Quantum Programming in Polylogarithmic Time

Florent Ferrari
École Normale Supérieure de Lyon, France

Emmanuel Hainry
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Romain Péchoux
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Mário Silva
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract

Polylogarithmic time delineates a relevant notion of feasibility on several classical computational models such as Boolean circuits or parallel random access machines. As far as the quantum paradigm is concerned, this notion yields the complexity class FBQPOLYLOG of functions approximable in polylogarithmic time with a quantum random access Turing machine. We introduce a quantum programming language with first-order recursive procedures, which provides the first programming language-based characterization of FBQPOLYLOG. Each program computes a function in FBQPOLYLOG (soundness) and, conversely, each function of this complexity class is computed by a program (completeness). We also provide a compilation strategy from programs to uniform families of quantum circuits of polylogarithmic depth and polynomial size, whose set of computed functions is known as QNC, and recover the well-known separation result FBQPOLYLOG $\subsetneq$ QNC.

2012 ACM Subject Classification Theory of computation → Quantum complexity theory

Keywords and phrases Quantum programming languages, Polylogarithmic time, Quantum circuits, Implicit computational complexity

Digital Object Identifier 10.4230/LIPIcs.MFCS.2025.47

Funding This work is supported by the the Plan France 2030 through the PEPR integrated project EPiQ ANR-22-PETQ-0007 and the HQI platform ANR-22-PNCQ-0002; and by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520), project NEASQC (Grant Agreement 951821), and project HPCQS (Grant Agreement 101018180).

1 Introduction

1.1 Motivation

Quantum computing is a field of research, which is drawing a great amount of interest as it leverages quantum superposition and interference to obtain computational advantage. The development of quantum programming languages is thus a key issue, which raises major technical and conceptual challenges to ensure their physicality. In order to check that quantum programs can be compiled and executed on a quantum computer, one has to design restrictions and constraints implying that quantum programs do not break the laws of quantum mechanics, for example, no-cloning of data and unitarity of operators. In addition, there is a need to tame their complexity in order to ensure their feasibility. By feasibility, we mean that quantum executions do not use too many ancillary qubits and run in tractable time.

Taking inspiration from the classical world, this kind of issue has lead to the definition and study of several quantum polynomial time classes. One of the most striking examples of such classes is BQP, the quantum analog of the class of bounded-error probabilistic polynomial time

problems BPP. By Yao's theorem [23], BQP corresponds exactly to what can be computed by uniform families of quantum circuits of polynomial size. This class, as well as its extension to functions, FBQP, have been characterized through various means, including function algebras [19] and first-order programs [12].

A natural question is then to study subpolynomial complexity classes. As for classical programs, it allows to express notions of parallel complexity, which is highly relevant in the quantum setting where program superpositions can be viewed as a kind of parallelism with interferences. Parallelization can be used to reduce the maximum number of operations performed in total on each qubit. This is particularly challenging as qubit fidelity, i.e., the ability of qubits to align with their intended states through time and unitary operations, is a bottleneck of quantum computation [14]. While polynomial time algorithms performed in sequence are useful in the *fewer qubits, higher fidelity* setting [12], parallelized computation becomes more interesting in the case of *more qubits, lower fidelity* [15], as the total number of operations on individual qubits, and their necessary coherence time, scale more slowly. Moreover, separation results between those small complexity classes could lead to a proof of the quantum advantage, as constant depth quantum circuits have been shown to be strictly more powerful than their classical counterparts [5].

In the literature, polylogarithmic (polylog) time has been introduced and studied on Quantum Random-Access Turing Machine (QRATM) [10]. As in the classical case, this definition uses random-access machines, as opposed to standard quantum Turing machines, because of the sub-linearity of time: although the machine cannot read its entire input, it can access any input bit or qubit. On quantum models, polylog time corresponds to problems that a QRATM can solve in a polylog number of steps, leading to the definition of the complexity class FBQPOLYLOG [20, 21] of functions computable with bounded-error in quantum polylog time.

A main open problem is to design programming languages characterizing such a polylog class abstracting from the low-level considerations (machines, uniformity conditions, etc.) [22].

1.2 Contributions

This paper makes a first step towards solving this problem. Towards that end, we introduce a quantum programming language with first-order recursive procedures, named PLP for PolyLog Programs (Figure 1), on which we obtain the following results:

- PLP programs are terminating (Theorem 7) and reversible (Theorem 8).
- PLP is sound for FBQPOLYLOG (Theorem 10), i.e., each PLP program computes a function in FBQPOLYLOG. Soundness relies on the use of a bounded recursion scheme for procedures to enforce the required polylog time properties, as illustrated by the binary search and square-log examples of Figure 4.
- PLP is complete for FBQPOLYLOG (Theorem 10), i.e., each function of this complexity class is computed by a PLP program. Completeness is shown by a direct encoding of polylog QRATM in PLP.
- PLP is also sound but not complete for QNC. For soundness, we outline a compilation algorithm that from a PLP program and its input size outputs a quantum circuit of polylog depth and polynomial size, i.e., circuits computing functions in QNC (Theorem 13). There is an implementation of this algorithm available at `https://gitlab.inria.fr/mmachado/pfoq-compiler`, whose compiler works for programs even beyond the PLP fragment, namely non-polylog programs. Completeness does not hold (Theorem 19) as it is well-known that FBQPOLYLOG is strictly included in QNC.

(Integers)	i	$\triangleq$	$x \mid k \mid i \pm k \mid i/2 \mid \lvert l \rvert$
(Booleans)	b	$\triangleq$	$i \geq i \mid \ldots \mid b \wedge b \mid \ldots$
(Qubit lists)	l	$\triangleq$	$\bar{q} \mid l \ominus [i] \mid l^{\boxminus} \mid l^{\boxplus}$
(Qubits)	q	$\triangleq$	$l[i]$
(Statements)	S	$\triangleq$	**skip**; $\mid q$ *= $U^g(i); \mid S\ S \mid$ **if** b **then** {S} **else** {S} $\mid$ **qcase** q **of** $\{0 \to S, 1 \to S\}$ $\mid$ **call** $\text{proc}(l_1, \ldots, l_n)$;
(Procedure decl.)	D	$\triangleq$	$\varepsilon \mid$ **decl** $\text{proc}(\bar{q}_1, \ldots, \bar{q}_n)\{S\}$, D
(Programs)	P	$\triangleq$	D :: S

Figure 1 Syntax of programs.

1.3 Related Work

Different characterizations of quantum complexity classes have been obtained for polynomial time using, non-exhaustively, lambda-calculus [8], function algebra [19], and a first-order programming language [12]. A characterization of BQPOLYLOG based on a function algebra has been provided in [20, 21], where a *fast quantum recursion scheme* is used to ensure that programs terminate in polylogarithmic time. Our work employs a similar bounded recursion scheme, using a simple divide-and-conquer strategy on qubits. This characterization can be seen as simpler and more natural approach since an imperative first-order programming language is more accessible to the typical programmer. Similar divide-and-conquer strategies have been explored in quantum computation not only for the purpose of finding quantum advantage [5, 6] but also to leverage classically-inspired techniques in the quantum scenario, where reversibility and unitarity must be satisfied [18, 17]. Our work is mostly focused on this second aspect, where the objective is to balance the expressivity of a quantum programming language with the statically-verifiable properties of its programs, namely unitarity, time complexity, as well as circuit size and depth.

2 First-Order Quantum Programs

2.1 Syntax

The considered language is a quantum programming language with first-order recursive procedures whose syntax is provided in Figure 1. There are four basic types τ for expressions:

1. *Integer* expressions are variables x, constant $k \in \mathbb{N}$, arithmetic operations like $i \pm k$ or $i/2$,[1] as well as the size $\lvert l \rvert$ of a list of qubits l.
2. *Boolean* expressions are defined in a standard way.
3. *Qubit lists* are lists of unique (i.e., non-duplicable) qubit pointers. A qubit list expression l is either a variable $\bar{q}$, the first (respectively second) half $l^{\boxminus}$ (resp. $l^{\boxplus}$) of the qubit list l, or a list $l \ominus [i]$ where the i-th element of l has been removed. We will also use some syntactic sugar for removing multiple elements of a list with $l \ominus [i_1, \ldots, i_k]$.
4. *Qubit* expressions are of the shape l[i], which denotes the i-th qubit in l. We also define syntactic sugar for pointing to the n-th *last* qubit in a list, by defining for any $n \geq 1$, $\bar{q}[-n] \triangleq \bar{q}[|\bar{q}| - n + 1]$.

