Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Integrating Dynamic Lifting into Qiskit

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Correlatore:
Dott.
ANDREA COLLEDAN

Presentata da:
OLGA BECCI

Sessione II
Anno Accademico 2022/2023

Introduction

In a forward-looking world, it is easy to be captivated by the excitement brought about by new technologies. One such technology is quantum computers.

Quantum computing allows us to solve some of the problems considered infeasible using classical computation. This is possibile because, with quantum computing, we alter the basic structure of information. Instead of working with classical information, we manipulate quantum information, namely qubits, and this allows us to exploit quantum phenomena like superposition, quantum interference, and entanglement to achieve the (sometimes superpolynomial) speed-up that makes this new approach to computation so appealing. There are several algorithms that demonstrate the possibilities of quantum computing in theory, most famously Shor's algorithm[18][17] and Glover's algorithm[14]. Nevertheless, we are still far away from a world where we can consider these hard problems solvable in practice, as building a quantum computer with a useful number of qubits remains a particularly difficult challenge. Even though practical quantum computers are not yet a reality, it is not surprising that the theoretical study of quantum computation is particularly attractive. The speedup promised by quantum computation does not come from the possibility of using more powerful hardware, instead you need to look at problems under a different light to find ways to take advantage of the aspects of quantum mechanics that bring about efficient solutions. In fact, even in a scenario where large quantum devices remain in the realm of science fiction, the insights gained by this new way of thinking about computation can still be useful in classical computation. Despite the challenges of building quantum computers, many are working towards making it a reality. Among those trying to bring quan-

tum computing to the real world is IBM, which has already built quantum devices with up to 127 qubits. IBM recognizes the importance of the software connected to this new hardware, and in their IBM Quantum Platform [1], they offer both access to quantum devices via the cloud and a learning platform to better understand quantum computation. IBM is where Qiskit was born. Qiskit [16] is an open-source quantum computing software development framework that serves as a gateway for both novices and experts to delve into the intricacies of quantum computation. It is accessible as a Python library and provides easy access to both real quantum processors and simulators for experimenting with quantum computing. However, the open source nature of the project and the fact that the development was primarily guided by what IBM's devices could allow, result in Qiskit being a cumbersome tool to design quantum algorithms.

With the goal of making tools like Qiskit more flexible, we look at dynamic lifting as a means to achieve this objective. Dynamic lifting refers to the capability of accessing the intermediate state of a wire during the circuit construction process, usually in the form of a boolean value, often for control flow reasons.

The ability to instruct a quantum computer to perform an operation such as "do this quantum operation when this wire has this value" is more challenging than one may realize. It is possibile to implement circuits using these types of conditionals without checking the values of qubits mid-circuit execution [19]. However, from the perspective of someone designing an algorithm, the structure "if this, then that" is way more intuitive than the corresponding complex circuit without the conditional. Algorithms that require these mechanics (like quantum teleportation or error correction codes) become more intuitive once we introduce a form of dynamic lifting. We thus achieve better ergonomics.

In this context, we present the following work on introducing dynamic lifting in Qiskit, structured as follows:

1. In the first chapter, we give an introduction to quantum computing and some key quantum theory concepts. We discusse the nature of qubits, su-

perposition, the role of probability amplitudes, quantum interference and entanglement. We present the most common unitary operators and give an example of computation.

2. In the second chapter, we describe the quantum circuit model, an intuitive computational framework which gained prominence over the quantum Turing machine. We explore its features, universal gates, and representation guidelines. Additionally, we introduce Qiskit, an open-source framework for quantum computing implemented in Python. We discuss the core workflow of Qiskit, involving building, compiling, running, and analyzing quantum circuits, providing insights into practical quantum algorithm development.

3. In the third chapter, we explain our implementation of dynamic lifting in Qiskit. We introduce the `LiftedValue` class and the `ifdl` command and a global circuit, which enhance Qiskit's capabilities for conditional quantum circuit construction. This refinement is exemplified in quantum teleportation and error correction demonstrations.

4. In the fourth chapter, we lay the foundation for a formal definition of the language resulting from the implementation. We provide a syntax and an operational semantics for the language, paying particular attention to the operational interpretation of dynamic lifting.

Contents

1 Quantum Computing **1**

1.1 Quantum States . . . 2

1.2 Dynamics of Quantum Systems . . . 6

1.3 Measurement in Quantum Systems . . . 9

1.3.1 Example: Bell State Construction . . . 9

2 Quantum Circuit Description Languages **13**

2.1 Quantum Circuits . . . 14

2.2 Languages . . . 15

2.3 Qiskit . . . 16

3 Qiskit with Dynamic Lifting **23**

3.1 Language . . . 24

3.1.1 `LiftedValue` . . . 24

3.1.2 `ifdl` . . . 25

3.1.3 Global Circuit . . . 27

3.2 Examples . . . 27

3.2.1 Quantum Teleportation . . . 28

3.2.2 Error Correction . . . 30

4 The Operational Semantics of Dynamic Lifting **35**

4.1 Syntax . . . 35

4.2 Operational Semantics . . . 40

4.2.1 An Example 46

Conclusions **51**

A Code **53**

Bibliography **71**

List of Figures

4.1 Full syntax. 38
4.2 Arithmetic and Boolean expressions operational semantic rules. . . . 49
4.3 Classical commands operational semantic rules. 50

Chapter 1

Quantum Computing

When discussing computation within the classical context, our inclination is often to think about it purely in abstracts terms, separating the physical reality of computation from the theoretical framework. There is no prerequisite for a lesson in electronics before starting to learn about programming. Nevertheless, it is true though that computation is intrinsically linked to the physical objects on which it operates, a connection that becomes even more pronounced in the realm of quantum computing.

For instance, a foundational aspect of designing a quantum algorithm involves constructing quantum interference to amplify desired results and diminish undesired outcomes. This seemingly straightforward definition, however, poses a challenge for comprehension when one lacks familiarity with the concept of quantum interference.

Furthermore, quantum physics concepts as quantum interference not only shape theoretical constructs but intricately influences the physical architecture of quantum computers. Importantly, the realization of quantum interference is contingent upon fundamental alterations to the hardware itself. This intertwining of theory and hardware underscores the trasformative nature of quantum computing.

While a comprehensive exploration of quantum mechanics is outside the scope of this work, our aim in this first chapter is to elucidate the fundamental concepts of quantum computing, with a focus on the facets essential for comprehending the

subsequent sections of this thesis.

1.1 Quantum States

When discussing quantum computation, what we talk about is the manipulation of quantum information. The smallest unit of quantum information is represented by **qubits**.

The key distinction between a qubit and a classical bit lies in the fact that while a bit can only represent either a 1 or a 0, a qubit can exist in both states simultaneously. This counterintuitive and intriguing aspect of quantum mechanics is known as *superposition* and it will be the first mechanic we will encounter in our exploration of quantum computation.

To comprehend how superposition is possible, we need to take a moment to reason about probability. Quantum theory introduces the fact that probabilities can be expressed using complex numbers. We do this by introducing the concept of *amplitude*, represented by a complex number c whose square of the absolute value $|c|^2$ is interpreted as a probability.

Just as with probabilities, amplitudes can be combined: when we have events occurring as a sequence of independent steps, their amplitudes are multiplied; when we have events with alternative outcomes, their amplitudes are added.

Now, let us return to the discussion of qubits and bits. A bit has two alternative ways of exciting, 0 or 1, and when it is in state 0 it has 100% chance of being 0. In terms of amplitudes, we can say that a bit in state 0 results from the two alternative events: "being in state 0" or "being in state 1", with a probability amplitude of 1 for the former and 0 for the latter.

