar{q} \ominus [3], [1,4,5]) \Downarrow_{\mathcal{L}(\mathbb{N})} [1,4]$ (the third qubit has been removed).

Statements. Let $\mathrm{Conf}_n \triangleq (\mathrm{Statements} \cup \{\top, \bot\}) \times \mathcal{H}_{2^n} \times \mathcal{P}(\mathbb{N}) \times \mathcal{L}(\mathbb{N})$ be the set of configurations of n qubits. In a configuration $c \triangleq (\mathrm{S}, |\phi\rangle, \mathcal{S}, l)$, S is the statement to be executed, $|\phi\rangle$ is the quantum state, $\mathcal{S}$ is a set of accessible (pointers to) qubits and l is the list of qubit pointers under consideration. In case of error the program exits and the two special symbols $\top$ and $\bot$ are markers for success (termination) and failure (error), respectively. The set $\mathcal{S}$ of accessible qubits is used to ensure that unitary operations on qubits can be physically implemented. For example, statements S_0 and S_1 of a quantum branch **qcase** $\bar{q}[i]$ **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ cannot access $\bar{q}[i]$ to ensure that the operation can be physically implemented by a controlled-circuit.

The big-step semantics $\cdot \xrightarrow{\cdot} \cdot$ is defined as a relation in $\bigcup_{n \in \mathbb{N}} \mathrm{Conf}_n \times \mathbb{N} \times \mathrm{Conf}_n$. When $c \xrightarrow{m} c'$ holds the *level* m is an integer corresponding to the maximum number of procedure calls performed over each (condition and quantum) branch during the evaluation of c. More formally, the level of a program $\mathrm{P} = \mathrm{D} :: \mathrm{S}$ on n qubits, denoted $\mathrm{level}_{\mathrm{P}}(n)$, is defined by

$$\mathrm{level}_{\mathrm{P}}(n) \triangleq \max\{m \in \mathbb{N} \mid \exists\, |\phi\rangle, |\phi'\rangle \in \mathcal{H}_{2^n},\ (\mathrm{S}, |\phi\rangle, \mathcal{S}_n, l_n) \xrightarrow{m} (\top, |\phi'\rangle, \mathcal{S}_n, l_n)\},$$

with $\mathcal{S}_n \triangleq \{1, \ldots, n\}$ and $l_n \triangleq [1, \ldots, n]$.

▶ **Example 3.** Consider the program PAIRS of Figure 2. We have that each procedure call removes two qubits until it reaches a case of input size $|\bar{q}|$ either 1 or 0 (depending on if n is odd or even) and for both sizes there are no more procedure calls. On an empty sorted set, the program exits after the first call. Then, $\mathrm{level}_{\mathrm{PAIRS}}(n) = \lfloor \frac{n}{2} \rfloor + 1 = O(n)$.

2.3 Polytime fragments of FOQ

In [10], the polynomial-time fragment of FOQ, denoted PFOQ, is defined by placing two restrictions on procedure calls: a well-foundedness criterion for termination and a restriction on the number of admissible recursive calls per (classical or quantum) branch to avoid exponentiation.

PFOQ and Basic FOQ. Given a program $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$, the call relation $\rightarrow_{\mathrm{P}} \subseteq \text{Procedures} \times \text{Procedures}$ is defined for any two procedures $\mathrm{proc}_1, \mathrm{proc}_2 \in \mathrm{S}$ as $\mathrm{proc}_1 \rightarrow_{\mathrm{P}} \mathrm{proc}_2$ whenever $\mathrm{proc}_2 \in \mathrm{S}^{\mathrm{proc}_1}$. The partial order $\geq_{\mathrm{P}}$ is then the transitive closure of $\rightarrow_{\mathrm{P}}$, and $\sim_{\mathrm{P}}$ denotes the equivalence relation where $\mathrm{proc}_1 \sim_{\mathrm{P}} \mathrm{proc}_2$ if $\mathrm{proc}_1 \geq_{\mathrm{P}} \mathrm{proc}_2$ and $\mathrm{proc}_2 \geq_{\mathrm{P}} \mathrm{proc}_1$ both hold. **call** $\mathrm{proc}' \in \mathrm{S}^{\mathrm{proc}}$ is a *recursive* procedure call whenever $\mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}'$. The strict order $>_{\mathrm{P}}$ is defined as $\mathrm{proc}_1 >_{\mathrm{P}} \mathrm{proc}_2$ if $\mathrm{proc}_1 \geq_{\mathrm{P}} \mathrm{proc}_2$ and $\mathrm{proc}_1 \not\sim_{\mathrm{P}} \mathrm{proc}_2$ both hold.

A program P is in WF if it satisfies the constraint that each recursive procedure call removes at least one qubit in its parameter. Programs in WF are terminating (well-founded) but still allow the programmer to write programs with exponential runtime. To avoid such programs, we use the notion of *width of a procedure* proc in a program P, noted $\mathrm{width}_{\mathrm{P}}(\mathrm{proc})$, defined inductively on procedure declarations by counting the number of recursive calls in the procedure body, taking the maximum of each (classical or quantum) conditional.

▶ **Definition 4** (Width of a procedure). *Given* $\mathrm{P} \in$ FOQ *and* $\mathrm{proc} \in \mathrm{P}$*, 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:*

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

Let $\text{WIDTH}_{\leq 1}$ be the set of programs with procedures of width at most 1. Finally, define PFOQ $\triangleq$ FOQ$\cap$WF$\cap\text{WIDTH}_{\leq 1}$. We will show that PFOQ can be restricted to a fragment, denoted BFOQ (for Basic FOQ), that is still complete for polynomial time and where the compilation procedure conserves the intuitive complexity of the program. Let BASIC denote the set of programs where i) procedures do not use classical inputs ii) procedure call parameters are restricted to sorted set variables $\bar{\mathrm{q}}$ or to a fixed sorted set s (s is fixed for each BFOQ program). Then, we define BFOQ $\triangleq$ BASIC $\cap$ PFOQ. It trivially holds that BFOQ $\subsetneq$ PFOQ $\subsetneq$ FOQ.