[1] The semantics of /2 will be defined as the ceiling of the result, hence it preserves the set of integers.

Throughout the paper, e, d, ... will denote arbitrary expressions of any type. Given a syntactic object t, let $Var(t)$ be the set of qubit variables used in t, e.g., $Var(\bar{\mathrm{q}} \ominus [2,3]) = \{\bar{\mathrm{q}}\}$ is the set of qubit variables in the expression $\bar{\mathrm{q}} \ominus [2,3]$.

A program $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$ is defined in Figure 1 as a list of (possibly recursive) procedure declarations D, followed by a program statement S. We assume that $Var(\mathrm{S}) \subseteq Var(\mathrm{P})$ holds. In what follows, it will be convenient to order the set $Var(\mathrm{P}) = \{\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_m\}$ by $\bar{\mathrm{q}}_1 < \ldots < \bar{\mathrm{q}}_m$ to fix a precise memory representation of quantum states.

Let Procedures be an enumerable set of procedure names proc. We write $\mathrm{proc} \in \mathrm{P}$ to denote that proc appears in P. Each procedure of name $\mathrm{proc} \in \mathrm{P}$ is defined by a unique procedure declaration **decl** $\mathrm{proc}(\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_n)\{\mathrm{S}\} \in \mathrm{D}$, which takes the lists of qubits $\bar{\mathrm{q}}_i$ as parameters. It always holds that $Var(\mathrm{S}) \subseteq \{\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_n\}$. We will sometimes write $\mathrm{S}^{\mathrm{proc}}$ to explicitly state that S is the statement of proc.

Statements include the no-op instruction, (single-qubit) unitary application, sequences, conditional, quantum case, and procedure calls. For sake of universality [4], in a unitary application q `*=` $\mathrm{U}^g(\mathrm{i})$;, the unitary transformation $\mathrm{U}^g(\mathrm{i}) \in \mathbb{K}^{2\times 2}$ can take an integer i and a function $g \in \mathbb{Z} \to [0, 2\pi)$ as optional arguments[2], and we omit them when they are of no use. As we shall see in next section, unitary transformations will be restricted to phase gate, rotation gates, and NOT gate.

The quantum conditional **qcase** q **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ allows branching by executing statements S_0 and S_1 in superposition according to the state of qubit q. When we want to treat cases on multiple qubits, we will sometimes simplify the nested qcases, for example **qcase** $\mathrm{q}_1, \mathrm{q}_2$ **of** $\{00 \to \mathrm{S}_{00}, 01 \to \mathrm{S}_{01}, 10 \to \mathrm{S}_{10}, 11 \to \mathrm{S}_{11}\}$ is a shorthand notation for

```
qcase q1 of {
    0 → qcase q2 of {
        0 → S00,
        1 → S01
    },
    1 → qcase q2 of {
        0 → S10,
        1 → S11
    }
}
```

In each procedure call **call** $\mathrm{proc}(\mathrm{l}_1, \ldots, \mathrm{l}_n)$;, the no-cloning theorem of quantum mechanics imposes the restriction that $\forall i \neq j,\ Var(\mathrm{l}_i) \neq Var(\mathrm{l}_j)$.

The syntax of Figure 1 can be used to define typical quantum computing primitives such as controlled-NOT, swap, as well as Toffoli gates, as syntactic sugar:

$$\begin{aligned}
\mathrm{CNOT}(\mathrm{q}_1, \mathrm{q}_2) &\triangleq \textbf{qcase } \mathrm{q}_1 \textbf{ of } \{0 \to \textbf{ skip};, 1 \to \mathrm{q}_2 \texttt{ *= } \mathrm{NOT};\} \\
\mathrm{SWAP}(\mathrm{q}_1, \mathrm{q}_2) &\triangleq \mathrm{CNOT}(\mathrm{q}_1, \mathrm{q}_2)\ \mathrm{CNOT}(\mathrm{q}_2, \mathrm{q}_1)\ \mathrm{CNOT}(\mathrm{q}_1, \mathrm{q}_2) \\
\mathrm{TOF}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{q}_3) &\triangleq \textbf{qcase } \mathrm{q}_1 \textbf{ of } \{0 \to \textbf{ skip};, 1 \to \mathrm{CNOT}(\mathrm{q}_2, \mathrm{q}_3)\}
\end{aligned}$$

[2] In the case of quantum polynomial time, Adleman et al. [1] showed how the choice of amplitudes can affect the expressivity of classes such as BQP, requiring the restriction of *polynomial-time approximable complex amplitudes*. How the set of amplitudes influences the class FBQPOLYLOG remains an open question, as discussed in [20, 21], and so we abstain from the use of the entire set of complex numbers and instead use a field $\mathbb{K}$ which may refer to, for instance, polynomial-time approximable complex amplitudes.

$$\frac{\forall i \leq n,\ (\mathrm{e}_i, f) \Downarrow_{[\![\tau_i]\!]} x_i}{(\mathrm{op}(\mathrm{e}_1, \ldots, \mathrm{e}_n), f) \Downarrow_{[\![\mathrm{op}(\tau_1,\ldots,\tau_n)]\!]} [\![\mathrm{op}]\!](x_1, \ldots, x_n)} \qquad \frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_n]}{(|\mathrm{l}|, f) \Downarrow_{\mathbb{Z}} n}$$

$$\frac{\bar{\mathrm{q}} \in Var(\mathrm{P})}{(\bar{\mathrm{q}}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} f(\bar{\mathrm{q}})} \qquad \frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \in \{1, \ldots, m\}}{(\mathrm{l} \ominus [\mathrm{i}], f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m]}$$

$$\frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \notin \{1, \ldots, m\}}{(\mathrm{l} \ominus [\mathrm{i}], f) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} \qquad \frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad m > 1}{(\mathrm{l}^{\boxminus}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_{\lceil m/2 \rceil}]}$$

$$\frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad m > 1}{(\mathrm{l}^{\boxplus}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_{\lceil m/2 \rceil + 1}, \ldots, x_m]} \qquad \frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \notin \{1, \ldots, m\}}{(\mathrm{l}[\mathrm{i}], f) \Downarrow_{\mathbb{N}} 0}$$

$$\frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} l \qquad |l| \leq 1 \qquad \pm \in \{\boxminus, \boxplus\}}{(\mathrm{l}^{\pm}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} \qquad \frac{(\mathrm{l}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \qquad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \in \{1, \ldots, m\}}{(\mathrm{l}[\mathrm{i}], f) \Downarrow_{\mathbb{N}} x_k}$$

Figure 2 Semantics of expressions.

2.2 Semantics

Let $\mathbb{B} \triangleq \{0, 1\}$ denote the set of Booleans and $\mathcal{L}(\mathbb{N})$ denote the set of lists of natural numbers, [] being the empty list. We interpret basic types as follows:

$$[\![\text{Integers}]\!] \triangleq \mathbb{Z} \qquad [\![\text{Booleans}]\!] \triangleq \mathbb{B} \qquad [\![\text{Qubit lists}]\!] \triangleq \mathcal{L}(\mathbb{N}) \qquad [\![\text{Qubits}]\!] \triangleq \mathbb{N}$$

Qubits are interpreted as integers (pointers) and qubit lists are interpreted as lists of pointers. Each $\mathrm{op} \in \{\pm, /2, \geq, \wedge, \ldots\}$ of arity n comes with a basic type signature $\mathrm{op} :: \tau_1 \times \ldots \times \tau_n \to \tau$ and computes a fixed total function $[\![\mathrm{op}]\!] \in [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau]\!]$. We set $\mathrm{op}(\tau_1, \ldots, \tau_n) \triangleq \tau$. For example, $[\![/2]\!] \triangleq m \mapsto \lceil m/2 \rceil$. Constants k are treated as particular operators of arity 0. Given a program P, for each basic type τ, the semantics of expressions is described standardly in Figure 2 as a function

$$\Downarrow_{[\![\tau]\!]} : \tau \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})) \to [\![\tau]\!].$$

$(\mathrm{e}, f) \Downarrow_{[\![\tau]\!]} v$ holds when expression e of type τ evaluates to the value $v \in [\![\tau]\!]$ under the context $f \in Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})$. The context f is just a map from each program input to a list of qubit pointers taken into consideration when evaluating e. For instance, we have that $(\bar{\mathrm{q}}[2], \bar{\mathrm{q}} \mapsto [1, 4, 5]) \Downarrow_{\mathbb{N}} 4$ (the second qubit is of index 4), $(\bar{\mathrm{q}} \ominus [4], \bar{\mathrm{q}} \mapsto [1, 4, 5]) \Downarrow_{\mathcal{L}(\mathbb{N})} []$ ([] is used for errors on type $\mathcal{L}(\mathbb{N})$), $(\bar{\mathrm{q}}[4], \bar{\mathrm{q}} \mapsto [1, 4, 5]) \Downarrow_{\mathbb{N}} 0$ (index 0 is used for error on type $\mathbb{N}$), and $(\bar{\mathrm{q}} \ominus [3], \bar{\mathrm{q}} \mapsto [1, 4, 5]) \Downarrow_{\mathcal{L}(\mathbb{N})} [1, 4]$ (the third qubit has been removed).