Mathematically, we can represent the states of a classical bit as two-dimensional vectors, where the coefficients serve as the amplitudes:

$$|\mathbf{0}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |\mathbf{1}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

When discussing qubits, as mentioned earlier, we have to take into account the possibility that they can exist in both states simultaneously. What this means is that the two alternative events "being in state 0" or "being in state 1", can actually happen with any amplitude c_1 and c_2 as long as $(c_0 + c_1)^2 = 1$. Mathematically, we can see a generic qubit as the two-dimensional vector:

$$|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \qquad \text{where } (c_0 + c_1)^2 = 1, \quad c_0, c_1 \in \mathbb{C}$$

This representation places a qubit within the $\mathbb{C}^2$ vector space.

We usually represent qubits as a linear combination of basis states of the $\mathbb{C}^2$ vector space, usually $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

To gain further insight on what is happening, let us focus again on probabilities.

We have the two alternative outcomes (state $|0\rangle$ and $|1\rangle$), with probability amplitudes c_0 and c_1. When determining the probabilities of either event occurring we have to add the amplitudes and then take their absolute value squared[1]:

$$\begin{aligned} p = |c_0 + c_1|^2 &= |c_0|^2 + |c_1|^2 + c_0^* c_1 + c_0 c_1^* \\ &= p_0 + p_1 + |c_0||c_1|(e^{i(\phi_1 - \phi_0)} + e^{-i(\phi_1 - \phi_0)}) \\ &= p_0 + p_1 + 2\sqrt{p_0 p_1} cos(\phi_1 - \phi_0). \end{aligned}$$

This calculation reveals that when determining probabilities using amplitudes, a new term appears alongside the classical probabilities one would expect: $2\sqrt{p_0 p_1} cos(\phi_1 - \phi_0)$. This phenomenon is known as *quantum interference*. Depending on whether the term $\phi_1 - \phi_0$ (the *relative phase*) is positive or negative,

[1] We use $p_0 = |c_0|^2$ and $p_1 = |c_0|^2$ to indicate the probabilities of the two events independently. We use the polar representation $c_0 = |c_0|e^{i\phi_0}$ and $c_1 = |c_1|e^{i\phi_1}$ for the complex numbers.

we can have a constructing interference that amplifies the probability of the outcome, or a destructive interference that suppresses it. It is important to notice that the relative phase depends on both outcomes simultaneously, underscoring the necessity of superposition for interference to happen.

Ultimately, quantum computing is about modifying the phase of a quantum state to manipulate the interference in such a way as to amplify the probability of the desired outcomes.

Until now, we have explored the properties of a single qubit. Surprisingly, even with just one qubit, we can achieve more than one might imagine. However, to delve into more intriguing applications, we need more than one qubit. Mathematically, to combine qubits to form a multi-qubit system, we use the tensor product. Suppose we have two qubits $|x\rangle = x_0 |0\rangle + x_1 |1\rangle$ and $|y\rangle = y_0 |0\rangle + y_1 |1\rangle$. Their combined state is given by $|x\rangle \otimes |y\rangle$, which we denoted as $|xy\rangle$:

$$|xy\rangle = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \otimes \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 y_0 \\ x_0 y_1 \\ x_1 y_0 \\ x_1 y_1 \end{pmatrix} = x_0 y_0 |00\rangle + x_0 y_1 |01\rangle + x_1 y_0 |10\rangle + x_1 y_1 |11\rangle .$$

Obviously, we can expand beyond two qubit. For instance, an 8-qubit system (*qubyte*) would belong to the $\mathbb{C}^{256}$ vector space.

The vector representation of a qubyte is:

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{98} \\ c_{99} \\ c_{100} \\ \vdots \\ c_{255} \end{pmatrix}$$

This complexity illustrates why simulating quantum computation on classical computers is challenging. Storing a single qubyte requires 256 complex values.

When working with multiple qubits, we encounter another fascinating and puzzling phenomenon of quantum mechanics: *quantum entanglement*. Quantum entanglement occurs when two or more particles become interconnected in a way that changes in one particle's state instantaneously affect the other's, regardless of distance. This phenomenon arises in complex systems that cannot be written as a combination of their constituent subsystems.

Let us delve into how it works. Consider a two-qubit system and the state $|\phi\rangle$ in this system:

$$|\phi\rangle = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Now, let us attempt to express this state in terms of the tensor product of two independent one-qubit systems:

$$(c_0|0\rangle + c_1|1\rangle) \otimes (d_0|0\rangle + d_1|1\rangle) = c_0d_0|00\rangle + c_0d_1|01\rangle + c_1d_0|10\rangle + c_1d_1|11\rangle .$$

To do so we have to find c_0, c_1, d_0, d_1 such that $c_0d_1 = c_1d_0 = \frac{1}{\sqrt{2}}$ and $c_0d_0 = c_1d_1 = 0$, which is impossible. Thus, $|\phi\rangle$ is an entangled state.

It is intuitively reasonable that there are composite states that cannot be expressed in terms of their constituents since the set of elements in $\mathbb{C}^2 \otimes \mathbb{C}^2$ is larger than the number of elements in $\mathbb{C}^2 + \mathbb{C}^2$. Consequently, the combined space of $\mathbb{C}^2 + \mathbb{C}^2$ lacks sufficient elements to fully capture the entirety of $\mathbb{C}^2 \otimes \mathbb{C}^2$.

Returning to the example of the state $|\phi\rangle$, let us explore how the 'spooky action at a distance' defining quantum entanglement works.

When we calculate the probabilities of finding the system in any state, we find the probabilities $p(|00\rangle) = 0$, $p(|01\rangle) = \frac{1}{2}$, $p(|10\rangle) = \frac{1}{2}$ and $p(|11\rangle) = 0$. This means we have a 50/50 chance of observing the states $|10\rangle$ and $|01\rangle$. The state is in a super position. If the first qubit 'collapses' into state $|0\rangle$ when observed, we will know for certain that the second bit must be in state $|1\rangle$, because there is a probability 0 that the system could have collapsed into state $|00\rangle$. This stays true no metter how far apart we take the two qubits.

1.2 Dynamics of Quantum Systems

We have established what a state is in a quantum system and we have explored the basic quantum phenomena we want to exploit in quantum computation. Now, we need a way to actually influence and change the quantum system. We want tools to put a qubit in a superposition, to entangle a system, and manipulate the interference within the system. We need a way to describe the evolution of a system from one state to another over time. We need to talk about its **dynamics**.

In a quantum system, the evolution is described by *unitary operators*. If U is a unitary matrix representing a unitary operator and $|\phi(t)\rangle$ is the state of the system at time t, then the system at the next time step is given by

$$|\phi(t+1)\rangle = U\,|\phi(t)\rangle\,.$$

Multiple unitary operators $U_1, U_2, \ldots, U_n$ are combined as

$$|\phi(t+1)\rangle = U_n \ldots U_1\,|\phi(t)\rangle\,.$$

Although the connection between unitary operators and their physical implementation is interesting, it is outside our scope, so we focus only on their mathematical interpretation.

Unitary operators can be seen as a counterpart to logical operators in the classical realm. Unitary operators are useful in describing the evolution of quantum systems because all the operations modifying the system must be reversible, which unitary operators inherently are. Like with qubits, the coefficients in the unitary matrices representing the operators can be complex numbers, and there exists an uncountable number of possibile unitary operators. As with everything in quantum physics, there are unitary operators that, when analyzed in terms of classical logic or classical probability, are hard to make sense of (an example is the existence of the square root of the logical not).

We will now list a set unitary operators, often referred to as **quantum gates**, commonly used when designing quantum algorithms:

Pauli Gates

These gates act on a single qubit and are represented by 2-by-2 matrices:

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

If we consider the qubit as a vector in a Bloch sphere, these operators correspond to rotations along the x, y, or z axes. Notably, the X gate is significant due to its resemblance to the logical not operation. Additionally, the Z gate is an important first example of unitary operator that affects the phase of the qubit.