▶ **Example 5** (Quantum Full Adder $\in$ BFOQ). Let *ADD* encode the unitary transformation such that, for $a, b \in \{0,1\}^n$ and $c_{\mathrm{in}}, c_{\mathrm{out}} \in \{0,1\}$, *ADD* performs the following transformation, where η represents the n least significant bits of $(a + b + c_{\mathrm{in}})$.

$$ADD\,|a_n b_n \dots a_1 b_1 0^n c_{\mathrm{in}}\rangle \triangleq |a_n b_n \dots a_1 b_1 c_{\mathrm{out}} \eta_1 \dots \eta_n\rangle\,,$$

where c_{in} and c_{out} encode the *carry-in* and *carry-out* values, respectively. The operator

ADD corresponds to the circuit in Figure 6 and is described by the program FA below.

```
decl fullAdder(q̄){
  if |q̄| > 3 then /* q̄[1] = a, q̄[2] = b, q̄[−2] = |0⟩ and q̄[−1] = c_in */
    TOF(q̄[1], q̄[2], q̄[−2])
    CNOT(q̄[1], q̄[2])
    TOF(q̄[2], q̄[−1], q̄[−2]) /* c_out = (a · b) ⊕ (c_in · (a ⊕ b)) */
    CNOT(q̄[2], q̄[−1]) /* η = a ⊕ b ⊕ c_in */
    CNOT(q̄[1], q̄[2])
    call fullAdder(q̄ ⊖ [1, 2, −1]);
  else skip; },
:: call fullAdder(q̄);
```

It holds that FA $\in$ FOQ $\cap$ WF and that $\text{width}_{\text{FA}}(\texttt{fullAdder}) = 1$. Therefore, FA $\in$ PFOQ. Also, FA is clearly in BFOQ as there is only one recursive call and no integer parameter.

▶ **Example 6** (Quantum Fourier Transform $\in$ BFOQ)**.** The quantum Fourier transform can be described by the program QFT below

```
decl qft(q̄){
  q̄[1] *= H;
  call rot(q̄);
  call shift(q̄);
  call qft(q̄ ⊖ [−1]); },

decl shift(q̄){
  if |q̄| > 1 then
    SWAP(q̄[1], q̄[−1])
    call shift(q̄ ⊖ [−1]);
  else skip; },

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

:: call qft(q̄);
```

The program consists of three procedures, `qft`, `shift`, and `rot`, and is in PFOQ since it is in WF and $\text{width}_{\text{QFT}}(\texttt{qft}) = \text{width}_{\text{QFT}}(\texttt{shift}) = \text{width}_{\text{QFT}}(\texttt{rot}) = 1$. All procedure calls are performed on the set $\bar{q}$ or $\bar{q} \ominus [-1]$, and therefore the program is also in BASIC. This program can be compiled to the circuit Figure 5 for input size 4, implementing the quantum Fourier transform. This circuit differs from (but is equivalent to) the standard implementation of the quantum Fourier transform. This is due to some of the restrictions put in BFOQ. The standard circuit can be obtained directly through compilation of a PFOQ program.

Properties of PFOQ and BFOQ programs. PFOQ programs have a consecutive number of procedure calls in the distinct branches of their execution (i.e., level) that is bounded polynomially in the input size (number of qubits). The degree of the polynomial can be obtained from the rank, which can be inferred syntactically.

▶ **Lemma 7** (Polynomial level [10])**.** *For any* PFOQ *program* P*,* $\text{level}_{\text{P}}(n) = O(n^{rk(\text{P})})$.

The set PFOQ of programs was shown to be sound and complete for the class FBQP, of function computable in quantum polynomial time [10].

▶ **Theorem 8** (PFOQ-Soundness and Completeness [10])**.** *For every function f in* FBQP*, there is a* PFOQ *program* P *that computes f with probability $\frac{2}{3}$ using at most a polynomial number of extra ancilla. Conversely, given a program* P *in* PFOQ *, if* P *computes $f : \{0,1\}^* \to \{0,1\}^*$, with probability $p \in (\frac{1}{2}, 1]$ then $f \in$* FBQP*.*

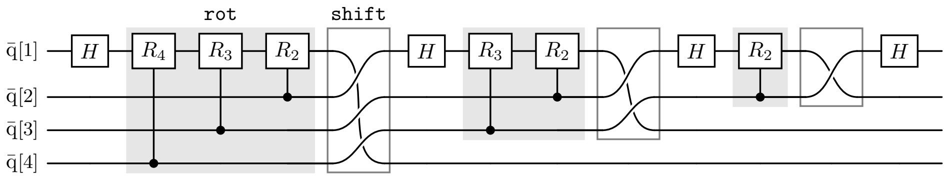


Figure 5 Circuit for the QFT as defined by the program in Example 6.

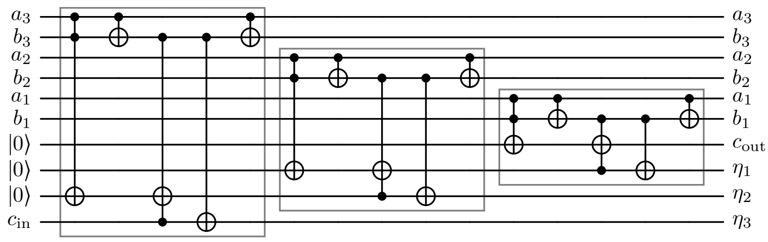


Figure 6 Quantum Full Adder circuit (Example 5) for input size 10.

Being a strict subset of PFOQ, BFOQ is trivially sound. Surprisingly, it is also complete.

▶ **Theorem 9** (BFOQ-Soundness and Completeness). *Theorem 8 holds for BFOQ.*

Proof. Soundness is trivial since BFOQ is contained in PFOQ, and completeness is given analogously to the proof of [10, Theorem 5], by noticing that all programs constructed in the proof are also contained in BFOQ. ◀

3 Circuit compilation

In this section, we introduce a compilation strategy for PFOQ programs that strictly improves on the compilation algorithm of [10]. We also show that, in the BFOQ fragment, circuit complexity scales in such a way that the cost of branching is the maximum cost of each branch, thereby avoiding branch sequentialization.

