A Programming Language Characterizing Quantum Polynomial Time

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

June 14, 2024

Abstract

We introduce a first-order quantum programming language, named FOQ, whose terminating programs are reversible. We restrict FOQ to a strict and tractable subset, named PFOQ, of terminating programs with bounded width, that provides a first programming-language-based characterization of the quantum complexity class FBQP. We finally present a tractable semantics-preserving algorithm compiling a PFOQ program to a quantum circuit of size polynomial in the number of input qubits. Published in FoSSaCS 2023 and available at `https://doi.org/10.1007/978-3-031-30829-1_8`.

1 Introduction

The promise of a quantum computer requires the development of different layers of hardware and software, together referred to as the *quantum stack*. Programming languages constitute one of the highest layers of this stack, providing the tools for reasoning about quantum programs across different models.

One interesting application of programming languages is the ability to statically (i.e. without running the program) infer properties such as validity or time complexity. This is related to the field of *implicit computational complexity* (ICC) [1, 3, 5]. In ICC, complexity classes are characterized in different computational paradigms, using syntactical restrictions that can be automatically checked.

In the quantum scenario, the set of efficiently computable functions is the complexity class FBQP (functions with bounded-error in quantum polynomial time). FBQP is defined as the set of functions that can be approximated by a quantum Turing machine running in polynomial time [2], but it has also been shown to be equivalent to the set of functions approximated by uniform polysized quantum circuits [8].

Contribution.

- We introduce a quantum programming language, named FOQ, that includes first-order recursive procedures.
- After showing that terminating FOQ programs are reversible, we restrict programs to a strict subset, named PFOQ, for *polynomial-time* FOQ. Restrictions of PFOQ programs are tractable (i.e., can be decided in polynomial time in the program size), and ensure that programs terminate in polynomial time on any input.
- We show that the class of functions computed by PFOQ programs is *sound* and *complete* for the quantum complexity class FBQP. Hence the language PFOQ is, to our knowledge, the first programming language characterizing quantum polynomial time functions.
- We also describe a polynomial-time deterministic algorithm that takes in a PFOQ program P and an integer n and outputs a quantum circuit of size polynomial in n that implements P on an input of n qubits. We show how our compilation technique avoids the exponential blowup from quantum branching identified in [9].

Related work. We note two results in the study of characterizations of quantum polytime classes: [4], providing a characterization of BQP based on a quantum lambda-calculus, and [7], characterizing FBQP via a function algebra.

Our work is greatly inspired by [7]. However, we claim that PFOQ is more expressive, as we show that any function in [7] can be simulated by a PFOQ program, with the syntactical restrictions of PFOQ being much more permissive than the *multi-qubit recursion* scheme described in [7]. As an example, the quantum Fourier transform (Figure 2) requires an additional initial quantum function to be defined in [7].

(Integers)	i	$\triangleq$	n | x | i + n | i − n | |s|, with $n \in \mathbb{N}$
(Booleans)	b	$\triangleq$	i > i | i ⩾ i | i = i | b ∧ b | b ∨ b | ¬b
(Sorted Sets)	s	$\triangleq$	nil | q̄ | s ⊖ [i]
(Qubits)	q	$\triangleq$	s[i]
(Operators)	$\mathrm{U}^f(\mathrm{i})$	$\triangleq$	NOT | $\mathrm{R}_Y^f(\mathrm{i})$ | $\mathrm{Ph}^f(\mathrm{i})$, with $f \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$
(Statements)	S	$\triangleq$	**skip**; | q *= $\mathrm{U}^f(\mathrm{i})$; | S S | **if** b **then** S **else** S | **qcase** q **of** {0 → S, 1 → S} | **call** proc[i](s);
(Procedure declarations)	D	$\triangleq$	ε | **decl** proc[x](q̄){S}, D
(Programs)	P(q̄)	$\triangleq$	D :: S

Figure 1: Syntax of FOQ programs.

2 Syntax and example

A program contains a set D of procedure declarations followed by a program statement S which includes the instructions of the program, possibly including calls to procedures in D. These procedures are allowed to be recursive, for example, by including calls to themselves, and they may include an integer input x that is used in the procedure body.

The syntax of the language (see Figure 1) does not include measurements, as these are assumed to be performed only at the end of the program, and therefore the running time only depends on the input size. We can perform certain operations depending on the state of a qubit (**qcase**) or on a classical boolean conditions (**if**), which may depend on some integer values or the size of the input.

As basic operators we consider the NOT gate, the phase-shift gate $PHASE_\theta$ and the y-rotation gate ROT_θ, for polytime approximable angles $\theta \in [0, 2\pi)$, which form a universal gate set that can be efficiently implemented.