Hadamard Gate

The Hadamard gate operates also on a single qubit and is of particular importance because it creates an equal superposition of state when applied to a basis state:

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Many quantum algorithms start with the application of an Hadamard gate, because to manipulate the quantum interference, we first need to put the state in a superposition, and the Hadamard gate does just that.

Control Not (CNOT) Gate

The CNOT gate acts on 2 qubits, with one as the control and the other as the target. The target qubit flips if the control qubit is $|1\rangle$:

$$\mathbf{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

One typical use case for this gate is to facilitate the creation of entangled states.

Phase Shift Gate

The phase shift gates map $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $e^{i\phi}|1\rangle$, introducing a phase change without altering the likelihood of measuring $|0\rangle$ or $|1\rangle$:

$$\mathbf{R}(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{\phi} \end{pmatrix}$$

The Pauli gate **Z** represents a phase shift of angle π. There are other shift gates that are used often and thus are named are:

$$\mathbf{S} = \mathbf{R}(\frac{\pi}{\mathbf{2}}) = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

$$\mathbf{T} = \mathbf{R}(\frac{\pi}{\mathbf{4}}) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

Toffoli Gate

The Toffoli gate, acting on 3 qubits, flips the target qubit only when both control qubits are in state $|1\rangle$:

$$\textbf{Toffoli} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

1.3 Measurement in Quantum Systems

As previously stated, the measurement of a qubit results in either $|0\rangle$ or $|1\rangle$. Let us delve deeper in the meaning of measurements in quantum computation.

Measurements are represented by Hermitian operators, known as *observables* in quantum computation. They represent the physical quantities one can detect (*observe*) in each state of the system. Given an observable, the only possible values that can be observed are its eigenvalues, which are always real values, due to their Hermitian nature.

The act of applying an observable to a state is a measurement, and the result of a measurement is a specific eigenvalue of the observable. Importantly, the measurement of a quantum state does not leave the state unchanged. The state after measurement will be an eigenvector corresponding to the observed eigenvalue.

The probability that a normalized state $|\phi\rangle$ will be found in a specific eigenvector $|e\rangle$ after measurement (and so the probability of having the corresponding eigenvalue as a result of the measurement) is given by the length squared of the projection of $|\phi\rangle$ over $|e\rangle$: $|\langle e|\phi\rangle|^2$.

Let us illustrate this with a qubit. Suppose we have a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. If we want to determine the probability of the qubit being in state $|0\rangle$ after measurement, we calculate:

$$|\langle 0|\psi\rangle|^2 = |\langle 0|(\alpha|0\rangle + \beta|1\rangle)|^2 = |\alpha\langle 0|0\rangle + \beta\langle 0|1\rangle|^2 = |\alpha|^2$$

This expression shows the connection between the probability amplitudes and the likelihood of observing a particular value, as discussed earlier.

1.3.1 Example: Bell State Construction

In summary, quantum computation involves setting up a state, manipulating it through unitary operations (quantum gates), and measuring the final state.

Let us illustrate these concepts by constructing a **Bell state**. Bell states represent a fundamental illustration of entangled states, specifically comprising a system of two maximally entangled qubits. The four distinct Bell states are defined as follows:

$$
\begin{aligned}
\left|\Phi^{+}\right\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\
|\Phi^{-}\rangle &= \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\
\left|\Psi^{+}\right\rangle &= \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\
|\Psi^{-}\rangle &= \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)
\end{aligned}
$$

To construct a Bell state, let us begin with a system of two qubits both initialized in the state $|0\rangle$. The first step is to apply a Hadamard gate to one qubit, placing it in a superposition. Since the Hadamard gate operates on a single qubit, we tensor it with the appropriate identity matrix for the second qubit:

$$
\begin{aligned}
\mathbf{H} \otimes \mathbf{I} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}
\end{aligned}
$$

Next, we apply this matrix to the 2-qubit state $|00\rangle$:

$$\mathbf{H} \otimes \mathbf{I} |00\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

Now, for the essential part of entanglement, we apply a **CNOT** gate with the first qubit as the control and the second qubit as the target:

$$\mathbf{CNOT} \frac{|00\rangle + |10\rangle}{\sqrt{2}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

This resulting state is the Bell state $\left|\Phi^{+}\right\rangle$. When measuring these two qubits, if the top qubit is measured as $|0\rangle$, the bottom qubit will also be $|0\rangle$; similarly, if the top qubit is observed as $|1\rangle$, the bottom qubit will also be $|1\rangle$.

Chapter 2

Quantum Circuit Description Languages

In the realm of quantum computing, two prominent computational models have emerged: the quantum Turing machine [10][7] and the quantum circuit [11][20]. While in the classical case, one could argue that the Turing machine is the best model to reason about computation, the quantum circuit model has gained more popularity in the field of quantum computing.

Although the Turing machine model provides a solid mathematical abstraction for discussing quantum computing, in retrospect it is considered quite cumbersome and not particularly well-suited for discussing algorithms. On the other hand, the quantum circuit model is more intuitive to use and, thanks to Yao [20], we know that it is equivalent to the Turing machine model in computational power. As a result, quantum circuits have become the preferred computational model in the exploration of quantum algorithms and complexity theory, effectively replacing quantum Turing machines.

Furthermore, quantum circuits present a model that can be more easily translated into a practical implementation on a quantum computers.

In this chapter, we provide an overview on how the quantum circuit model works. Subsequently, we focus on programming languages based on this model,

specifically Qiskit.

2.1 Quantum Circuits

The quantum circuit model is a natural quantum generalization of the classical circuit. A circuit involves input and output wires that carry information around, and gates that perform simple computational tasks. In the particular case of quantum circuits, each wire represents a qubit, and the gates are quantum gates, meaning that they represents a unitary operation with dimensions $2^n \times 2^n$, where n is the number input and output qubits. All the unitary operations discussed in the previous chapter can be used as quantum gates.

Furthermore, quantum circuits need to have some more particular features compared to classical circuits. First of all, the circuits need to be acyclic, meaning that feedback from one part of the circuit to another is not allowed. Second, fanin (when wires are joined together with a bitwise OR) is not allowed because it is not a reversible operation. Third, fanout (producing copies of a wire) is not allowed because of the no-cloning theorem, which states that it is impossible to create an identical and independent copy of a quantum state.

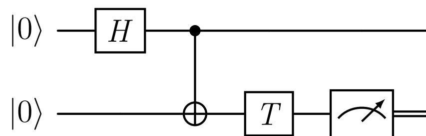
In the realm of classical logic gates, a small set of gates can be used to compute any classical function. Such small set of gates is then considered to be universal. In the case of quantum computation, we have a similar concept, where a small set of gates is said to be universal for quantum computation if, using the gates in the set, we can approximate any unitary operation to the desired level of accuracy. We talk about calculating them approximately because there exists an uncountably infinite number of gates, so it is not possible to represent them all perfectly with a countable set of gates. Some example of sets of universal quantum gates are: {CNOT, H, T} and {Toffoli, H}

Quantum circuits are conventionally represented using specific guidelines:

- Time progresses from left to right.
- Each qubit is depicted as a horizontal wire.

- Single-qubit gates are illustrated as labeled boxes and are positioned on the wire representing the qubit on which the gate operates.
- Controlled gates are denoted by a dot on the wire representing the control qubit, connected by a vertical line to the target gate. The widely used controlled-NOT (CNOT) gate deviates from the conventional box representation by depicting the NOT operation as the $\oplus$ symbol.
- A special gate for measurement is represented as a "meter" symbol

For illustrative purposes, the following example demonstrates how to depict a quantum circuit according to the established conventions:



2.2 Languages

To bridge the conceptual realm of quantum circuits with practical implementations, quantum circuit description languages play a pivotal role. These languages provide a standardized syntax and structure for representing quantum circuits, enabling seamless communication between theoretical quantum algorithms and real-world quantum processors. These languages span a spectrum from low-level assembly-like counterparts such as OpenQasm [9] to more abstract and high-level languages like Quipper [13].