Let $\mathcal{H}_{2^n}$ denote the Hilbert space $\mathbb{C}^{2^n}$ of n qubits with tensor product $\otimes$ and let $\mathcal{P}(\mathbb{N})$ denote the powerset of $\mathbb{N}$. Given a program P, let the *length* of P be a function mapping each qubit variable $\bar{\mathrm{q}} \in Var(\mathrm{P})$ to an integer $\mathtt{len}(\bar{\mathrm{q}}) \in \mathbb{N}$. We write $\mathtt{len}_{\mathrm{P}}$ as a shorthand for $\sum_{\bar{\mathrm{q}} \in Var(\mathrm{P})} \mathtt{len}(\bar{\mathrm{q}})$ and $\mathtt{len}_{\mathrm{P}}^{<\bar{\mathrm{q}}}$ as a shorthand for $\sum_{\bar{\mathrm{q}}' \in Var(\mathrm{P}),\ \bar{\mathrm{q}}' < \bar{\mathrm{q}}} \mathtt{len}(\bar{\mathrm{q}}')$.

A *configuration* c of program P over $\mathtt{len}_{\mathrm{P}}$ qubits is of the shape

$$(\mathrm{S}, |\psi\rangle, A, f) \in (\text{Statements} \cup \{\top, \bot\}) \times \mathcal{H}_{2^{\mathtt{len}_{\mathrm{P}}}} \times (Var(\mathrm{P}) \to \mathcal{P}(\mathbb{N})) \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})),$$

where $\top$ and $\bot$ are two special symbols denoting termination and error, respectively, where $|\psi\rangle \in \mathcal{H}_{2^{\mathtt{len}_{\mathrm{P}}}}$ is a quantum state, and where, for each qubit list $\bar{\mathrm{q}} \in Var(\mathrm{P})$, $A(\bar{\mathrm{q}})$ is the

set of qubit pointers accessible from $\bar{\mathrm{q}}$ and $f(\bar{\mathrm{q}})$ is the list of qubit pointers assigned to $\bar{\mathrm{q}}$. Given a qubit q such that $Var(\mathrm{q}) = \{\bar{\mathrm{q}}\}$, we write $A(\mathrm{q})$ as a shorthand for $A(\bar{\mathrm{q}})$ and we write $A_{\mathrm{q}\setminus\{n\}}$ for the function $A' \in Var(\mathrm{P}) \to \mathcal{P}(\mathbb{N})$ defined by $A'(\bar{\mathrm{q}}') \triangleq A(\bar{\mathrm{q}}')$, $\forall \bar{\mathrm{q}}' \neq \bar{\mathrm{q}}$, and $A'(\bar{\mathrm{q}}) \triangleq A(\bar{\mathrm{q}})\setminus\{n\}$.

Given a program $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$, with $n = \mathtt{len}_{\mathrm{P}}$, let Conf_n be the set of configurations over n qubits. The initial configuration in Conf_n on input state $|\psi\rangle \in \mathcal{H}_{2^n}$ is $c_{init}(|\psi\rangle) \triangleq (\mathrm{S}, |\psi\rangle, \bar{\mathrm{q}} \mapsto \{1, \ldots, \mathtt{len}(\bar{\mathrm{q}})\}, \bar{\mathrm{q}} \mapsto [1, \ldots, \mathtt{len}(\bar{\mathrm{q}})])$. A final configuration can be defined in the same way as $c_{final}(|\psi\rangle) \triangleq (\top, |\psi\rangle, \bar{\mathrm{q}} \mapsto \{1, \ldots, \mathtt{len}(\bar{\mathrm{q}})\}, \bar{\mathrm{q}} \mapsto [1, \ldots, \mathtt{len}(\bar{\mathrm{q}})])$.

Each unitary transformation U of a unitary application q `*=` $\mathrm{U}^g(\mathrm{i})$; comes with a function $[\![\mathrm{U}]\!]$ assigning a unitary matrix $[\![\mathrm{U}]\!](g)(n) \in \mathbb{K}^{2\times 2}$ to each integer n and function $g \in \mathbb{Z} \to [0, 2\pi)$. We restrict ourselves to three kinds of gates: the phase gate Ph, rotation gate $\mathrm{R_Y}$ and NOT gate NOT, whose semantics is defined as follows:

$$[\![\mathrm{Ph}]\!](g)(n) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{ig(n)} \end{pmatrix}$$

$$[\![\mathrm{R_Y}]\!](g)(n) \triangleq \begin{pmatrix} \cos(g(n)) & -\sin(g(n)) \\ \sin(g(n)) & \cos(g(n)) \end{pmatrix}$$

$$[\![\mathrm{NOT}]\!](\cdot)(\cdot) \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The big-step semantics $\cdot \longrightarrow \cdot$ is defined in Figure 3 as a relation in $\bigcup_{n\in\mathbb{N}} \mathrm{Conf}_n \times \mathrm{Conf}_n$. The symbols $\bot$ and $\top$ for error and termination, respectively, are terminal states: they cannot appear on the left-hand-side of a rule. In Figure 3, the function A of accessible qubits is used to ensure that unitary operations on qubits can be physically implemented. For example, the statements S_0 and S_1 of a quantum branch **qcase** q **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ cannot access the control qubit q to ensure reversibility. To avoid cumbersome use of nested conditionals in the program syntax, each procedure call on a (at least one) empty qubit list is semantically equivalent to a **skip**; (see Figure 3).

We write $[\![\mathrm{P}]\!](|\psi\rangle) = |\psi'\rangle$, whenever $c_{init}(|\psi\rangle) \longrightarrow c_{final}(|\psi'\rangle)$ holds. If the program P terminates on all inputs (i.e., always reaches a final configuration) then $[\![\mathrm{P}]\!]$ is a total function on quantum states. Note that if a program terminates then it is obviously error-free (i.e., does not reach a configuration with a $\bot$) but the converse property does not hold. However, every program P can be efficiently transformed into an error-free program $\mathrm{P}_{\neg\bot}$ such that $\forall |\psi\rangle$, if $[\![\mathrm{P}]\!](|\psi\rangle)$ is defined then $[\![\mathrm{P}]\!](|\psi\rangle) = [\![\mathrm{P}_{\neg\bot}]\!](|\psi\rangle)$. This can be done, for instance, by checking the size of qubit lists before accessing them. Hence we will restrict ourselves to error-free programs in what follows.

When an input state is defined by different qubit lists, we denote them in subscript. For instance, for $x, y \in \{0,1\}^*$ and $m \in \mathbb{N}$, we have that $|x\rangle_{\bar{\mathrm{q}}_1} |y\rangle_{\bar{\mathrm{q}}_2}$ indicates state $|x\rangle$ given as input to qubit list $\bar{\mathrm{q}}_1$, state $|y\rangle$ given as input to qubit list $\bar{\mathrm{q}}_2$.

▶ **Example 1** (Binary search). Let $x \in 0^*1^*2^*$ be a sorted string and $\hat{x}$ denote the encoding of x as a binary given by $\hat{0} \triangleq 00$, $\hat{1} \triangleq 01$, and $\hat{2} \triangleq 10$. Program `SEARCH` in Figure 4 computes the function $[\![\mathtt{SEARCH}]\!](|\hat{x}\rangle_{\bar{\mathrm{q}}_1} |0\rangle_{\bar{\mathrm{q}}_2}) = |\hat{x}\rangle_{\bar{\mathrm{q}}_1} |b\rangle_{\bar{\mathrm{q}}_2}$, where $b \in \{0,1\}$ indicates whether x contains a 1 or not.