3.1 A new compilation algorithm

The compilation algorithm **compile**$^+$ takes as input a program P and a natural number n (the number of input qubits) and returns a circuit implementation of P for an input size of n qubits. **compile**$^+$ is defined by its subroutine **compr**$^+$ (Algorithm 1) in the following way:

$$\textbf{compile}^+(\mathrm{P}, n) \triangleq \textbf{compr}^+(\mathrm{P}, [1, \ldots, n], \cdot, \{\}),$$

where P is the program to be compiled, $[1, \ldots, n]$ is list of qubit pointers (initially all qubits), $\cdot$ is an empty control structure, and $\{\}$ an empty dictionary. A *control structure* is a partial function in $\mathbb{N} \to \{0, 1\}$ mapping qubit pointers to their control values in a quantum case. For $n \in \mathbb{N}$ and $k \in \{0, 1\}$, $cs[n := k]$ is the control structure obtained from cs by setting $cs(n) \triangleq k$. We denote by $dom(cs)$ the domain of the control structure. For a given $x \in \{0, 1\}^\star$, we say that state $|x\rangle$ *satisfies* cs if, $\forall n \in dom(cs)$, $cs(n) = k$ implies that $x_n = k$. The purpose of control structures in the algorithm is to preserve the information of each quantum branch and to allow for merging using ancillas.

Algorithm 1 ($\mathbf{compr}^+$)

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

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

else if $\mathrm{S} = \mathrm{s[i]}$ ***=** $\mathrm{U}^f(\mathrm{j})$; **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 $\mathrm{S} = \mathrm{S}_1\, \mathrm{S}_2$ **then**
  $C \leftarrow \mathbf{compr}^+(\mathrm{D} :: \mathrm{S}_1, l, cs, \mathrm{Anc}) \circ \mathbf{compr}^+(\mathrm{D} :: \mathrm{S}_2, l, cs, \mathrm{Anc})$ ▷ Composition

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

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

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

else if $\mathrm{S} =$ **call** proc[i](s); **and** $(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l' \neq [\,]$ **and** $(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n$ **then** ▷ Procedure call
  $a \leftarrow$ **new** *ancilla*()
  $\mathrm{Anc}[\mathrm{proc}', n, |l'|] \leftarrow (a, l')$;
  $C \leftarrow \mathbf{optimize}^+(\mathrm{D}, [(\cdot[a = 1], \mathrm{S}^{\mathrm{proc}}\{n/\mathrm{x}\})], \mathrm{proc}, l', \mathrm{Anc})$
end if
return C

The aim of subroutine $\mathbf{compr}^+$ is just to generate the quantum circuit corresponding to P on n qubits inductively on the statement of P. When the analyzed statement is a (possibly recursive) procedure call, $\mathbf{compr}^+$ calls the $\mathbf{optimize}^+$ subroutine (Algorithm 2) to perform an optimization of the generated quantum circuit. $\mathbf{optimize}^+$ has the same inputs as $\mathbf{compr}^+$ with the addition of a list of *controlled statements* l_{Cst} and the name proc of the procedure under analysis. A controlled statement is defined as a pair (cs, S) where cs is a control structure and S is a FOQ statement. This will allow us to generate the circuit implementation of S (which may contain multiple gates) while keeping track of the branch on which it is implemented.

The compilation algorithm $\mathbf{compile}^+$ is based on the process of merging procedure calls by reasoning about the orthogonality relations within the circuit. It is similar to the compilation algorithm **compile** of [10] based on the subroutines **compr** and **optimize**, with the differences highlighted in the code of Algorithms 1 and 2: $\mathbf{optimize}^+$ strictly improves on **optimize** (Theorem 12) by extending this analysis to procedures of different ranks.

3.2 Soundness and optimization

The correctness of $\mathbf{optimize}^+$ is a consequence of an orthogonality property between elements of the circuit being compiled that remains invariant throughout the compilation. In Algorithm 2, a recursive procedure proc is compiled by generating three separate circuits C_{L}, C_{M}, and C_{R}. The compilation process makes use of a list l_{Cst} of controlled statements which

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

$C_{\mathrm{L}} \leftarrow \mathbb{1}$; $C_{\mathrm{R}} \leftarrow \mathbb{1}$; $C_{\mathrm{M}} \leftarrow \mathbb{1}$; $\mathrm{P} \leftarrow \mathrm{D} :: \mathbf{skip}$;
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**
        $\mathrm{Anc}' \leftarrow \mathrm{Anc}.copy()$ /* create copy of ancilla dictionary */
        if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 1$ **then**
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_1)]$; $C_{\mathrm{M}} \leftarrow \mathbf{compr}^{+}(\mathrm{D} :: \mathrm{S}_2, l, cs, \mathrm{Anc}') \circ C_{\mathrm{M}}$
        else
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_2)]$; $C_{\mathrm{M}} \leftarrow \mathbf{compr}^{+}(\mathrm{D} :: \mathrm{S}_1, l, cs, \mathrm{Anc}') \circ C_{\mathrm{M}}$
        end if
    end if

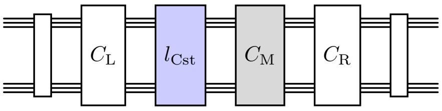
    if $\mathrm{S} = \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \mathrm{S}_{\mathbf{true}}\ \mathbf{else}\ \mathrm{S}_{\mathbf{false}}$ **and** $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} b$ **then**
        if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_b) = 1$ **then**
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs, \mathrm{S}_b)]$
        else
            $C_{\mathrm{M}} \leftarrow \mathbf{compr}^{+}(\mathrm{D} :: \mathrm{S}_b, l, cs, \mathrm{Anc}) \circ C_{\mathrm{M}}$
        end if
    end if

    if $\mathrm{S} = \mathbf{qcase}\ \mathrm{s[i]}\ \mathbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ **and** $(\mathrm{s[i]}, l) \Downarrow_{\mathbb{N}} n$ **then**
        if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 1$ **and** $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 1$ **then**
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0), (cs[n := 1], \mathrm{S}_1)]$
        else if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 0$ **then**
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0)]$;
            $C_{\mathrm{M}} \leftarrow \mathbf{compr}^{+}(\mathrm{D} :: \mathrm{S}_1, l, cs[n := 1], \mathrm{Anc}) \circ C_{\mathrm{M}}$
        else if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 0$ **then**
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(cs[n := 1], \mathrm{S}_1)]$;
            $C_{\mathrm{M}} \leftarrow \mathbf{compr}^{+}(\mathrm{D} :: \mathrm{S}_0, l, cs[n := 0], \mathrm{Anc}) \circ C_{\mathrm{M}}$
        end if
    end if