One noteworthy tool that falls under the quantum description languages umbrella is Qiskit.

Qiskit (**Q**uantum **I**nformation **S**oftware Kit) is an open-source framework for working with quantum computers at the level of circuits, algorithms, and application modules [16], based on Python.

The project falls under the umbrella of the IBM quantum computing ecosystem. It is designed to work together with IBM's real quantum processors or quantum simulators available online on the IBM quantum platform [1].

The full documentation for Qiskit can be found at [4]. Here, we provide a brief overview of the tool.

2.3 Qiskit

The core workflow of Qiskit involves four steps:

- **Build**: design the quantum circuit to solve the problem at hand.
- **Compile**: compile the circuit for the specific service utilized.
- **Run**: run the compiled circuit on either local or cloud-based quantum services.
- **Analyze**: study the results of the experiments with visualizations and statistics.

For our purpose, we will focus on the build step of the workflow.

As previously stated, Qiskit is not an independent language, but it is implemented as a Python library. To import the entire Qiskit library, it is sufficient to do the following:

```
from qiskit import *
```

To access the simulation tools it is useful to include the following import:

```
from qiskit_aer import AerSimulator
```

The core of quantum computation is the quantum circuit. In Qiskit, this fundamental element is implemented by the `QuantumCircuit` class. A circuit in Qiskit is a list of instructions bound to some registers.

To create a new circuit, one can simply write the following:

```
circ = QuantumCircuit()
```

This creates a new circuit named `circ`. A circuit created in this way is not connected to any qubit or classical bit.

There are two ways to create a quantum circuit with the associated bits:

- Give the circuit a sequence of register objects, quantum registers and/or classical registers, to include in the circuit. For example:

  ```
  QuantumCircuit(QuantumRegister(4))
  QuantumCircuit(QuantumRegister(4), ClassicalRegister(3))
  QuantumCircuit(QuantumRegister(4, ‘qr0’), QuantumRegister(2,
      ‘qr1’))
  ```

  `QuantumRegister` and `ClassicalRegister` are the classes that implement quantum and classical registers, respectively. `QuantumRegister(n, ‘qr’)` creates a register with n qubits called `qr`, and `ClassicalRegister(n, ‘cr’)` creates a register with n bits called `cr`.

- Give the circuit a sequence of integers, which indicates the amount of qubits and/or classical bits to include in the circuit. If only one number is given, it indicates the number of qubits in the circuit. If there are two numbers, they indicate the number of qubits and classical bits, respectively. For example:

  ```
  QuantumCircuit(4)       # A QuantumCircuit with 4 qubits
  QuantumCircuit(4, 3)    # A QuantumCircuit with 4 qubits
                          # and 3 classical bits
  ```

The registers can be added to the circuit at a later time using the method `add_register`:

```
circ = QuantumCircuit(QuantumRegister(4, ‘qr0’), QuantumRegister
    (2, ‘qr1’))
```

is the same as:

```
circ2 = QuantumCircuit()
circ2.add_register(QuantumRegister(4, ’qr0’))
circ2.add_register(QuantumRegister(2, ’qr1’))
```

Qiskit provides an implementation of a collection of well-studied and valuable gates in the library `qiskit.circuit.library`. A comprehensive list of these gates can be found in [5]. Each element can be plugged into a circuit using the `QuantumCircuit.append()` method, but there is an easier way to add gates to the circuit. Most gates can be accessed as methods of the `QuantumCircuit` class. So, for example, to build a simple Bell state circuit, one can simply do:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
```

We have created a quantum circuit with two qubits, and we apply the H gate to the first qubit. Then, we apply the controlled-not gate with the first qubit as the control and the second qubit as the target.

The command to measure qubits is implemented as a method of the quantum circuit: `QuantumCircuit.measure(qubit, cbit)`.

This operation performs a measurement of a qubit along the Z-axis and stores the result in a classical bit. This operation is non-reversible.

The parameters of `measure` can be either a single qubit and a classical bit or a list of qubits and classical bits. In the second case, the number of qubits and classical bits must be the same. Let us see an example where we perform the measurement of a simple Bell state:

```
qc = QuantumCircuit(2, 2)
qc.h(0)
```

```
qc.cx(0, 1)
qc.measure([0, 1], [0, 1])
```

It is possible to apply a gate on the condition that a specific value is stored in a classical register. This can be achieved via the `InstructionSet.c_if()` method. For example:

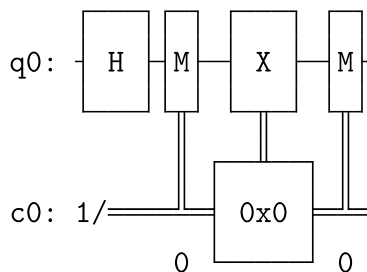
```
qr = QuantumRegister(1)
cr = ClassicalRegister(1)
qc = QuantumCircuit(qr, cr)
qc.h(0)
qc.measure(0, 0)

qc.x(0).c_if(cr, 0)
qc.measure(0, 0)
```

The X gate will be executed only if we find the value 0 in the classical register.

Qiskit provides a way to visualize the designed circuit. It offers various formats of visualization that are used in many textbooks and research articles. To see the default visualization, we use the following command:

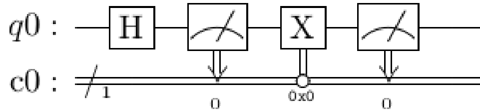
```
qc.draw()  # qc is the circuit defined in the previous
           # examples.
```

This will produce a text-based representation:



There are many other circuit drawer backends, for example, for the LaTeX backend:

```
qc.draw(output="latex")
```



After building the circuit, it is time to compile it and run it on the desired quantum service. Qiskit provides many different backends, from real quantum devices accessible through the cloud [1] to local simulators. Let us see how to compile and run our example on the Qiskit Aer simulator.

```
simulator = AerSimulator()
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit, shots=1000)
result = job.result()
counts = result.get_counts(qc)
```

The transpile function compiles the circuit (`qc`) for the specified backend (`simulator`). It is possible to indicate how many times the circuit is to be executed by using the `shots` argument of the `run` method. In this case, the simulation was configured to run 1000 shots, with the default being 1024. After obtaining a `result` object, one can retrieve the count data by using the `get_counts(qc)` method, which provides a summary of the experiment's collective outcomes.

If one desires to incorporate the outcomes of the circuit's execution in the context of a more intricate algorithm by accessing the individual results of the experiment, the `get_memory()` function can be employed. For instance, to access the results of the experiment in the previous examples, the following steps can be taken:

```
simulator = AerSimulator()
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit, shots=1000, memory=True)
result = job.result()
memory = result.get_memory()
```

Furthermore, by changing the simulator, we can have better insight on the circuit we are building. For example, we can visualize the state vector after it is transformed by the quantum operations described in the circuit. To do so, we can do the following.