Quantum variables are handled using *sorted sets*, which simply means that the elements of q̄ are ordered and point to different qubits, and any subset passed on to a function call does not contain repeated elements. Operations with sorted subsets include accessing the i-th entry, q̄[i], and the set without this entry, denoted q̄ ⊖ [i]. We can also access the length |q̄| of the sorted set in order to branch classically.

Quantum Fourier Transform

We illustrate the language with the example of the QFT, used in Shor's algorithm [6], with code depicted in Figure 2. We use quantum-controlled versions of the Hadamard gate $H = NOT \cdot ROT_{\pi/4}$ and the rotation gate $R_m = PHASE_{\pi/2^{m-1}}$. These two gates are applied in a doubly recursive pattern by procedures rec and rot, followed by a reordering of the qubits which is also done recursively (inv). This program can then be compiled into the recognizable circuit in Figure 3.

```
decl rec(q̄){                    decl inv(q̄){
  q̄[1] *= H;                       if |q̄| > 1 then
  call rot[2](q̄);                    SWAP(q̄[1], q̄[|q̄|]);
  call rec(q̄ ⊖ [1]); },              call inv(q̄ ⊖ [1, |q̄|]);
                                   else skip; },

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

call rec(q̄); call inv(q̄);
```

Figure 2: A PFOQ program for QFT.

A property of the recursive procedures is that the available resources strictly decrease with each recursive call. This property, which we will describe more precisely in Section 3, ensures that the program terminates, and the fact that there is only one recursive call per procedure guarantees polytime termination.

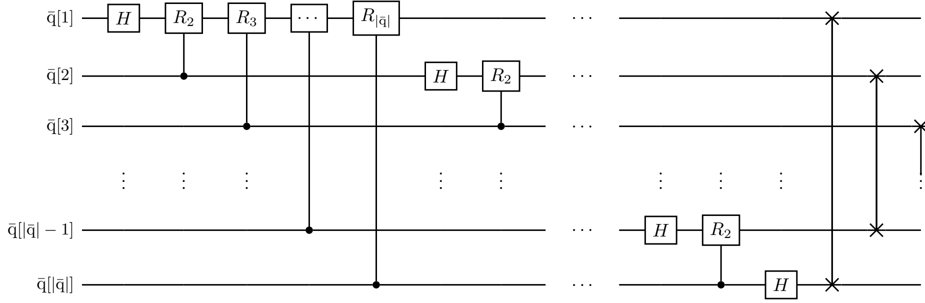

Figure 3: Circuit for the quantum Fourier transform.

3 Restrictions for polytime

To restrict FOQ to its polytime terminating fragment, we define two criteria: one for ensuring termination and another to prevent an exponential runtime from the doubling of procedure calls.

We prevent infinite nesting of procedures by requiring that each recursive call reduce the amount of available qubits. The relation $\text{proc}_i \sim \text{proc}_j$ indicates that there is a call to proc_j at some point in the running of proc_i, and vice-versa, i.e. they are mutually recursive. We then define the set WF of *well founded* programs as the set of programs where every mutually recursive procedure call strictly decreases the number of qubits.

To prevent an exponential number of procedure calls, we impose the condition that only one mutually recursive call may be included in each procedure. We define the width of a procedure as the number of (mutually) recursive procedure calls that appear in its body, counting the maximum of each branch. Therefore, $\text{width}_{\text{P}}(\text{proc}) \triangleq w_{\text{P}}^{\text{proc}}(\text{S}^{\text{proc}})$, where $w_{\text{P}}^{\text{proc}}$ is defined as follows:

$$w_{\text{P}}^{\text{proc}}(\textbf{skip};) \triangleq 0,$$
$$w_{\text{P}}^{\text{proc}}(\text{q} \mathrel{*}= \text{U}^f(\text{i});) \triangleq 0,$$
$$w_{\text{P}}^{\text{proc}}(\text{S}_1\ \text{S}_2) \triangleq w_{\text{P}}^{\text{proc}}(\text{S}_1) + w_{\text{P}}^{\text{proc}}(\text{S}_2),$$
$$w_{\text{P}}^{\text{proc}}(\textbf{call}\ \text{proc}'[\text{i}](\text{s});) \triangleq \begin{cases} 1 & \text{if proc} \sim_{\text{P}} \text{proc}', \\ 0 & \text{otherwise.} \end{cases}$$

and, in the case of branching, we have that

$$w_{\text{P}}^{\text{proc}}(\textbf{if}\ \text{b}\ \textbf{then}\ \text{S}_0\ \textbf{else}\ \text{S}_1),\ \text{and}$$
$$w_{\text{P}}^{\text{proc}}(\textbf{qcase}\ \text{q}\ \textbf{of}\ \{0 \to \text{S}_0, 1 \to \text{S}_1\})$$

are both defined as $\max(w_{\text{P}}^{\text{proc}}(\text{S}_0), w_{\text{P}}^{\text{proc}}(\text{S}_1))$. Then, the set PFOQ (polytime FOQ) is the set of WF programs that satisfy the condition:

$$\forall \text{proc} \in \text{P}, \text{width}_{\text{P}}(\text{proc}) \leqslant 1.$$