    if $\mathrm{S} = \mathbf{call}\ \mathrm{proc}'[\mathrm{i}](\mathrm{s})$ **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**
            Let $(a, l'') = \mathrm{Anc}[\mathrm{proc}', n, |l'|]$ **in**
            $e \leftarrow$ **new** $ancilla()$; /* compatible procedure already exists: merging case */
            $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ NOT(cs, e) \circ NOT(\cdot[e = 1], a) \circ SWAP(\cdot[e = 1], l', l'')$;
            $C_{\mathrm{R}} \leftarrow SWAP(\cdot[e = 1], l'', l') \circ NOT(\cdot[e = 1], a) \circ NOT(cs, e) \circ C_{\mathrm{R}}$
        else
            $a \leftarrow$ **new** $ancilla()$ /* no compatible procedure: create new ancilla */
            $\mathrm{Anc}[\mathrm{proc}', n, |l'|] \leftarrow (a, l')$;
            $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ NOT(cs, a)$; $C_{\mathrm{R}} \leftarrow NOT(cs, a) \circ C_{\mathrm{R}}$;
            $l_{\mathrm{Cst}} \leftarrow l_{\mathrm{Cst}}@[(\cdot[a = 1], \mathrm{S}^{\mathrm{proc}'}\{n/\mathrm{x}\})]$
        end if
    end if
end while
return $C_{\mathrm{L}} \circ C_{\mathrm{M}} \circ C_{\mathrm{R}}$

---

have not yet been compiled. Given a program P and an input procedure proc, the list l_{Cst} contains controlled statements (cs, S) such that $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}) = 1$, whereas C_{M} contains circuits for statements such that $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}) = 0$ but that are nonetheless orthogonal to those in l_{Cst}, for which merging will be possible. Circuits C_{L} and C_{R} contain circuits for statements for which $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}) = 0$ and that are not orthogonal to the elements of l_{Cst}. The circuit below pictures how the output circuit is generated from the circuits C_{L}, C_{M}, and C_{R} and the list l_{Cst}.



The steps of **optimize**$^{+}$ are depicted in Figure 7, where in each case we treat a controlled statement $(cs, \mathrm{S}) \in l_{\mathrm{Cst}}$. A gate placed inside the violet box denotes the new controlled statement that replaces (cs, S) in l_{Cst}. A gate placed inside a grey box indicates a circuit that is compiled and added to C_{M}. The notation is agnostic to the precise placement of these objects within l_{Cst} and C_{M}, making use of the orthogonality relation described in Lemma 11 which renders the choice inconsequential. Figures 7a, 7b, and 7c contain two circuits consisting of the different possible cases of compilation:

- In Figure 7a, we consider a sequence $\mathrm{S}_1\ \mathrm{S}_2$. This includes the case where S_2 is recursive (left) and the one where S_1 is recursive (right). Given the $\textsc{width}_{\geq 1}$ condition, there are no more cases.
- In Figure 7b, the case of classical control, we have the step where S_b contains a recursive call (above) and where it does not (below).
- For the case of quantum branching (Figure 7c), it is possible that only one of the two statements, say S_0, contains a recursive call (left) and the case where both do (right).
- Finally, we consider the case of a procedure call. Either there is already a compatible procedure and merging is performed (Figure 7d) or a new ancilla is used to anchor the procedure (Figure 7e).

We formalize orthogonality between controlled statements as follows.

▶ **Definition 10** (Orthogonality between control structures). *We say that two control structures are orthogonal, also denoted $cs \perp cs'$, if $\exists i \in \mathbb{N}$ such that $i \in dom(cs) \cap dom(cs')$ and where $cs(i) + cs'(i) = 1$.*

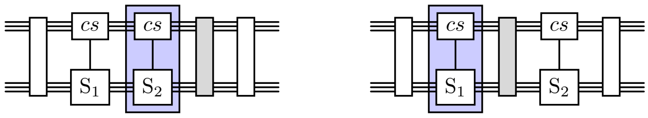
Hence, two control structures are orthogonal if there is no base state that satisfies them both. We now show that the steps of **optimize**$^{+}$ respect an orthogonality invariant.

▶ **Lemma 11** (Orthogonality invariant). *At each step of the subroutine* **optimize**$^{+}$*, the list l_{Cst} and the circuit C_M satisfy the following properties:*

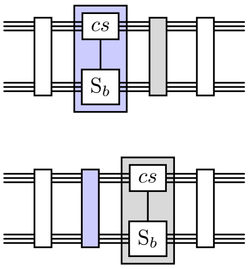
1. All controlled statements in l_{Cst} are mutually orthogonal:
$$\forall (cs, \mathrm{S}), (cs', \mathrm{S}') \in l_{\mathrm{Cst}} \text{ such that } (cs, \mathrm{S}) \neq (cs', \mathrm{S}'), \text{ we have that } cs \perp cs'.$$
2. Any controlled statement in l_{Cst} commutes with any element of C_{M}:
$$\forall (cs, \mathrm{S}) \in l_{\mathrm{Cst}}, \forall M(cs', [n]) \in C_M \text{ we have that } cs \perp cs'.$$

Proof. We start **optimize**$^{+}$ with a single procedure statement and an empty control structure, i.e. C_{M} is empty and $l_{\mathrm{Cst}} = \{(\cdot, \mathrm{S}^{\mathrm{proc}})\}$, in which case the lemma is clearly true.

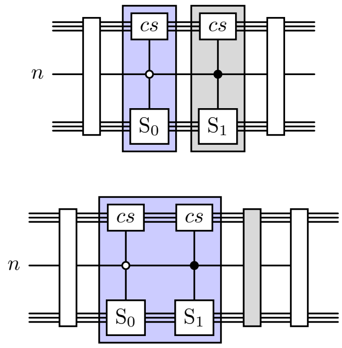
We now prove by induction that it is an invariant. Let $(cs, \mathrm{S}) \in l_{\mathrm{Cst}}$ be the controlled statement being treated. If $\mathrm{S} = \mathrm{S}_1\ \mathrm{S}_2$, let $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 1$ and $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_2) = 0$. Then, (cs, S)



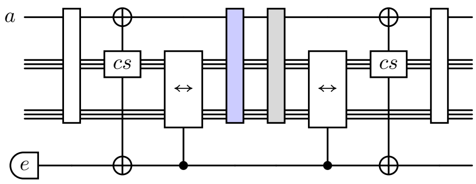
(a) $\mathrm{S} = \mathrm{S}_1\ \mathrm{S}_2$



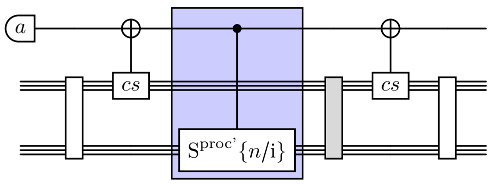
(b) $\mathrm{S} =$ **if** b **then** S_0 **else** S_1 and $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} b$.