```
simulator = Aer.get_backend('statevector_simulator')
result = execute(qc, backend=simulator).result()
statevector = result.get_statevector()
```

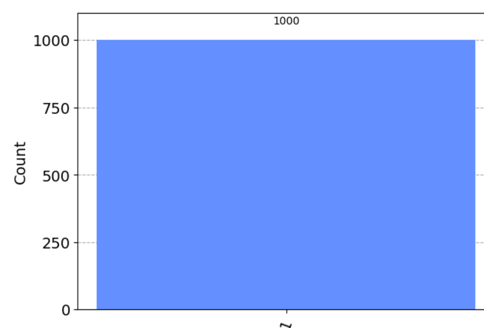
Another useful tool in understanding the circuit is the `unitary_simulator` , which allows us to visualize the unitary matrix resulting from the composition of gates used in our circuit:

```
simulator = Aer.get_backend('unitary_simulator')
result = execute(qc, backend=simulator).result()
unitary = result.get_unitary()
```

It is important to note that, in order to make this command work, we cannot have measurements in the circuit.

Qiskit provides many ways to visualize the results of the experiments through the `qiskit.visualization` library. An example is the `plot_histogram` function, to view the results:

```
plot_histogram(counts)
```



Chapter 3

Qiskit with Dynamic Lifting

In the last few years, IBM has equipped their quantum computers with methods to allow interactions with classical computing instructions during the execution of quantum circuits. This development has led to the introduction of what IBM calls "dynamic quantum circuits" [6]. As a result of this enhancement, the quantum circuit model now incorporates features such as mid-circuit measurement, mid-circuit reset, and classically-controlled quantum operations.

Examining the implementation of these concepts at the programmatic level within Qiskit reveals a serviceable yet somewhat cumbersome approach. Our research endeavors to seek a more refined and elegant method of incorporating these functionalities. To achieve this objective, we draw inspiration from a feature prevalent in other quantum programming languages, known as dynamic lifting.

The term "dynamic lifting" pertains to the capability of accessing the intermediate state of a qubit during the circuit construction process, usually in the form of a boolean value. Notably, dynamic lifting is a feature inherent to Quipper [13], a functional quantum language, and it is subjected to more precise and comprehensive formalization within select Proto-Quipper programming languages [12].

3.1 Language

We opt for a limited implementation of dynamic lifting, centering our efforts on the capability to enable the conditional application of gates based on the outcome of a measurement.

To achieve this, we introduce the following components to the Python language and the Qiskit library:

- A new class of object `LiftedValue`
- A new command `ifdl`
- Establishment of a default global circuit.

3.1.1 `LiftedValue`

The `LiftedValue` class serves as the conduit for accessing values during circuit construction. The whole implementation can be found in Appendix A.

The objects that can be lifted are Qiskit's `Clbit` s. To lift a `Clbit` , one must instantiate a new `LiftedValue` , providing the targeted `Clbit` as an argument:

```
c = Clbit()
lifted_value = LiftedValue(c)
```

In this way, `lifted_value` can be used as a condition.

The `LiftedValue` class implements some private methods to allow for the use of Boolean expressions on lifted values. Specifically, the `LiftedValue` class overloads the `&` , `|` , and ∼ Python symbols (utilized in Python to implement the built-in bitwise *and*, *or*, and *not* operations, respectively) to facilitate the AND, OR, and NOT Boolean operations on lifted values.

Since Qiskit does not provide any instrument to manipulate classical values, the Boolean instruction is implemented through a quantum circuit. We will use the expression `l1 & l2` (where `l1` and `l2` are lifted values) to illustrate an example of the implementation of these circuits.

First, we create a sub-circuit with 3 qubits and 1 classical bit: two qubits for the input values of the expression, one qubit for the result of the expression, and a classical bit for storing the measurement of the result. The two input values are initialized to the values of `l1` and `l2`, respectively, while the third qubit is set to $|0\rangle$. We then apply the gate implementing the quantum Boolean AND provided by Qiskit to the three qubits[1]. In the end, we measure the result qubit, storing the result in the classical bit.

This sub-circuit is appended to the global circuit, and the `Clbit` storing the result of the expression is lifted, creating a new lifted value.

In the `LiftedValue` class, we can find the implementation of the `__enter__` and `__exit__` methods used to manage the implementation of the `ifdl`. We will go into more detail about them in the next section.

3.1.2 `ifdl`

Since in Python, the host language of Qiskit, it is not possible for the user to modify the behavior of the `if` construct on specific objects, we need to introduce a new command. We refer to this command as `ifdl`, and it is used to implement an if-like statement that allows the conditioning of the building of circuits on lifted values.

Specifically, the aim is to define a block of code influenced by the lifted value. Additionally, there is a need to implement instructions for setup before the conditioned block and instructions for cleanup after the conditioned block.

We want to minimize changes to the Python compiler and perform much of the work through libraries. To achieve this, the implementation of `ifdl` follows the paradigm of the native Python `with` statement [2], utilizing the `LiftedValue` objects as a context manager. A context manager is an object that defines the `__enter__` and `__exit__` methods. The `__enter__` function executes before the block of code identified by the `ifdl` statement, while the `__exit__` function executes after the block of code.

[1] In the AND operation the state $|1\rangle$ is interpreted as true. The result qubit is flipped if the two input qubits are true [16].

An illustrative example elucidating the functionality of `ifdl` is presented below:

```
ifdl(lv):               # lv is a lifted value
        X(q)            # X is the x Pauli gate, q is a qubit
```

The execution of the `ifdl` statement unfolds as follows:

- The lifted value (`lv`) is evaluated to obtain a context manager. If the guard of the `ifdl` includes a lifted value expression (e.g., `l1 & l2`), the circuit evaluating the expression is appended to the global circuit. The resulting lifted value represents the outcome of the expression and becomes the guard of the `ifdl` .
- The `__enter__` method of the lifted value is invoked:
  - A flag named `dl_flag` is set to `True` to indicate the scope of the `ifdl` .
  - The lifted value is added to the list of active lifted values (`lifted_values`) in the circuit.
- The body of the `ifdl` (`X(q)`) is executed.
- The `__exit__` method of the lifted value is invoked:
  - The last lifted value inserted in the `lifted_values` list is removed.
  - If there are no more active lifted values in the circuit, the `dl_flag` is set to `False` to signify the end of the `ifdl` scope.

The `dl_flag` is checked each time a gate is applied to the global circuit. When the flag is set to `True` , it means that the circuit application is occurring within the body of an `ifdl` statement. Consequently, this operation is conditioned on the lifted values found in (`lifted_values`). The conditionality is practically implemented using the `.c_if` construct provided by Qiskit. If the flag is `False` , it means that we are not within the scope of `ifdl` statement, and we proceed with the standard application of the gate.

3.1.3 Global Circuit

In Qiskit, the introduction of an implicit global quantum circuit streamlines the process of writing quantum programs and facilitates the implementation of dynamic lifting, aligning with the approach adopted by other quantum programming languages like Quipper.

When implementing the `ifdl` command, access to two key elements is essential:

- The `dl_flag`, indicating the scope of the `ifdl` command.
- The list of active lifted values in the circuit (`lifted_values`).

These parameters are realized as global variables, consistently referencing the global circuit when applicable.

To enhance user convenience, a wrapper is implemented for quantum circuit operations. This wrapper enables the use of these operations in the code without an explicit reference to a particular circuit. For instance, the syntax of the application of a gate transitions from `circuit.h(q)` to simply `H(q)`, where the target circuit is implicitly the global one. Additionally, the wrapper checks for the `dl_flag` to ensure that operations are conditioned only on the global circuit when lifted values are involved. The code for the implementation of the wrapper can be found in Appendix A.

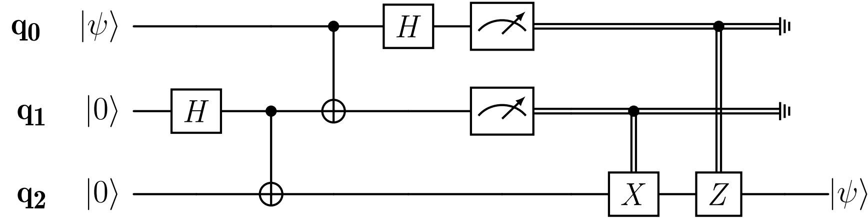
These changes contribute to a more streamlined and user-friendly quantum programming experience in Qiskit.

3.2 Examples

In the following section, we present the implementation of two simple algorithms, both in basic Qiskit and with dynamic lifting.

3.2.1 Quantum Teleportation

The quantum teleportation protocol is a straightforward procedure enabling the transfer of the state of an arbitrary qubit from one location to another. The circuit that implements this protocol is as follows:



To illustrate the functioning of the protocol, let us consider a scenario involving two individuals, Alice and Bob, situated in different locations. Their goal is to transfer a quantum bit, denoted as $|\psi\rangle$, from Alice to Bob. The protocol operates as follows:

- Initially, a Bell state involving two qubit q_1 and q_2 is generated, and q_1 is sent to Alice, while q_2 is sent to Bob.
- Alice performs a Bell measurement on the qubit she intends to transfer, $|\psi\rangle$, and the Bell state qubit in her possession, q_1. This measurement produces one of three possible values, which can be encoded using two classical bits.
- The classical information obtained from Alice's measurement is then transmitted classically to Bob.
- Due to the phenomenon of entanglement, Bob's Bell state qubit, q_2, undergoes a change based on Alice's measurement. Consequently, Bob can reconstruct the state of $|\psi\rangle$ by applying specific operations to q_2, guided by the classical values received from Alice.

The following figures depict the corresponding circuit in Qiskit and the Qiskit implementation with dynamic lifting.