▶ **Example 2** (Square Log). Program `SQLOG` of Figure 4 defines two recursive procedures `f` and `g` and its complexity is square logarithmic in the size of the first qubit list $\bar{\mathrm{q}}_1$. The procedure `g` has logarithmic complexity in $|\bar{\mathrm{q}}_1|$ as each recursive call to `g` divides the size of the first argument by 2. Similarly, the procedure `f` calls itself a logarithmic number of times and calls `g` each time, hence accounting for a $O(\log^2(|\bar{\mathrm{q}}_1|))$ complexity.

$$\frac{}{(\mathbf{skip};,|\psi\rangle,A,f)\longrightarrow(\top,|\psi\rangle,A,f)} \qquad \frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\notin A(\mathrm{q})}{(\mathrm{q}\ \texttt{*=}\ \mathrm{U}^g(\mathrm{j});,|\psi\rangle,A,f)\longrightarrow(\bot,|\psi\rangle,A,f)}$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad (\mathrm{i},f)\Downarrow_{\mathbb{N}} m \qquad j=\mathtt{len}_{\mathrm{P}}^{<\mathrm{q}}+n}{(\mathrm{q}\ \texttt{*=}\ \mathrm{U}^g(\mathrm{i});,|\psi\rangle,A,f)\longrightarrow(\top,I_{2^{j-1}}\otimes[\![\mathrm{U}]\!](g)(m)\otimes I_{2^{\mathtt{len}_{\mathrm{P}}-j}}|\psi\rangle,A,f)}$$

$$\frac{(\mathrm{S}_0,|\psi\rangle,A,f)\longrightarrow(\top,|\psi'\rangle,A,f) \qquad (\mathrm{S}_1,|\psi'\rangle,A,l)\longrightarrow(\diamond,|\psi''\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\mathrm{S}_0\ \mathrm{S}_1,|\psi\rangle,A,f)\longrightarrow(\diamond,|\psi''\rangle,A,f)}$$

$$\frac{(\mathrm{S}_0,|\psi\rangle,A,f)\longrightarrow(\bot,|\psi\rangle,A,f)}{(\mathrm{S}_0\ \mathrm{S}_1,|\psi\rangle,A,f)\longrightarrow(\bot,|\psi\rangle,A,f)}$$

$$\frac{(\mathrm{b},f)\Downarrow_{\mathbb{B}} b \qquad (\mathrm{S}_b,|\psi\rangle,A,f)\longrightarrow(\diamond,|\psi'\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \{\mathrm{S}_1\}\ \mathbf{else}\ \{\mathrm{S}_0\},|\psi\rangle,A,f)\longrightarrow(\diamond,|\psi'\rangle,A,f)}$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad \forall k\in\mathbb{B},\ (\mathrm{S}_k,|\psi\rangle,A_{\mathrm{q}\setminus\{n\}},f)\longrightarrow(\top,|\psi_k\rangle,A_{\mathrm{q}\setminus\{n\}},f) \qquad j=\mathtt{len}_{\mathrm{P}}^{<\mathrm{q}}+n}{(\mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\longrightarrow(\top,\sum_{k\in\{0,1\}}|k\rangle_j\langle k|_j|\psi_k\rangle,A,f)}$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad \exists k\in\mathbb{B},\ (\mathrm{S}_k,|\psi\rangle,A_{\mathrm{q}\setminus\{n\}},f)\longrightarrow(\bot,|\psi_k\rangle,A_{\mathrm{q}\setminus\{n\}},f)}{(\mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\longrightarrow(\bot,|\psi\rangle,A,f)}$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\notin A(\mathrm{q})}{(\mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\longrightarrow(\bot,|\psi\rangle,A,f)}$$

$$\frac{\forall j\le n,\ (\mathrm{l}_j,f)\Downarrow_{\mathcal{L}(\mathbb{N})} l_j\neq[] \qquad (\mathrm{S}^{\mathrm{proc}}\{\mathrm{l}_j/\bar{\mathrm{q}}_j\},|\psi\rangle,A,f)\longrightarrow(\diamond,|\psi'\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\mathbf{call}\ \mathrm{proc}(\mathrm{l}_1,\ldots,\mathrm{l}_n);,|\psi\rangle,A,f)\longrightarrow(\diamond,|\psi'\rangle,A,f)}$$

$$\frac{\exists j\le n,\ (\mathrm{l}_j,f)\Downarrow_{\mathcal{L}(\mathbb{N})}[]}{(\mathbf{call}\ \mathrm{proc}(\mathrm{l}_1,\ldots,\mathrm{l}_n);,|\psi\rangle,A,f)\longrightarrow(\top,|\psi\rangle,A,f)}$$

Figure 3 Semantics of statements.

2.3 Polylogarithmic Time Restrictions

We now define some restrictions on the admissible programs to guarantee that they terminate in polylogarithmic time (i.e., each procedure cannot perform more than a polylogarithmic number of recursive calls in the input size) and that their total number of sequential procedure calls (i.e., calls that are not in superposition) is bounded polylogarithmically.

Towards that end, we define a relation between procedures to account for recursion. Given a program $\mathrm{P}\triangleq\mathrm{D}::\mathrm{S}$, the call relation $\to_{\mathrm{P}}\subseteq\text{Procedures}\times\text{Procedures}$ is defined for any two procedures $\mathrm{proc}_1,\mathrm{proc}_2\in\mathrm{S}$ as $\mathrm{proc}_1\to_{\mathrm{P}}\mathrm{proc}_2$ whenever $\mathrm{proc}_2\in\mathrm{S}^{\mathrm{proc}_1}$. The relation $\succeq_{\mathrm{P}}$ is then the transitive closure of $\to_{\mathrm{P}}$. The relation $\sim_{\mathrm{P}}$ is defined by $\mathrm{proc}_1\sim_{\mathrm{P}}\mathrm{proc}_2$ if $\mathrm{proc}_1\succeq_{\mathrm{P}}\mathrm{proc}_2$ as well as $\mathrm{proc}_2\succeq_{\mathrm{P}}\mathrm{proc}_1$ both hold. Finally, the relation $\succ_{\mathrm{P}}$ is defined as $\mathrm{proc}_1\succ_{\mathrm{P}}\mathrm{proc}_2$ if $\mathrm{proc}_1\succeq_{\mathrm{P}}\mathrm{proc}_2$ and $\mathrm{proc}_1\not\sim_{\mathrm{P}}\mathrm{proc}_2$ both hold. A procedure proc is *recursive* whenever $\mathrm{proc}\sim_{\mathrm{P}}\mathrm{proc}$ holds. Two procedures proc and proc′ are *mutually recursive* whenever $\mathrm{proc}\sim_{\mathrm{P}}\mathrm{proc}'$ holds

▶ **Definition 3.** *A program* P *is said to be* recursively halving, *denoted* $\mathrm{P}\in\textsc{half}$, *if for each procedure* $\mathrm{proc}\in\mathrm{P}$ *and for each procedure call* $\mathbf{call}\ \mathrm{proc}'(\mathrm{l}_1,\ldots,\mathrm{l}_n);\in\mathrm{S}^{\mathrm{proc}}$,

if $\mathrm{proc}\sim_{\mathrm{P}}\mathrm{proc}'$ *then there are* $1\le i\le n$ *and* l *such that* $\mathrm{l}^{\boxminus}$ *or* $\mathrm{l}^{\boxplus}$ *appears in* l_i.

SEARCH

```
1  decl search(q̄1, q̄2) {
2    if |q̄1| > 1 then {
3      qcase q̄1[|q̄1|/2, |q̄1|/2 + 1] of {
4        00 → call search(q̄1^⊞ ⊖ [1], q̄2); ,
5        01 → q̄2[1] *= NOT; ,
6        10 → call search(q̄1^⊟ ⊖ [−1], q̄2); ,
7        11 → skip; }
8    } else { skip; }
9  } ::
10 call search(q̄1, q̄2);
```

SQLOG

```
1  decl f(q̄1, q̄2) {
2    q̄1[1] *= Ph^{λx.2π/x}(|q̄1|);
3    call f(q̄1^⊞, q̄2);
4    call g(q̄1, q̄2 ⊖ [1]); }
5  decl g(q̄1, q̄2) {
6    qcase q̄1[|q̄1|/2] of {
7      0 → call g(q̄1^⊟, q̄2); ,
8      1 → q̄2[1] *= NOT;
9  }} ::
10 call f(q̄1, q̄2);
```

Figure 4 Examples of PLP programs.