(c) $\mathrm{S} =$ **qcase** q **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ and $(\mathrm{q}, l) \Downarrow_{\mathbb{N}} n$.



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



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

Figure 7 A step of the **optimize**$^+$ subroutine.

is replaced with (cs, S_1) in l_{Cst} and C_{M} is unchanged – therefore, the invariant property remains true. The case where $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1) = 0$ and $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_2) = 1$ is analogous.

If $\mathrm{S} =$ **if** b **then** S_0 **else** S_1, then consider the case $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} 0$, where if $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 1$ we have that (cs, S) is replaced with (cs, S_0) in l_{Cst} and C_{M} remains the same, and therefore the property remains true. If $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) = 0$ we have that (cs, S) is removed from l_{Cst} and $C_{\mathrm{M}} \leftarrow [\![(cs, \mathrm{S})]\!] \circ C_{\mathrm{M}}$. By the induction hypothesis all other cs_i in l_{Cst} are orthogonal to cs and therefore the property is conserved. The same can be shown for the case of $(\mathrm{b}, l) \Downarrow_{\mathbb{B}} 1$.

If $\mathrm{S} =$ **qcase** $\mathrm{s}[\mathrm{i}]$ **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ with $(\mathrm{s}[\mathrm{i}], l) \Downarrow_{\mathbb{N}} n$, notice first that $cs_i \perp cs_{i'}$ implies both $cs_i \perp cs_{i'}[n = 0]$ and $cs_i \perp cs_{i'}[n = 1]$, and that clearly $cs_{i'}[n = 0] \perp cs_{i'}[n = 1]$. Then, all cases preserve the condition.

If $\mathrm{S} =$ **call** $\mathrm{proc}'[\mathrm{i}](\mathrm{s})$ with $(\mathrm{s}, l) \Downarrow_{\mathcal{L}(\mathbb{N})} l' \neq [\,]$ and $(\mathrm{i}, l) \Downarrow_{\mathbb{Z}} n$, we consider two cases:

(i) A corresponding ancilla a already exists (merging), which by the constraints of PFOQ implies that $\cdot[a = 1] \perp cs_i$ for all cs_i in l_{Cst}. Therefore, by the induction hypothesis, adding cs to a preserves the orthogonality conditions.

(ii) No corresponding ancilla exists (anchoring), in which case the creation of the ancilla does not change the orthogonality between statements, as intended. ◀

We now show that algorithm **compile**$^+$ strictly improves on the asymptotic size of circuit, compared to the **compile** algorithm of [10].

▶ **Theorem 12.** *For any* PFOQ *program* P*,* $\#\mathbf{compile}^{+}(\mathrm{P}, n) = O(\#\mathbf{compile}(\mathrm{P}, n))$*. Furthermore, there exist programs for which* $\#\mathbf{compile}^{+}(\mathrm{P}, n) = o(\#\mathbf{compile}(\mathrm{P}, n))$*.*

Proof. The circuit size in both cases is asymptotically bounded by the number of ancillas created. Since we do more merging than before the result follows. Table 1 provides some examples to show the second claim. ◀

▶ **Theorem 13** (No branch sequentialization)**.** *For* $\mathrm{P} \in$ BFOQ *and* $n \in \mathbb{N}$*,* $\#\mathbf{compile}^{+}(\mathrm{P}, n) = O(\mathrm{level}_{\mathrm{P}}(n))$*.*

Proof. The theorem can be shown by structural induction on the program body, by checking that it is the case in each scenario that the circuit size scales with the level of the program. All cases are straightforward except the one of the quantum control case, which is proven at the end. The BASIC restrictions give us the following two properties during the compilation: (a) merging can be done in constant time, since there is no need for controlled-swap gates, and (b) a call to a recursive function only result in at most $O(n)$ calls to procedures of the same rank with unique ancillas.

We proceed by structural induction on the program body, considering that the statement is part of a procedure call for procedure proc.

- ($\mathrm{S} = \mathbf{skip};$ or $\mathrm{S} = \bar{\mathrm{q}}[i]$ *= U;) in this case we have that, $\mathrm{level}_{\mathrm{P}}(n) = 0$ and the the circuit is of constant size.
- ($\mathrm{S} = \mathrm{S}_1\ \mathrm{S}_2$) In this case, S_1 and S_2 are compiled in series (Figure 7a). The size of the circuit for S is then given by the sum of the sizes of the circuits of S_1 and S_2, and by definition the level of S is the sum of the levels.
- ($\mathrm{S} = \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \mathrm{S}_0\ \mathbf{else}\ \mathrm{S}_1$) Depending on the value of b the circuit for S either it corresponds to the circuit for S_0 or S_1. Therefore the size of the circuit is bounded by the maximum of between the two statements, as in the definition of level.
- (S = **call** proc[i](s);) This case also follows the definition of level since the circuit size is the one given inductively by the non-procedure-call operations (constant size) plus the circuit given by the procedure calls.

Notice that, for all statements besides the **qcase**, the size of the circuit follows the definition of level. We check that the number of ancillas created for S is bounded by the maximum number of ancillas for S_0 and S_1 separately. To show this, we proceed by induction on the rank r of the procedure.

The base case is given by (b), therefore we may consider $r > 1$. For the inductive case, we consider three possible scenarios:

- $w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_0) = w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_1) = 0$. Therefore, S_0 and S_1 contain only calls to procedures of rank strictly lower than r. This may only occur a constant number of times in the depth of a program, therefore we may simply consider the sum of the number of ancillas as a sufficient upper bound on the asymptotic number of ancillas for S.
- $w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_0) = w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_1) = 1$. In this case, S_0 and S_1 are of the same rank, r, and all their procedure calls may be merged. Therefore, the asymptotic number of such calls is bounded between the maximum between S_0 and S_1 (consider that, if there is no overlap between the ancillas needed, their number is still bounded linearly). Applying the IH on the procedures of rank $r-1$ we obtain the desired result.
- $w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_0) = 0$ and $w^{\mathrm{P}}_{\mathrm{proc}}(\mathrm{S}_1) = 1$. Therefore, S_0 contains calls to procedures of rank $r' < r$ whereas S_1 contains calls to procedures of rank r. The number of procedures of rank r' is bounded asymptotically by the maximum between those in S_0 and S_1, therefore we obtain our result. ◀

		Circuit complexity	
Problem	**Example**	[10]	**compile**$^+$
Full Adder	Example 5	$\Theta(n)$	$\Theta(n)$
Quantum Fourier Transform	Example 6	$\Theta(n^2)$	$\Theta(n^2)$
Palindromes	Example 14	$\Theta(n)$	$\Theta(n)$
Chained Substring	Example 15	$\Theta(n^3)$	$\Theta(n)$
Sum(r), $r \geq 1$	Example 18	$\Theta(n^r)$	$\Theta(n)$

Table 1 Circuit size complexity bounds given by the compilation strategy in [10] and **compile**$^+$ described in this work. For all of these problems, we give the corresponding programs in BFOQ.