```
qr = QuantumRegister(3)
crz = ClassicalRegister(1)
crx = ClassicalRegister(1)

q_teleport = QuantumCircuit(qr, crz, crx)

q_teleport.h(qr[1])
q_teleport.cx(qr[1], qr[2])
q_teleport.cx(qr[0], qr[1])
q_teleport.h(qr[0])

q_teleport.measure(qr[0], crz)
q_teleport.measure(qr[1], crx)

q_teleport.x(2).c_if(crx, 1)
q_teleport.z(2).c_if(crz, 1)
```

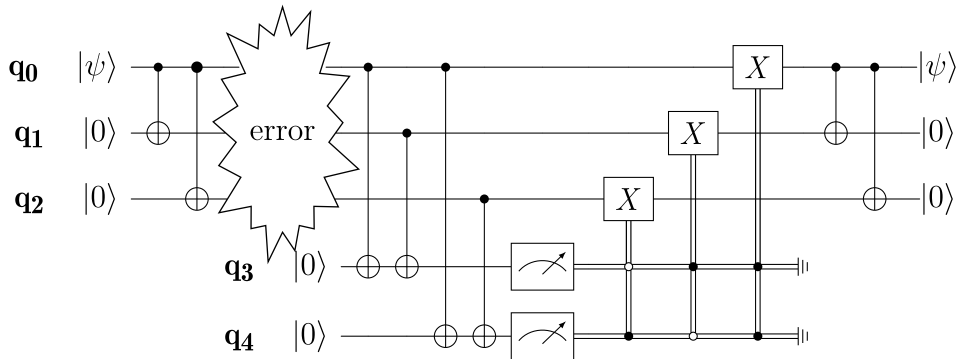
Program 3.1: Quantum Teleportation in Qiskit

```
qr = QuantumRegister(3)
crz = ClassicalRegister(1)
crx = ClassicalRegister(1)
lift_crz= LiftedValue(crz)
lift_crx = LiftedValue(crx)

addRegisterGlobal(cl_reg, crz, crx)

H(qr[1])
Cx(qr[1],qr[2])
Cx(qr[0],qr[1])
H(qr[0])
Measure(qr[0], crz)
Measure(qr[1], crx)

ifdl (lift_crx):
    X(qr[2])
ifdl (lift_crz):
    Z(qr[2])
```

Program 3.2: Quantum Teleportation in Qiskit with dynamic lifting

3.2.2 Error Correction

Quantum error correction is essential for safeguarding quantum information against errors arising from decoherence and other sources of quantum noise. We present a straightforward error correction protocol designed for bit flip errors, employing a three-qubit repetition code. The circuit implementing this protocol is illustrated below:



We begin with an unknown quantum state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$. This quantum state is encoded using three qubits, q_0, q_1 and q_2:

- q_0 is set to the value of $|\phi\rangle$.
- We apply the CNOT operations with q_0 as control and q_1 as target.
- We apply the CNOT operations with q_0 as control and q_2 as target.

This results in the entangled state $\alpha|000\rangle + \beta|111\rangle$.

Subsequently, a moment of decoherence occurs, introducing the possibility of four potential errors:

- No error,

- Bit flip error on q_0,
- Bit flip error on q_1,
- Bit flip error on q_2.

Detecting and correcting these errors requires the introduction of two auxiliary qubits, q_3 and q_4. The detection happens through the application of four CNOT operations:

1. Two CNOT operations with q_3 as target, when the control is first q_1 and then q_2,
2. Two CNOT operations with q_4 as target, while the control is first q_1 and then q_2.

The measurement of q_3 yields 0 if q_0 and q_1 are in the same state. Similarly, the measurement of q_4 yields 0 if q_0 and q_2 are in the same state. If both auxiliary qubits measure zero, no error has occurred; otherwise, an error is indicated. Once the error type is detected, the corrupted qubit is flipped, and the three-qubit system is decoded using the inverse operation of the encoding to restore the original state before decoherence.

The figures below illustrate the circuit in Qiskit and in Qiskit with dynamic lifting.

```
q= QuantumRegister(5)
c = ClassicalRegister(2)

bit_flip = QuantumCircuit(q, end_m, bit_check)

bit_flip.cx(q[0], q[1])
bit_flip.cx(q[0], q[2])

# Here is what we want to protect

bit_flip.cx(q[0], q[3])
bit_flip.cx(q[1], q[3])
```

```
bit_flip.cx(q[0], q[4])
bit_flip.cx(q[2], q[4])

bit_flip.measure(q[3], c[0])
bit_flip.measure(q[4], c[1])

bit_flip.x(q[0]).c_if(c, 0b11)
bit_flip.x(q[1]).c_if(c, 0b01)
bit_flip.x(q[2]).c_if(c, 0b10)

bit_flip.cx(q[0], q[1])
bit_flip.cx(q[0], q[2])
```

Program 3.3: Error Correction in Qiskit

```
q= QuantumRegister(5)
c = ClassicalRegister(2)
lift0= LiftedValue(c[0])
lift1= LiftedValue(c[1])

addRegisterGlobal(q, c)

Cx(q[0], q[1])
Cx(q[0], q[2])

# Here is what we want to protect

CX(q[0], q[3])
Cx(q[1], q[3])
Cx(q[0], q[4])
Cx(q[2], q[4])

Measure(q[3], c[0])
Measure(q[4], c[1])

ifdl (lift0 & lift1):
    X(q[0])
```

```
ifdl (lift0 & (~lift1)):
    X(q[1])
ifdl ((~lift0) & lift1):
    X(q[1])