A WF program always terminates, and a program in PFOQ will terminate in polynomial time. Any terminating program admits an inverse. In the following, we denote by $[\![\text{P}]\!]$ the semantics of program P.

Theorem 1. *All terminating* FOQ *programs are reversible: if* P *terminates, we can construct a program* P^{-1} *such that* $[\![\text{P}^{-1}]\!] \circ [\![\text{P}]\!] = Id$.

The following theorems state more precisely the relation PFOQ and FBQP.

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

Given a polynomial Q and quantum state $|x\rangle$ let $\phi_Q(|x\rangle)$ be the function that returns the state $|x\rangle$ preceded by some extra writing space, linear in Q, used to perform internal computations.

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

Our completeness proof is done by showing that PFOQ is expressive enough to implement the function algebra in [7]. In particular, we allow for a more general recursion scheme.

Our soundness proof is done by showing that, for any PFOQ program P, there exists a polytime quantum Turing machine that implements $[\![\text{P}]\!]$. We also give a second (and more practical) soundness proof by describing a polynomial time algorithm that, given a PFOQ program P and a natural number n, computes the circuit implementing $[\![\text{P}]\!]$ on an input of n qubits.

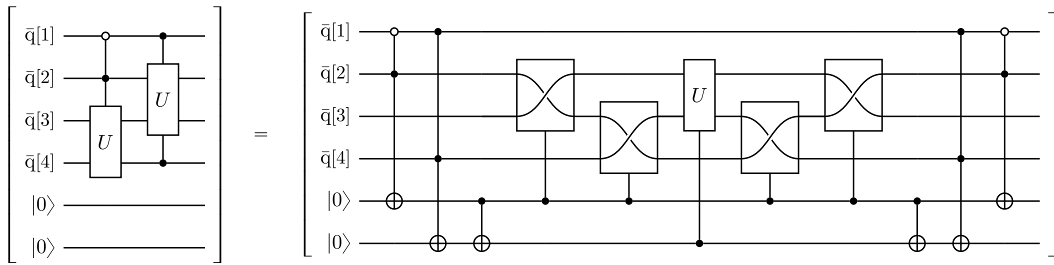

Figure 4: Example of circuit optimization.

4 Compilation strategy

Given Yao's equivalence result between uniform families of circuits and polytime quantum Turing machines [8], the existence of uniform families of circuits implementing PFOQ programs is not surprising. However, any practical implementation requires a direct compilation strategy, without an intermediate quantum Turing machine representation.

Surprisingly, it is not obvious how to directly generate the circuit for a program while preserving its polynomial time complexity. If we consider the statement for quantum branching (**qcase**), while the time complexity in the QTM model is the maximum of each branch, in the circuit model we are required to perform them in sequence. This disanalogy between branching in QTMs and circuits, more recently coined as "branch sequentialization" [9], can generate an exponential blowup in the size of the final circuit.

In this work, we describe a merging technique, exemplified in Figure 4, that uses ancillas and controlled swap operations to combine equal procedure calls that occur in different branches. This allows us to avoid the exponential blowup from branch sequentialization in PFOQ programs.

Acknowledgements

Tha authors would like to thank the anonymous reviewers for their feedback on this extended abstract.

This work is supported by the Inria associate team TC(Pro)^{3}v, the Plan France 2030 through the PEPR integrated project EPiQ ANR-22-PETQ-0007, and the HQI initiative ANR-22-PNCQ-0002.

References

[1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *computational complexity*, 2(2):97–110, Jun 1992.

[2] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[3] Ugo Dal Lago. A short introduction to implicit computational complexity. In *ESSLLI 2010*, pages 89–109, 2011.

[4] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[5] Romain Péchoux. Implicit computational complexity: past and future. Mémoire d'habilitation à diriger des recherches, 2020. Université de Lorraine.

[6] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[7] Tomoyuki Yamakami. A schematic definition of quantum polynomial time computability. *J. Symb. Log.*, 85(4):1546–1587, 2020.

[8] Andrew Chi-Chih Yao. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 352–361, 1993.

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