4 Examples

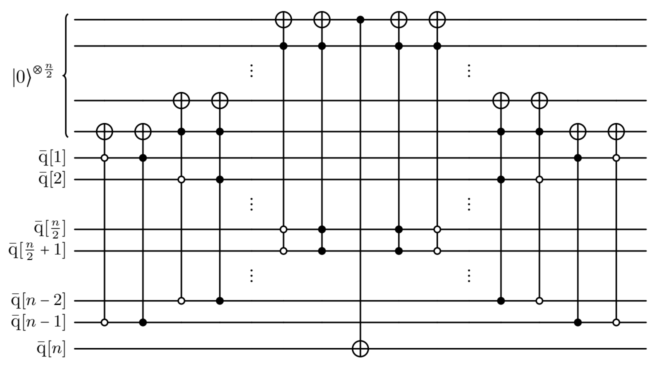
In this section, we provide several examples illustrating our results, including general examples on regular expressions. We show that any regular language can be decided by a BFOQ program whose compiled quantum circuit of linear size (Theorem 17). A benchmark, illustrating the difference between our compilation algorithm and the one in [10], is provided in Table 1.

▶ **Example 14** (Palindromes). Consider the following BFOQ program PALINDROME.

```
decl palindrome(q̄){
  if |q̄| > 2 then
    qcase q̄[1, |q̄| − 1] of {
      00 → call palindrome(q̄ ⊖ [1, −2]);
      01 → skip;
      10 → skip;
      11 → call palindrome(q̄ ⊖ [1, −2]);
      }
  else q̄[−1] *= NOT; }
 :: call palindrome(q̄);
```

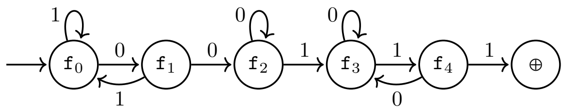
PALINDROME $\in$ WF since all recursive procedure calls decrease the input sorted set. Furthermore, at most one recursive call is done per branch, and therefore PALINDROME $\in$ WIDTH$_{\leq 1}$ and so the program is also in PFOQ. Further checking that all procedure calls in the program are either of the form $\bar{q}$ or $\bar{q} \ominus [1, |\bar{q}| - 1]$, we conclude that it is also in BFOQ. We are therefore in a position to apply Theorem 13.

Since $rk(\text{PALINDROME}) = rk_{\text{PALINDROME}}(\texttt{palindrome}) = 1$, by Lemma 7, we obtain the conclusion that $\#\textbf{optimize}^+(\mathrm{P}, n) = O(n)$, i.e., the compilation procedure generates a circuit of size linear on the input. Indeed, for PALINDROME, **compile**$^+$ generates the following circuit in the case where n is even:



The circuit makes use of $n/2$ ancillas that are reset to zero, only applying a NOT gate to $\bar{\mathrm{q}}[n]$ if $\bar{\mathrm{q}}[1,\dots,n-1]$ forms a palindrome.

▶ **Example 15** (Chained substring)**.** Let $s_0 = 001$ and $s_1 = 11$. Let $\mathcal{L}$ be the regular language defined by identifying strings containing an instance of s_0 followed eventually by an instance of s_1 in a word, i.e., the language defined by the regular expression $*s_0 * s_1*$. We can define a BFOQ program (Appendix C) that detects inputs in $\mathcal{L}$ using the following call graph:



The program has as body a procedure call **call** $\mathtt{f}_0(\bar{\mathrm{q}})$; and consists of 5 procedures $\mathtt{f}_i$ and a terminating procedure $\oplus$. An arrow $s \to^b t$ with $b \in \{0,1\}$ indicates a procedure call of the form **call** $t(\bar{\mathrm{q}} \ominus [1])$; appears in the body of procedure s done in a branch with $\bar{\mathrm{q}}[1]$ in state b.

The maximum rank of a procedure is 3 (for $\mathtt{f}_0$ and $\mathtt{f}_1$) and the circuit obtained by the technique in [10] gives a circuit of size $\Theta(n^3)$. On the other hand, the size of the circuit produced by $\mathbf{compile}^+$ grows linearly on the input size.

In the previous example, the bound obtained by $\mathbf{compile}^+$ was linear, which is the expected complexity in the case of detecting a regular language. It is straightforward to show that this is the case for any regular language, using the bound on the size of BFOQ circuits given in Theorem 13.

▶ **Definition 16.** *Let $A : \{0,1\}^* \to \{0,1\}$ be a decision problem. Given a* FOQ *program* P*, we say that* P *decides* A *if, for $\bar{x} \in \{0,1\}^*$ and $y \in \{0,1\}$, we have that $[\![\mathrm{P}]\!](|\bar{x}y\rangle) = |\bar{x}(y \oplus A(\bar{x}))\rangle$.*

▶ **Theorem 17** (Regular languages)**.** *For any regular language $\mathcal{L}$, there exists a* BFOQ *program* P *that decides if $\bar{x} \in \mathcal{L}$, for any $\bar{x} \in \{0,1\}^*$, such that $\#\mathbf{compile}^+(\mathrm{P}, n) \in O(n)$.*

Proof sketch. Since $\mathcal{L}$ is regular, there exists a deterministic finite automaton $\mathcal{D}$ that decides it. It is relatively simple to construct from $\mathcal{D}$ an BFOQ program that decides the language in the sense given in Definition 16. Since $\mathcal{D}$ is deterministic, the level of the corresponding program is bounded linearly. Using Theorem 13 we obtain the desired result. ◀

▶ **Example 18.** Let SUM_r be the decision problem of checking if an input bitstring contains precisely r 1s. This corresponds to identifying bitstrings in the regular expression $(0^*1)^r0^*$ and therefore, by Theorem 17, there exists a BFOQ program deciding SUM_r such that $\mathbf{compile}^+$ outputs a family of circuits of linear size.