Cx(q[0], q[1])
Cx(q[0], q[2])
```

Program 3.4: Error Correction in Qiskit with dynamic lifting

Chapter 4

The Operational Semantics of Dynamic Lifting

In this chapter, our primary objective is to establish a formal foundation for the previously discussed work. We aim to treat quantum components as integral parts of the language, not merely as a library. To build a comprehensive formal framework, we adopt a simplified version of Python, formalizing both its syntax and operational semantics.

Furthermore, we delve into the formalization of the construction of quantum circuits. While we provide formal structures for both Python and quantum components, our emphasis is directed towards the intricacies of the quantum circuit description process, with particular emphasis on dynamic lifting.

4.1 Syntax

In simplifying the Python language for our purpose, we retain essential objects such as arithmetic expressions, boolean expressions, assignments, `if` and `while` commands, and concatenation of commands. For the incorporation of quantum aspects, we introduce specialized objects representing qubits, classical bits, and unitary operations. Subsequently, we establish mechanisms for incorporating qubits

and bits into the circuit, along with constructs for lifted values and their integration into the circuit. Notably, the language features the `ifdl` construct, enabling conditional statements based on lifted values, and includes a measurement operation.

The quantum circuit objects, namely `Qubit`, `Clbit`, and `LiftedValue`, are treated as native objects. To achieve this, we introduce a set of indices for referencing qubits, bits, and lifted values. The other fundamental objects manipulated by the language include numbers, boolean values, and variables (representing memory locations).

In summary, the fundamental objects manipulated by the language belong to the following sets:

- $\mathscr{N}$: the set of positive and negative numbers, including zero
- $\mathscr{B}$: the set of boolean values: $\{\texttt{True}, \texttt{False}\}$
- $\mathscr{V}$: the set of variables. It consists of non-empty strings form by valid characters for Python identifiers: the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9 [3].
- $\mathscr{I}$: a set of indices. elements of this set will be used to refer to qubits, bits and lifted values.

The following metavariables will be employed in the description of the syntax and, subsequently, in the discussion of the operational semantics. Each of these metavariables can be primed or subscripted to refer to different objects.

- n ranges over numbers in $\mathscr{N}$
- x ranges over variables in $\mathscr{V}$
- i to range over elements of $\mathscr{I}$
- t ranges over boolean values

- a ranges over arithmetic expressions
- b ranges over boolean expressions
- c ranges over commands
- qb ranges over qubits
- cl ranges over classical bits
- lv ranges over lifted values
- lv_ex ranges over lifted values expressions
- U ranges over unitary operators.

The full syntax of the language is in Figure 4.1.

Let us emphasize the commands designed for quantum computation.

The commands `Qubit()` and `Clbit()` serve as foundational commands for instantiating qubits and classical bits. The command `LiftedValue(`clb`)` enables the lifting of classical values. Once lifted, these values can be used to perform boolean operations through expressions denoted as lv_ex. These expressions can be assigned to variables, allowing for convenient access through variable names.

Conditional statements on lifted values are implemented using the `ifdl` command, whose guards are lifted variable expressions (lv_ex). It is essential to note that within the body of `ifdl`, all commands are applicable, but the conditioning specifically applies to quantum gates. For example, it is possible to write:

```
ifdl(lv):
    X(q)
    v = 42
```

But only the command `X(q)` will be conditioned on `lv`, while `v = 42` will not.

A set of commands is introduced for adding qubits, bits, and lifted values to the circuit we are describing: `addQubit(`qb`)`, `addClbit(`clb`)`, `addLiftedValue(`lv`)`.

$a ::= n \mid x \mid a_0\texttt{+}a_1 \mid a_0\texttt{-}a_1 \mid a_0\texttt{*}a_1 \mid \texttt{-}a \mid a_0\texttt{/}a_1 \mid a_0\texttt{\%}a_1$
$\quad\mid a_0\texttt{//}a_1 \mid a_0\texttt{**}a_1$

$b ::= \texttt{True} \mid \texttt{False} \mid x \mid a_0\texttt{==}a_1 \mid a_0\texttt{!=}a_1 \mid a_0\texttt{<}a_1 \mid a_0\texttt{>}a_1$
$\quad\mid a_0\texttt{<=}a_1 \mid a_0\texttt{>=}a_1 \mid b_0 \texttt{ and } b_1 \mid b_0 \texttt{ or } b_1 \mid \texttt{not } b_0$

$lv ::= \texttt{LiftedValue(}clb\texttt{)} \mid x$

$lv_ex ::= lv \mid lv_ex_1\texttt{\&}lv_ex_2 \mid lv_ex_1\texttt{|}lv_ex_2 \mid \sim lv_ex$

$qb ::= \texttt{Qubit()} \mid x$

$clb ::= \texttt{Clbit()} \mid x$

$c ::= x\texttt{=}a \mid x\texttt{=}b \mid x\texttt{=}qb \mid x\texttt{=}clb \mid x\texttt{=}lv$
$\quad\mid c_0\ \ c_1$
$\quad\mid \texttt{if } b\texttt{:} c_0\ \ \texttt{else:} c_1$
$\quad\mid \texttt{while } b\texttt{ :}\quad c$
$\quad\mid \texttt{addQubit(}qb\texttt{)}$
$\quad\mid \texttt{addClbit(}clb\texttt{)}$
$\quad\mid \texttt{addLiftedValue(}lv\texttt{)}$
$\quad\mid \texttt{ifdl } lv_ex\texttt{:}\quad c$
$\quad\mid U\texttt{(}qb_0, \ldots, qb_n\texttt{)}$
$\quad\mid \texttt{Measure(}qb, clb\texttt{)}$

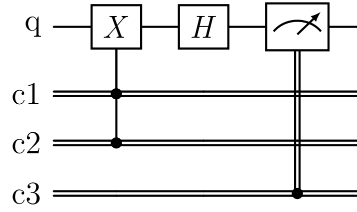
Figure 4.1: Full syntax.

Moreover, the command $U(qb_0, \ldots, qb_n)$, represents the application of the primitive quantum unitary operation U to qubits $qb_0, \ldots, qb_n$. The set of unitary operation which U ranges over is assumed to contain all gates and transformations which are usually employed when describing quantum algorithms. For example, for the single qubit Hadamard gate, we will have H(q_0). Finally, the Measure(qb, clb) command implements the measurement of qubit qb, whose outcome is stored in bit clb.

To provide a practical illustration, consider the following example of a quantum circuit constructed in accordance with the presented syntax:

```
c1 = Clbit()
c2 = Clbit()
c3 = Clbit()
q = Qubit()

l1 = LiftedValue(c1)
l2 = LiftedValue(c2)

addClbit(c1)
addClbit(c2)
addClbit(c3)
addQubit(q)
addLiftedValue(l1)
addLiftedValue(l2)

ifdl(l1 & l2):
    X(q)
H(q)