This restriction can be viewed as a recursion scheme, which implies a polylogarithmic time bound on programs in HALF by ensuring that in every (mutually) recursive procedure call at least one of the input qubit lists is cut in half.

Now we impose a further condition on the number of sequential (mutually) recursive procedure calls. For that purpose, we define the *width* of a program P in the following way.

▶ **Definition 4.** *Given a program* P*, the* width *of a procedure* proc ∈ P *is defined by* $\mathrm{width}_{\mathrm{P}}(\mathrm{proc}) \triangleq w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}^{\mathrm{proc}})$ *where* $w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S})$ *is defined inductively on statements by:*

$$
\begin{aligned}
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{skip};) &\triangleq 0\\
w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{q}\ \texttt{*=}\ \mathrm{U}^{g}(\mathrm{i});) &\triangleq 0\\
w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0\,\mathrm{S}_1) &\triangleq w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0) + w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1)\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{if}\ \mathrm{b}\ \textbf{then}\ \{\mathrm{S}_1\}\ \textbf{else}\ \{\mathrm{S}_0\}) &\triangleq \max(w_{\mathrm{P}}^{\mathrm{proc}}(S_0), w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1))\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}) &\triangleq \max(w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_0), w_{\mathrm{P}}^{\mathrm{proc}}(\mathrm{S}_1))\\
w_{\mathrm{P}}^{\mathrm{proc}}(\textbf{call}\ \mathrm{proc}'(\mathrm{l}_1,\ldots,\mathrm{l}_n);) &\triangleq \begin{cases} 1, & \textit{if}\ \mathrm{proc} \sim_{\mathrm{P}} \mathrm{proc}',\\ 0, & \textit{otherwise.}\end{cases}
\end{aligned}
$$

The width *of a program* width(P) *is defined by* $\mathrm{width}(\mathrm{P}) \triangleq \max_{\mathrm{proc}\in\mathrm{P}} \mathrm{width}_{\mathrm{P}}(\mathrm{proc})$*. Let* WIDTH$_{\leq 1}$ *be defined by* $\mathrm{WIDTH}_{\leq 1} \triangleq \{\mathrm{P} \mid \mathrm{width}(\mathrm{P}) \leq 1\}$.

▶ **Definition 5.** *The set* PLP *of* PolyLog Programs *is defined by* $\mathrm{PLP} \triangleq \mathrm{HALF} \cap \mathrm{WIDTH}_{\leq 1}$.

As we will see in next Section (Theorem 10), the restriction to PLP programs ensures that they can be simulated by quantum random-access Turing machines running in polylogarithmic time.

▶ **Example 6.** Both programs SEARCH and SQLOG of Figure 4 can be shown to be in PLP. Let us consider the case of SQLOG which has two recursive procedures: f and g. We have $\mathtt{f} \sim_{\mathtt{SQLOG}} \mathtt{f} \succ_{\mathtt{SQLOG}} \mathtt{g} \sim_{\mathtt{SQLOG}} \mathtt{g}$. It is easy to check that SQLOG ∈ HALF as the procedure f

recursively calls itself on $\bar{q}_1^{\boxplus}$ and g recursively calls itself on $\bar{q}_1^{\boxminus}$. Furthermore, we verify that:

$$\begin{aligned}\mathrm{width}(\texttt{SQLOG}) &= \max(\mathrm{width}_{\texttt{SQLOG}}(\mathtt{f}), \mathrm{width}_{\texttt{SQLOG}}(\mathtt{g}))\\ &= \max(w^{\mathtt{f}}_{\texttt{SQLOG}}(\mathrm{S}^{\mathtt{f}}), w^{\mathtt{g}}_{\texttt{SQLOG}}(\mathrm{S}^{\mathtt{g}}))\\ &= \max(0 + w^{\mathtt{f}}_{\texttt{SQLOG}}(\textbf{call } \mathtt{f}(\bar{q}_1^{\boxplus}, \bar{q}_2);) + w^{\mathtt{f}}_{\texttt{SQLOG}}(\textbf{call } \mathtt{g}(\bar{q}_1, \bar{q}_2 \ominus [1]);),\\ &\qquad \max(w^{\mathtt{g}}_{\texttt{SQLOG}}(\textbf{call } \mathtt{g}(\bar{q}_1^{\boxminus}, \bar{q}_2);), w^{\mathtt{g}}_{\texttt{SQLOG}}(\bar{q}_2[1] \texttt{ *= } \mathrm{NOT};)))\\ &= \max(0 + 1 + 0, \max(1, 0)) = 1\end{aligned}$$

2.4 Properties of PLP Programs

Because of the HALF condition, programs in PLP can be shown to be terminating.

▶ **Theorem 7.** *If* $\mathrm{P} \in$ PLP*, then* P *terminates.*

Proof. This is trivially ensured by the HALF condition, which shows that the size of arguments is strictly decreasing in recursive calls whenever the arguments are not empty as seen in the semantics of $\mathrm{l}^{\boxminus}/\mathrm{l}^{\boxplus}$ in Figure 2, and the program semantics of Figure 3, as procedure calls terminate on empty list. ◀

As PLP programs are quantum programs, they must be reversible. We show that we can constructively define a PLP program P^{-1} that computes the inverse of P.

▶ **Theorem 8** (Reversibility). *There exists a program transformation* $\cdot^{-1}$ *such that, for any* $\mathrm{P} \in$ PLP*,* $[\![\mathrm{P}^{-1}]\!] \circ [\![\mathrm{P}]\!]$ *is the identity and* $\mathrm{P}^{-1} \in$ PLP*.*

Proof. The program transformation can be constructively defined on program statements. For instance, $(\mathrm{q} \texttt{ *= } \mathrm{U}^g(\mathrm{i});)^{-1} \triangleq \mathrm{q} \texttt{ *= } (\mathrm{U}^g(\mathrm{i}))^\dagger$; and $(\mathrm{S}_0\ \mathrm{S}_1)^{-1} \triangleq \mathrm{S}_1^{-1}\ \mathrm{S}_0^{-1}$. ◀

Note that Theorems 7 and 8 can also be obtained as corollaries of the FBQPOLYLOG-soundness that will be proved in Theorem 10. We consider the ensured properties (termination and reversibility) as consistency checks before going into the complexity results.

3 A Characterization of Quantum Polylog Time

In this section, we will show that PLP characterizes exactly the functions in FBQPOLYLOG, the class of quantum polylog time approximable functions. That is, programs in PLP compute functions in FBQPOLYLOG (Soundness) and, reciprocally, for any function in FBQPOLYLOG, there exists a PLP program that computes it (Completeness).

3.1 Quantum Random Access Turing Machines and Polylog Time

To define the class FBQPOLYLOG, we introduce the computational model of *quantum random-access* Turing machines (QRATMs) [20, 21]. FBQPOLYLOG is not defined on standard quantum Turing machines because, due to its sub-linear time complexity, such a machine would not be able to access all of its input. Random-access machines solve part of the problem by allowing the machine to jump over its input.

A QRATM has a random access input tape, a log-space index tape, and c work tapes and is then defined as a triplet (Q, Σ, δ), where Q is a finite set of states containing an initial state s_0 and two (disjoint) subsets Q_{acc} and Q_{rej} for accepting and rejecting states, $\Sigma = \{0, 1, \#\}$ is the tape alphabet, and the transition function δ is such that

$$\delta : Q \times \Sigma^{1+c} \to (Q \times \Sigma^{1+c} \times \{L, R, N\}^{1+c} \to \mathbb{C}).$$

This transition maps the state and read symbols on the index tape and work tapes to a function mapping each state, each written symbol, and each head move to an amplitude. Note that the input tape does not have a tape head, hence is not taken into account in this transition function.

To get access to any character of the input, a special transition of the machine is defined: when the machine is in a special state s_{query}, the cell of input tape corresponding to the binary number written on the index tape is swapped with the cell under the work tape head, and the machine transitions to a state s_{accept}. Note that, in contrast with [20, 21], the input tape is not read-only as we consider a class of functions rather than decision problems, hence having the modified input be part of the output is necessary. This allows for example to consider the identity function.