5 Conclusions and Future Work

In this paper, we have delineated an expressive fragment, named BFOQ, of the first-order quantum programming language with quantum control of [10]. We have shown that BFOQ is sound and complete for polynomial time computation (Theorem 9) and that the branch sequentialization problem introduced by [19] is solved for BFOQ programs: the compiled circuit has size upper-bounded by the maximal complexity of program branches (Theorem 13). As a consequence, the compilation procedure generates circuits with a better size complexity than the compilation algorithm of [10] (Theorem 12). This result and the expressivity of BFOQ are illustrated by the Examples of Table 1. A future and challenging research direction includes the extension of this work to higher-order.

References

1. Samson Abramsky. No-cloning in categorical quantum mechanics. *Semantic Techniques in Quantum Computation*, pages 1–28, 2009.
2. John Baez. Quantum quandaries: a category-theoretic perspective. *The structural foundations of quantum gravity*, pages 240–265, 2006.
3. Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. `doi:10.1137/S0097539796300921`.
4. Hans J. Briegel, David E. Browne, Wolfgang Dür, Robert Raussendorf, and Maarten Van den Nest. Measurement-based quantum computation. *Nature Physics*, 5(1):19–26, 2009. `doi:10.1038/nphys1157`.
5. Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010. `doi:10.1016/j.tcs.2009.07.045`.
6. Vincent Danos and Elham Kashefi. Determinism in the one-way model. *Physical Review A*, 74(5):052310, 2006. `doi:10.1103/PhysRevA.74.052310`.
7. Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. Fast and effective techniques for T-count reduction via spider nest identities. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.TQC.2020.11`.
8. Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoît Valiron. Realizability in the unitary sphere. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.
9. Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 440–453, 2020.
10. Emmanuel Hainry, Romain Péchoux, and Mário Silva. A programming language characterizing quantum polynomial time. In *Foundations of Software Science and Computation Structures*, pages 156–175. Springer, 2023. `doi:10.1007/978-3-031-30829-1_8`.
11. Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: an introduction*. Oxford University Press, 2019.
12. Aleks Kissinger and John van de Wetering. Reducing the number of non-Clifford gates in quantum circuits. *Phys. Rev. A*, 102:022406, Aug 2020. `doi:10.1103/PhysRevA.102.022406`.
13. Emanuel Knill, Raymond Laflamme, and Gerard J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46–52, Jan 2001. `doi:10.1038/35051009`.
14. D. Maslov, G.W. Dueck, D.M. Miller, and C. Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, March 2008. `doi:10.1109/tcad.2007.911334`.
15. Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, May 2018. `doi:10.1038/s41534-018-0072-4`.
16. Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.
17. Tomoyuki Yamakami. A Schematic Definition of quantum Polynomial Time Computability. *J. Symb. Log.*, 85(4):1546–1587, 2020. `doi:10.1017/jsl.2020.45`.
18. Andrew Chi-Chih Yao. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 352–361, 1993. `doi:10.1109/SFCS.1993.366852`.
19. Charles Yuan and Michael Carbin. Tower: Data Structures in Quantum Superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, Oct 2022. `doi:10.1145/3563297`.

A Semantics of FOQ programs

In Section 2, we have defined $\mathcal{L}(\mathbb{N})$ as the set of lists of natural numbers $[n_1, \dots, n_k]$ (the empty list being denoted by $[]$), which are used to represent list of (unique) qubit pointers in the semantics.

Basic data types τ are interpreted as follows:

$$[\![\text{Integers}]\!] \triangleq \mathbb{Z} \qquad [\![\text{Booleans}]\!] \triangleq \mathbb{B} \qquad [\![\text{SortedSets}]\!] \triangleq \mathcal{L}(\mathbb{N})$$
$$[\![\text{Qubits}]\!] \triangleq \mathbb{N} \qquad [\![\text{Operators}]\!] \triangleq \tilde{\mathbb{C}}^{2\times 2}$$

Each basic operation $\text{op} \in \{+, -, >, \geq, =, \wedge, \vee, \neg\}$ of arity n, with $1 \leq n \leq 2$, has a type signature $\tau_1 \times \dots \times \tau_n \to \tau$ fixed by the program syntax. For example, the operation $+$ has signature $\text{Integers} \times \text{Integers} \to \text{Integers}$. A total function $[\![\text{op}]\!] \in [\![\tau_1]\!] \times \dots \times [\![\tau_n]\!] \to [\![\tau]\!]$ is associated to each basic operation op.

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

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

where $\tilde{\mathbb{C}}$ is the set of polynomial time computable complex numbers, i.e., complex numbers whose both real and imaginary part are in $\tilde{\mathbb{R}}$. Each of the above matrices is unitary, *i.e.*, the matrix M satisfies $M^* \circ M = M \circ M^* = I$, with M^* being the conjugate transpose of M and I being the identity matrix.

For each basic type τ, the reduction $\Downarrow_{[\![\tau]\!]}$ is a map in $\tau \times \mathcal{L}(\mathbb{N}) \to [\![\tau]\!]$. Intuitively, it maps an expression of type τ to its value in $[\![\tau]\!]$ for a given list l of pointers in memory. These reductions are defined in Figure 8, where e and d denote either an integer expression i or a Boolean expression b.

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

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

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

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

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

Figure 8 Semantics of expressions

Recall from Section 2 that the set of *configurations* over n qubits, denoted Conf_n, is defined by

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

where $\mathcal{P}(\mathbb{N})$ being the powerset over $\mathbb{N}$ and where $\top$ and $\bot$ are two special symbols for termination and error, respectively. Let $\diamond$ stand for a symbol in $\{\top, \bot\}$.

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

The program big-step semantics $\cdot \xrightarrow{\cdot} \cdot$, described in Figure 9, is defined as a relation in $\bigcup_{n\in\mathbb{N}} \mathrm{Conf}_n \times \mathbb{N} \times \mathrm{Conf}_n$.

$$\frac{}{(\mathbf{skip}, |\psi\rangle, \mathcal{S}, l) \xrightarrow{0} (\top, |\psi\rangle, \mathcal{S}, l)}\ (\mathrm{Skip}) \qquad \frac{(\mathrm{s[i]}, l) \Downarrow_{\mathbb{N}} n \notin \mathcal{S}}{(\mathrm{s[i]} \mathrel{{*}{=}} \mathrm{U}^f(\mathrm{j});, |\psi\rangle, \mathcal{S}, l) \xrightarrow{0} (\bot, |\psi\rangle, \mathcal{S}, l)}\ (\mathrm{Asg}_\bot)$$

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

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

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

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

$$\frac{(\mathrm{s[i]}, l) \Downarrow_{\mathbb{N}} n \in \mathcal{S} \qquad (\mathrm{S}_k, |\psi\rangle, \mathcal{S}\backslash\{n\}, l) \xrightarrow{m_k} (\top, |\psi_k\rangle, \mathcal{S}\backslash\{n\}, l)}{(\mathbf{qcase}\ \mathrm{s[i]}\ \mathbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}, |\psi\rangle, \mathcal{S}, l) \xrightarrow{\max_k m_k} (\top, \sum_k |k\rangle_n \langle k|_n |\psi_k\rangle, \mathcal{S}, l)}\ (\mathrm{Case}_\top)$$

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

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

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

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

Figure 9 Semantics of statements

B Proofs

In this section, we provide the full proof of Theorem 17. Towards that end, we first define a notion of call-graph.

▶ **Definition 19** (Call graph). *A* call graph $\mathcal{G}$ *is a triple* (proc, V, E) *where*

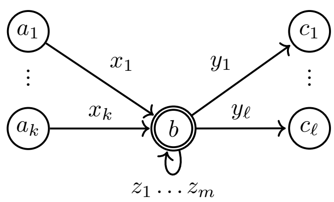
- $\mathrm{proc} \in V$ *is the entry node;*
- $V \subseteq \mathrm{Procedures}$ *is a set of nodes containing a special procedure* $\oplus$*;*
- $E \subseteq V \times L \times V$ *is a set of labeled directed edges, where labels correspond to combinations of quantum and classical conditionals.*

Procedure $\oplus$ *only applies a NOT gate to the last qubit in the input and terminates. Labels L are defined as follows: values* $\{0,1\}$ *denote a quantum if statement on the first qubit, and* $|\bar{\mathrm{q}}| = n$ *or* $|\bar{\mathrm{q}}| > n$ *denotes a boolean condition on the size of the input.*

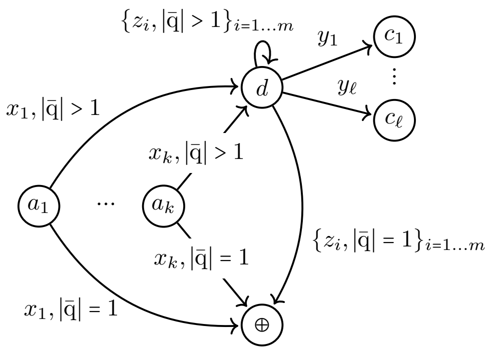
▶ **Theorem 17** (Regular languages). *For any regular language* $\mathcal{L}$*, there exists a* BFOQ *program* P *that decides if* $\bar{x} \in \mathcal{L}$*, for any* $\bar{x} \in \{0,1\}^*$*, such that* $\#\mathbf{compile}^+(\mathrm{P}, n) \in O(n)$*.*

Proof. Let $\mathcal{D}$ be a deterministic finite automaton deciding $\mathcal{L}$. We will construct the PFOQ program P by using $\mathcal{D}$ to define the call graph for P. The subtlety in the transformation is in the difference between the acceptance condition in $\mathcal{D}$ (i.e., termination in an accept state) and the *acceptance nodes* of the call graph, referring here to the $\oplus$ nodes. For a base state $|\bar{x}y\rangle$, the program P outputs $|\bar{x}(\neg y)\rangle$ iff it *ever* reaches a $\oplus$ node, at which point P terminates.

The call graph is then defined as follows. The call graph contains a node for each state of $\mathcal{C}$, with the same transitions, except for those that constitute incoming our outgoing edges of an accept state, i.e., edges x_i, y_i, z_i in the following diagram:



These edges are encoded in the call graph as follows:



with an extra procedure d for each accept state and using edges with classical conditions to handle the acceptance condition of the program. P is then defined as the program given by the call graph where the procedures consist only of the procedure calls defined in the graph. For a set of conditions c_i, with $i = 1 \dots m$, we denote by $\{c_i\}_{i=1\dots m}$ a set of m edges each labelled with a conditions c_i.

Since each procedure performs at most one procedure call per branch (recursive or otherwise) its level will be linear on the input size. ◀

C Regular languages (Example 15)

The following is the program defined by the call graph given in Example 15.

$$
\begin{array}{l}
\mathbf{decl}\ \mathtt{f}_0(\bar{q})\{\\
\quad \mathbf{qcase}\ \bar{q}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathtt{f}_1(\bar{q} \ominus [1]),\ 1 \to \mathbf{call}\ \mathtt{f}_0(\bar{q} \ominus [1])\},\\
\\
\mathbf{decl}\ \mathtt{f}_1(\bar{q})\{\\
\quad \mathbf{qcase}\ \bar{q}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathtt{f}_2(\bar{q} \ominus [1]),\ 1 \to \mathbf{call}\ \mathtt{f}_0(\bar{q} \ominus [1])\},\\
\\
\mathbf{decl}\ \mathtt{f}_2(\bar{q})\{\\
\quad \mathbf{qcase}\ \bar{q}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathtt{f}_2(\bar{q} \ominus [1]),\ 1 \to \mathbf{call}\ \mathtt{f}_3(\bar{q} \ominus [1])\},\\
\\
\mathbf{decl}\ \mathtt{f}_3(\bar{q})\{\\
\quad \mathbf{qcase}\ \bar{q}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathtt{f}_3(\bar{q} \ominus [1]),\ 1 \to \mathbf{call}\ \mathtt{f}_4(\bar{q} \ominus [1])\},\\
\\
\mathbf{decl}\ \mathtt{f}_4(\bar{q})\{\\
\quad \mathbf{qcase}\ \bar{q}[1]\ \mathbf{of}\ \{0 \to \mathbf{call}\ \mathtt{f}_3(\bar{q} \ominus [1]),\ 1 \to \mathbf{call}\ \oplus(\bar{q} \ominus [1])\},\\
\\
\mathbf{decl}\ \oplus(\bar{q})\{\\
\quad \bar{q}[-1] \mathrel{*}= \mathrm{NOT};\}\\
\\
::\ \mathbf{call}\ \mathtt{f}_0(\bar{q});
\end{array}
$$

It is straightforward to verify that all conditions of BFOQ are met and that the level of the program is linearly bounded.