A pure configuration of a QRATM is a tuple $(s, w, w_0, w_1, \ldots, w_c, z_0, z_1, \ldots, z_c) \in Q \times {\Sigma^*}^{2+c} \times \mathbb{Z}^{1+c}$ where s is a state, w the word written on the input tape, assumed to begin in cell 0, w_0 is the word on the index tape, w_1, ..., w_c the words written on the work tapes, z_0 is the position of the index tape head, z_1, ..., z_c the tape head positions for the work tapes, all positions are relative to the first character of the word. The initial configuration for input x is $\gamma(x) \triangleq (s_0, x, \epsilon, \ldots, 0, \ldots)$. We call a superposition of pure configurations a surface configuration. Surface configurations can be written as $\sum_r \alpha_r |r\rangle$, with r ranging over pure configurations, $\alpha_r \in \mathbb{C}$ is the amplitude associated with configuration r. QRATMs are also required to satisfy reversibility and well-formedness condition: a configuration may have only one predecessor, and the transition function must preserve the norm of configurations, that means that for all reachable surface configurations, $\sum_r |\alpha_r|^2 = 1$.

A QRATM halts in time t on input x if, starting from the initial configuration $\gamma(x)$, after t steps, its surface configuration is a superposition of pure configurations in accepting states. If for all input x, a QRATM M halts on input x in time $T(|x|)$, we say that M halts in time T. In particular, if there exists $k \in \mathbb{N}$ such that M halts in time $O(\log^k(n))$, M halts in *polylog time*. If a QRATM halts, its output is defined as the linear combination of the words on the input tape and work tapes, using the previous notations, it corresponds to $\sum_r \alpha_r |w^r, w_0^r, w_1^r, \ldots, w_c^r\rangle$. Given a function $f : \{0,1\}^* \to \{0,1\}^*$, we say that a QRATM M approximates f with probability p if for all input $x \in \{0,1\}^*$, starting from the initial configuration $\gamma(x)$, M halts with an output $\sum_r \alpha_r |w^r, w_0^r, w_1^r, \ldots, w_c^r\rangle$ such that $\sum_{r \in Q_{acc} \times \{f(x)\} \times \mathbb{Z}^{2+c}} |\alpha_r|^2 \geq p$.

▶ **Definition 9.** *The class* FBQPOLYLOG *is defined as the set of functions $f : \{0,1\}^* \to \{0,1\}^*$ such that there exists a QRATM approximating f with probability at least $\frac{2}{3}$ in polylog time.*

Although a natural theoretical model for describing polylogarithmic time, QRATMs are problematic because they are too semantic in nature: to characterise their complexity, the time bound must be given explicit and their halting condition depends on the inner state of the machine, which is a semantic condition. This motivates the provided characterization in the next section.

3.2 Main Result

We denote by $[\![\text{PLP}]\!]$ the set of functions computed by programs in PLP. That is $[\![\text{PLP}]\!] \triangleq \{[\![\text{P}]\!] \mid \text{P} \in \text{PLP}\}$. A program P approximates function $f : \{0,1\}^* \to \{0,1\}^*$ with probability $p \in [0,1]$ if $\forall x \in \{0,1\}^*, |\langle f(x) | [\![\text{P}]\!](x) \rangle|^2 \geq p$, in other words if for all input, the output of P coincides with f with probability at least p. The set of functions that can be approximated with probability at least p is denoted by $[\![\text{PLP}]\!]_{\geq p}$.

▶ **Theorem 10** (Soundness & Completeness). $[\![\text{PLP}]\!]_{\geq \frac{2}{3}} = \text{FBQPOLYLOG}$.

Proof. Soundness is proved via the fact that the HALF and $\text{WIDTH}_{\geq 1}$ restrictions imply a polylogarithmic bound for the depth of the call tree and a logarithmic bound on the degree of this tree. Then we show that if some PLP program P approximates f, then the poly-logarithmic time QRATM simulating P also approximates f and guarantees that $f \in$ FBQPOLYLOG.

Conversely, for completeness, given a polylog time QRATM M, we define a PLP program that simulates M. To achieve this, we represent the input tape, the index tape and the work tapes using qubit lists, the state of the Turing machine is encoded inside the work tape by writing it to the left of the character under the tape head. To simulate the execution of M, we define the following procedures:

- `access_input`: allows for QRATM-like access to the input tape by performing quantum branching on each cell of the index tape. The correct cell on the input tape to be read is determined by "splitting" in half the set of possible input tape addresses according to the value of each index tape cell.
- `local_step`: simulates a constant-time transition of M locally on three adjacent cells of the index tape and the work tape, or calls `access_input` for the query step.
- `full_step`: performs `local_step` iteratively to simulate a transition of M over the entirety of the index and work tapes.
- `iterate`: executes `full_step` a polylogarithmic number of times, simulating the entire run of M.

These procedures can be combined to obtain a PLP program simulating M, by encoding M's tapes in a way that allows for a local evolution of the state. ◀

4 Circuit Compilation

In this section, we sketch an algorithm that compiles PLP programs into circuits of polynomial size and polylogarithmic depth. An implementation of this algorithm, that also works for non-PLP programs, is available at `https://gitlab.inria.fr/mmachado/pfoq-compiler`. The two PLP programs `SEARCH` and `SQLOG` displayed in Figure 4 are also provided in the repository. Here, we describe the salient points of the compilation and show that the circuit obtained indeed has polylogarithmic depth.

The compilation strategy takes inspiration from [12, 13] which in particular uses ancillas to factorize the circuits representing procedure calls in branches so as to prevent exponential blow-up of the circuit size. Their technique is called *anchoring and merging* as when a procedure call is first encountered, an ancilla is associated to this call (*anchoring*), and when a subsequent call to this procedure happens with an input of the same size, this second call is then merged with the first (*merging*). This way, instead of doubling the size of the circuit whenever recursive calls appear in separate branches of a **qcase**, as in programs `SEARCH` and `SQLOG` of Figure 4, the size grows linearly in the number of nested recursive calls, hence preventing the exponential blow-up in complexity from the use of the quantum control statement [24].

Figure 5 exemplifies this phenomenon on the `SEARCH` program: the circuit on the left represented by a grey and white box the circuit for the `search` procedure applied to $\bar{q}_1, \bar{q}_2$. Since this procedure has two calls to itself, its natural compilation gives an in-depth duplication of the calls. The anchoring/merging process entails a single recursive compilation at the price of an overhead in terms of ancillary qubits and permutations.

4.1 Outline of the Compilation Algorithm

The compilation algorithm takes as input a program $\mathrm{P} \in \textsc{plp}$ together with a list of the sizes $\bar{s} \triangleq [s_1, \ldots, s_n]$ of its inputs. As in the semantics, the algorithm maintains a function that maps qubit list variables to lists of pointers to their qubits. Since the values of this function only depend on the size of the qubit variable, the generation of the circuit does not need to take qubit values into account.

The main compilation algorithm works by induction on the structure of the program statement. Compiling statements such as l[i] `*=` U^g(j); is straightforward: use the semantics to compute the wire number and which quantum gate to put into the circuit. Compiling a sequence is naturally done by composing the circuits obtained from compiling each statement. For compiling an **if** statement, note that Booleans only depend on constants and the size of qubit lists, and hence can be computed from the knowledge of list $\bar{s}$.

A **qcase** statement is compiled using controlled operations: consider **qcase** l[i] **of** $\{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$, the circuit compiled for S_0 should be controlled negatively on the wire computed for l[i], the circuit compiled for S_1 should be controlled positively on the same wire. To keep track of those controls, a structure is maintained that lists the control qubits and their state. The compilation of a non-recursive **call** consists in compiling the statement of the procedure after substituting its arguments by their expressions provided the qubit lists are non empty.

The only case that introduces complexity is that of recursive calls. A naive compilation strategy for recursive calls with a recursive procedure calling itself in both branches of a **qcase**, which is allowed by the $\textsc{width}_{\leq 1}$ restriction, would yield a number of gates in the compiled circuit exponential in the recursion depth. To prevent this blow-up, the process (anchoring and merging) maintains a dictionary of ancillaries that associates an ancillary qubit to pairs of procedure names and sizes of the inputs: if the dictionary does not have a key for the considered procedure call, an anchoring ancillary is created, this ancillary is initialized through a *multiple controlled-NOT* (Toffoli gate) encoding the **qcase** control considered and used to control the quantum circuit of the procedure statement; if the key is present in the dictionary, this ancillary is updated with another Toffoli gate and the two procedure calls are automatically merged as illustrated in Figure 5. This strategy is sound as the $\textsc{width}_{\leq 1}$ restriction ensures that two repeated calls always occur in orthogonal branches and can be simply combined in the same ancillary qubit.

To show the polylogarithmic depth bound on circuits implementing PLP programs, we first need to demonstrate that merging in the context of PLP can be done in polylogarithmic depth. Second, we show that the $\textsc{width}_{\geq 1}$ and HALF restrictions imply that the number of nested recursive procedure calls is polylogarithmically bounded.

4.2 Compilation to a Circuit of Polylog Depth

In a PLP program, a recursive procedure call always cuts one input qubit list in half. This means that the procedure will work either on the first half or on the second half of the qubit list. In the worst case, which is also the most typical, the procedure will be called recursively on each half depending on some condition. To avoid doubling the treatment of those calls, the anchoring and merging process merges those calls in such a way that the subcircuit for the procedure on half as many qubits is able to work on both halves. This implies conditionally swapping the two halves.

For instance, consider the `SEARCH` program in Figure 4 performing binary search. The procedure body of `search`, in the recursive case (i.e. where the input $\bar{\mathrm{q}}_1$ has size at least 4) consists of three (non-trivial) quantum cases. According to the state of the control qubits, we

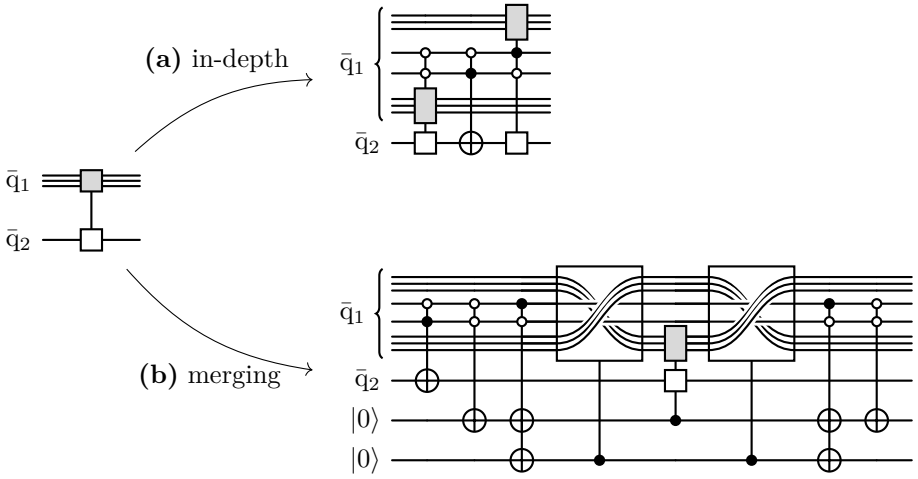

Figure 5 Compilation strategies for `search` defined in Figure 4.

either (1) perform a recursive call to `search` on the second half of $\bar{q}_1$, (2) apply a NOT gate to $\bar{q}_2$, or (3) perform a recursive call to `search` on the first half of $\bar{q}_1$. Compiling these three branches in sequence incurs a recursive doubling of instances of `search`, as shown on the left circuit of Figure 5 where the circuit corresponding to the `search` procedure is symbolized by a gray box and a white box. This doubling can be avoided by *merging* the two calls to `search` in the same circuit, using controlled-swap gates (also called Fredkin gates). This new circuit, given on the right of Figure 5, contains only a single call to `search`. Note that two ancillas are used: one for controlling whether the recursive `search` is executed, the other for controlling the swapping between the first and second half. This accounts for a constant added cost for each recursive call. In addition, the cost of the controlled permutation between the first and second halves of $\bar{q}_1$ should be taken into account.

▶ **Lemma 11.** *A controlled permutation on n qubits can be performed by a quantum circuit of size $O(n)$ and depth $O(\log n)$.*

Proof. Any permutation can be written as the composition of two sets of disjoint transpositions, and therefore any permutation can be performed in constant time, using two time steps [16]. To perform a *controlled* permutation, it suffices to create $O(n)$ ancillas with the correct controlled state, which can be done in $O(\log n)$ depth with $O(n)$ gates. ◀

Note that the ancillas used to control the permutation are linear in number but can be reused for nested recursive calls, hence the total number of ancillary qubits used for compiling a program on n qubits will be linear in n.

In the case of Figure 5, we are able to merge the two instances of `search` since they have the same input size and therefore encode precisely the same unitary operation, up to a renaming of qubits. In a general program, one needs to account for the total number of procedures that differ either in procedure name, input size and integer input. We prove that this number is polylogarithmically bounded.

▶ **Lemma 12.** *A procedure call occurring in an* PLP *program on n input qubits results in $O(\log n)$ calls to mutually recursive procedures with unique sizes.*

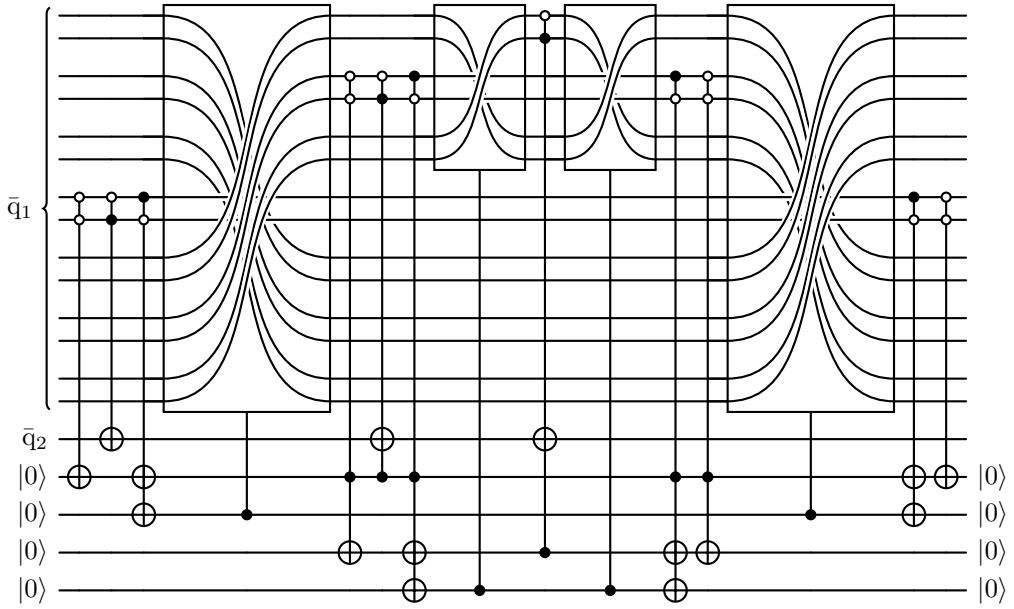

Figure 6 Quantum circuit resulting of compiling the SEARCH program.

In the above lemma, the number of input qubits is obtained by summing the sizes of (i.e., the number of qubits in) each operand of the procedure call. From those results, we obtain that the compilation process produces a quantum circuit of polynomial size and polylog depth in the size of the inputs.

▶ **Theorem 13** (Compilation). *Given a* PLP *program* P*, and input size* $n = \sum_{\bar{q} \in Var(P)} |\bar{q}|$*, the quantum circuit produced by the compilation process is of size* $O(n \operatorname{polylog}(n))$ *and depth* $O(\operatorname{polylog}(n))$*.*

▶ **Example 14.** To illustrate the compilation algorithm and the polylogarithmic depth bound, consider the `SEARCH` program from Figure 4. The compilation of this program with $|\bar{q}_1| = 14$ gives the quantum circuit depicted in Figure 6. Each recursive call yields a constant number of controlled gates and makes use of 2 ancillary qubits: 1 for anchoring and 1 to swap the first and second half. Thus the circuit obtained has depth $O(\log |\bar{q}_1|)$ and a number $O(\log |\bar{q}_1|)$ of ancillary qubits.

4.3 Limits of Quantum Polylogarithmic Time

Theorem 13 shows that programs in PLP can be compiled to quantum circuits of polylog depth and polynomial size, which means that they compute operators in QNC. In this section, we show that PLP is however not complete for QNC, thus recovering the known separation between FBQPOLYLOG and QNC.

QNC is defined in [16] as the union for all $k \in \mathbb{N}$ of the classes of quantum unitary transformations that can be computed by a family of quantum circuits of depth $O(\log^k n)$ with a polynomial number of ancillas. To compare with FBQPOLYLOG, we define FBQNC (for Bounded-error QNC) as in [7] as the class of functions $\{0,1\}^* \to \{0,1\}^*$ that can be approximated with probability at least $2/3$ by a QNC transformation.

To study the relationship between PLP and FBQNC, we use the *query model* of quantum computation which is the scenario in which one wishes to compute a function by making use of *black boxes* that are accessed via queries [2]. A black box performs a unitary transformation

on the quantum state according to some oracle function $\mathcal{O} : \{0,1\}^* \to \{0,1\}$, where the operation is usually defined as $|\bar{x}, y\rangle \mapsto |\bar{x}, y \oplus \mathcal{O}(\bar{x})\rangle$, which is considered to be performed in a single step.

Given a function f, we denote by $Q(f) : \mathbb{N} \to \mathbb{N}$ the function that maps $n \in \mathbb{N}$ to the least number of queries necessary for a quantum algorithm to approximate f with bounded-error on inputs of size n. The function $Q(f)$ gives a *lower bound* on the time complexity of approximating f. For instance, for the OR, AND, and PARITY defined as

$$\mathrm{OR}(\bar{x}) \triangleq \max_{i=1\ldots n} x_i \qquad \mathrm{AND}(\bar{x}) \triangleq \min_{i=1\ldots n} x_i \qquad \mathrm{PARITY}(\bar{x}) \triangleq \bigoplus_{i=1}^{n} x_i$$

we have that, while Grover's algorithm [11] allows for a quadratic speedup in the query complexity of AND and OR, no speedup exists for PARITY.

▶ **Lemma 15** ([25]). *For $f \in \{\mathrm{AND}, \mathrm{OR}\}$, we have that $Q(f) = \Theta(\sqrt{n})$.*

▶ **Lemma 16** ([9, 3]). *$Q(\mathrm{PARITY}) = \Theta(n)$.*

In contrast to the lower bounds on the query complexity for AND, OR, and PARITY, we can show that there is an upper bound on the query complexity of functions approximable by programs in PLP. This bound on quantum random-access Turing machines can be obtained by viewing the read-input transition as an oracle query as in [20, 21] and noting that the access to this oracle is bounded by the step-count of the QRATM.

▶ **Lemma 17.** *Let $f \in [\![\mathrm{PLP}]\!]_{\geq \frac{2}{3}}$. There exists $k \in \mathbb{N}$ such that $Q(f) = O(\log^k n)$.*

Lemma 17 gives a polylogarithmic upper bound on the query complexity of functions in $[\![\mathrm{PLP}]\!]_{\geq \frac{2}{3}}$. Lemmas 15 and 16 give lower bounds on the query complexity of AND, OR, and PARITY that are bigger than polylogarithms, hence proving that those functions are not in $[\![\mathrm{PLP}]\!]_{\geq \frac{2}{3}}$ even though they can be approximated by circuits of polylogarithmic depth and polynomial size.

▶ **Lemma 18.** $\mathrm{AND}, \mathrm{OR}, \mathrm{PARITY} \in \mathrm{FBQNC} \setminus [\![\mathrm{PLP}]\!]_{\geq \frac{2}{3}}$.

Combining this result with Theorem 13, we obtain a strict inclusion in FBQNC. Note that this strict inclusion is mandated by the well-known result that $\mathrm{FBQPOLYLOG} \subsetneq \mathrm{FBQNC}$.

▶ **Theorem 19.** $[\![\mathrm{PLP}]\!]_{\geq \frac{2}{3}} \subsetneq \mathrm{FBQNC}$.

5 Conclusion and Future Work

We presented a quantum programming language PLP that captures exactly FBQPOLYLOG, that is functions approximated in polylog time by a QRATM. This characterization relies on some restrictions, in particular on the arguments of recursive calls to guarantee the complexity bound. We show a compilation procedure that produces quantum circuits of polylog depth, hence recovering the inclusion in the class of quantum circuits of polylog depth and polynomial size QNC.

Extending PLP. The conditions imposed in the definition of PLP are by no means hard to extend while safeguarding most or even all the results presented here. For instance, the HALF condition can be relaxed such that we consider halving of the input not at every procedure call but in every closed loop of procedure calls within a given rank. Such an extension would increase the expressive power of the language, thus allowing a programmer more flexibility. In this paper we chose to have a streamlined language with restrictions that are simple to check in order to obtain a more readable characterization.

Characterizing QNC. To our knowledge, there are currently no implicit characterizations of FBQNC. Extending PLP to characterize this class would be particularly interesting. For example, adding a statement for recursively forking on each half of the quantum state would make it possible to capture AND, OR, and PARITY while still being sound for FBQNC.

References

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

2 Andris Ambainis. Understanding quantum algorithms via query complexity. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pages 3265–3285. World Scientific, 2018. doi:10.1142/9789813272880_0181.

3 Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald De Wolf. Quantum lower bounds by polynomials. *Journal of the ACM (JACM)*, 48(4):778–797, 2001. doi:10.1145/502090.502097.

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

5 Sergey Bravyi, David Gosset, and Robert König. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, 2018. doi:10.1126/science.aar3106.

6 Andrew M. Childs, Robin Kothari, Matt Kovacs-Deak, Aarthi Sundaram, and Daochen Wang. Quantum divide and conquer, 2022. doi:10.48550/arXiv.2210.06419.

7 Richard Cleve and John Watrous. Fast parallel circuits for the quantum Fourier transform. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, SFCS-00, pages 526–536. IEEE Comput. Soc, 2000. doi:10.1109/sfcs.2000.892140.

8 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.

9 Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Limit on the speed of quantum computation in determining parity. *Physical Review Letters*, 81(24):5442–5444, December 1998. doi:10.1103/physrevlett.81.5442.

10 Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, April 2008. doi:10.1103/PhysRevLett.100.160501.

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

12 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.

13 Emmanuel Hainry, Romain Péchoux, and Mário Silva. Branch sequentialization in quantum polytime. In *Formal Structures for Computation and Deduction, FSCD 2025*, volume 337 of *LIPIcs*, 2025.

14 W. Huang, C. H. Yang, K. W. Chan, T. Tanttu, B. Hensen, R. C. C. Leon, M. A. Fogarty, J. C. C. Hwang, F. E. Hudson, K. M. Itoh, A. Morello, A. Laucht, and A. S. Dzurak. Fidelity benchmarks for two-qubit gates in silicon. *Nature*, 569(7757):532–536, May 2019. doi:10.1038/s41586-019-1197-0.

15 Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1001–1014, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304023.

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

17 Yasuhiro Takahashi and Noboru Kunihiro. A fast quantum circuit for addition with few qubits. *Quantum Information and Computation*, 8(6):636–649, July 2008. doi:10.26421/QIC8.6-7-5.

18 Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Phys. Rev. A*, 54:147–153, July 1996. doi:10.1103/PhysRevA.54.147.

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

20 Tomoyuki Yamakami. Expressing power of elementary quantum recursion schemes for quantum logarithmic-time computability. In *Logic, Language, Information, and Computation (WoLLIC 2022)*, pages 88–104. Springer, 2022. doi:10.1007/978-3-031-15298-6_6.

21 Tomoyuki Yamakami. Elementary quantum recursion schemes that capture quantum polylogarithmic-time computability of quantum functions. *Mathematical Structures in Computer Science*, 34(7):710–745, August 2024. doi:10.1017/s0960129524000264.

22 Tomoyuki Yamakami. Quantum first-order logics that capture logarithmic-time/space quantum computability. In Ludovic Levy Patey, Elaine Pimentel, Lorenzo Galeotti, and Florin Manea, editors, *CiE 2024*, pages 311–323. Springer, 2024. doi:10.1007/978-3-031-64309-5_25.

23 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.

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

25 Christof Zalka. Grover's quantum searching algorithm is optimal. *Physical Review A*, 60(4):2746, 1999. doi:10.1103/PhysRevA.60.2746.