Measure(q,c3)
```

Which describes the circuit



4.2 Operational Semantics

In discussing the operational semantic, the context is given as a tuple $(\sigma, \kappa, \mathcal{I})$, where σ represents the memory state, κ the global quantum circuit and $\mathcal{I}$ is the set of lifted values active in the scope.

A configuration is a pair $\langle e, (\sigma, \kappa, \mathcal{I})\rangle$ where e is the expressions we want to evaluate and $(\sigma, \kappa, \mathcal{I})$ is the context we want to evaluate the expressions in.

The state σ is a map from variable names to values. It can be seen as a function $\sigma : \mathscr{V} \rightarrow \mathit{Value}$, where $\mathscr{V}$ is the set of strings indicating a variables and the values can be numbers, Boolean values or indexes: $\mathit{Value} = \mathscr{N} \cup \mathscr{B} \cup \mathscr{I}$. Therefore, writing $\sigma(x)$ will give us the value corresponding to the variable x in state σ.
The circuit κ is represented by a tuple $(\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv})$:

- In is a sequence of instructions
- Qb is the set of qubits that can be used in the circuit
- Clb is the set of classical bits that can be used in the circuit
- Lv is the set of lifted values available.

Formally:

$$\kappa \in \mathit{Instr}^* \times \mathcal{P}_{\mathit{fin}}(\mathscr{I}) \times \mathcal{P}_{\mathit{fin}}(\mathscr{I}) \times \mathcal{P}_{\mathit{fin}}(\mathscr{I}).$$

The set Instr is defined as follow:

$$Instr = String \times \mathscr{I}^* \times \mathscr{I}^* \times \mathscr{I}^*.$$

Its elements are tuples of the form $(\mathsf{Name}, \mathsf{qbs}, \mathsf{clbs}, \mathsf{lvs})$:

- Name represents the name (or identifier) of the instruction
- qbs represents the qubits on which the instruction acts
- clbs the bit on which the instruction acts
- lvs represents the lifted values the instruction maybe depend on.

Lastly, $\mathcal{I}$ is the least complex element of the context: it is a simple subset of indices which represent the lifted values active in the scope.

The evaluation of arithmetic and Boolean expressions does not significantly modify the context, in particular they interact only with the memory state σ.

For the arithmetic expressions, we have the following evaluation relation between configuration and numbers:

$$\langle a, (\sigma, \kappa, \mathcal{I})\rangle \to n$$

meaning that the arithmetic expression a evaluates to the number n in the $(\sigma, \kappa, \mathcal{I})$ context.

When evaluating, for example, the expression a_0+a_1, we first evaluate a_0 to get the number n_0, then we evaluate a_1 to get the number n_1, and then we add n_0 and n_1 to get $n = n_0 + n_1$ as the result of the evaluation of a_0+a_1.

Something similar happens for boolean expressions: $\langle b, (\sigma, \kappa, \mathcal{I})\rangle \to t$ means that the boolean expression b evaluates to the boolean value t in the $(\sigma, \kappa, \mathcal{I})$ context.

The complete set of rules for the evaluation of arithmetic and boolean expressions can be found in Figure 4.2.

The evaluation of single qubit, classical bit and lifted value similarly returns a single value, in this case an index : $\langle qb, (\sigma, \kappa, \mathcal{I})\rangle \to i$, $\langle clb, (\sigma, \kappa, \mathcal{I})\rangle \to i$ and

$\langle lv, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i$. In particular, the rules for instantiating qubits and classical bits are the following:

$$\frac{}{\langle \texttt{Qubit()}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow \mathit{next}(\kappa)}\ \mathit{qubit} \qquad \frac{}{\langle \texttt{Clbit()}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow \mathit{next}(\kappa)}\ \mathit{clbit}$$

Where we employ the auxiliary function $\mathit{next}(\kappa)$ to obtain the next index from $\mathscr{I}$ that has not yet been utilized in κ, which identifies the new qubit or bit.

A lifted value always originates from a classical bit, so the corresponding rule is:

$$\frac{\langle \mathit{clb}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}{\langle \texttt{LiftedValue(}\mathit{clb}\texttt{)}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}\ \mathit{dynlift}$$

The evaluation of a variable produces a single value as well. The rule is the following:

$$\frac{}{\langle x, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow \sigma(x)}\ \mathit{var}$$

Lifted values can be used to form lifted value expressions $\mathit{lv_ex}$. These expressions exhibit behavior similar to boolean expressions but operate on lifted values. To convey a semantic equivalent to the expressions' behavior in the implementation described in the previous chapter, we utilize a subcircuit to describe the logical operation.

The evaluation rule for lifted value expressions does not produce a single value, but rather a tuple:

$$\langle \mathit{lv_ex}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i, \kappa_m)$$

where i is a new lifted value that depends on the quantum logical operation, and κ_m is the subcircuit that implements the quantum logical operation.

The semantics of lifted value expressions is as follows:

$$\frac{\langle lv, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}{\langle lv, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i, \epsilon)}\ lv$$

$$\frac{\langle lv_ex_0, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i_0, \kappa_0) \qquad \langle lv_ex_1, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i_1, \kappa_1)}{\langle lv_ex_0 \& lv_ex_1, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow rev_and((i_0, \kappa_0), (i_1, \kappa_1))}\ lv_and$$

$$\frac{\langle lv_ex_0, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i_0, \kappa_0) \qquad \langle lv_ex_1, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i_1, \kappa_1)}{\langle lv_ex_0 \,|\, lv_ex_1, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow rev_or((i_0, \kappa_0), (i_1, \kappa_1))}\ lv_or$$

$$\frac{\langle lv_ex, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i, \kappa_0)}{\langle {\sim} lv_ex, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow rev_not((i, \kappa_0))}\ lv_not$$

Where ϵ denotes an empty circuit, which is needed to facilitate the use of lv in lifted values expressions. The auxiliary functions rev_and, rev_or and rev_not, are employed to implement their respective quantum logical operations using Toffoli gates. Their return value is a tuple (i, κ_m)

The evaluation of commands involves modifying the context, so, given a configuration $\langle c, (\sigma, \kappa, \mathcal{I})\rangle$, where c is the command to be executed, we define the relation

$$\langle c, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma', \kappa', \mathcal{I}')$$

which intuitively means that the execution of the command c in context $(\sigma, \kappa, \mathcal{I})$ terminates in the final context $(\sigma', \kappa', \mathcal{I}')$.

The first commands we analyze are the assignment commands x=a, x=b, x=qb, x=clb, x=lv. These commands update the memory state, so, for example we have

$$\langle x{=}42, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[42/x], \kappa, \mathcal{I})$$

where $\sigma[42/x]$ is the memory state updated with the association between x and 42.

The rules that describe all the assignment commands are the following:

$$\frac{\langle a, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow n}{\langle x\texttt{=}a, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[n/x], \kappa, \mathcal{I})} \quad \frac{\langle b, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow t}{\langle x\texttt{=}b, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[t/x], \kappa, \mathcal{I})}$$

$$\frac{\langle qb, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}{\langle x\texttt{=}qb, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[i/x], \kappa, \mathcal{I})} \quad \frac{\langle cl, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}{\langle x\texttt{=}cl, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[i/x], \kappa, \mathcal{I})}$$

$$\frac{\langle lv, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i}{\langle x\texttt{=}lv, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma[i/x], \kappa, \mathcal{I})}$$

Next, we have a set of commands that modify only the circuit in the context: `addQubit(`qb`)`, `addClbit(`clb`)` and `addLiftedValue(`lv`)`. These fundamental components operate in a similar fashion, they evaluate the object to be added to the circuit and then they modify the circuit by inserting the corresponding index at the appropriate location.

$$\frac{\langle qb, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow i}{\langle \texttt{addQubit(}qb\texttt{)}, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow (\sigma, (\mathit{In}, \mathit{Qb} \cup \{i\}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})}$$

$$\frac{\langle clb, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow i}{\langle \texttt{addClbit(}clb\texttt{)}, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb} \cup \{i\}, \mathit{Lv}), \mathcal{I})}$$

$$\frac{\langle lv, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow i}{\langle \texttt{addLiftedValue(}lv\texttt{)}, (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv}), \mathcal{I})\rangle \rightarrow (\sigma, (\mathit{In}, \mathit{Qb}, \mathit{Clb}, \mathit{Lv} \cup \{i\}), \mathcal{I})}$$

Next, we focus on the `ifdl` command. Initially, we evaluate the guard (lv_ex). This evaluation will return an index, denoted as i_0, which corresponds

to the lifted value holding the result of lv_ex, and the circuit κ_0 implementing the logical operation. Both elements will be added to the context used for the evaluation of the body of `ifdl`. Specifically, κ_0 will be added to the circuit, while i_0 will be added to $\mathcal{I}$ (the set of active lifted values). This new $\mathcal{I}$ set is particularly important when in the body of `ifdl` we have the unitary operator. In such cases, the unitary is conditioned upon the lifted values in $\mathcal{I}$.

The operational semantic rule for `ifdl` is outlined as follows:

$$\frac{\langle lv_ex, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (i_0, \kappa_0) \qquad \langle c, (\sigma, \kappa + \kappa_0, \mathcal{I} \cup \{i_0\})\rangle \rightarrow (\sigma', \kappa', \mathcal{I}')}{\langle \texttt{ifdl}\, lv_ex\texttt{:}\;\; c, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma', \kappa', \mathcal{I})}\ ifdl$$

where $\kappa + \kappa_0$ is to be interpreted as: if $\kappa = (In', Qb', Clb', Lv')$ and $\kappa_0 = (In'', Qb'', Clb'', Lv'')$, then $\kappa + \kappa_0 = (In' :: In'', Qb' \cup Qb'', Clb' \cup Clb'', Lv' \cup Lv'')$ (we use the :: symbol to indicate the concatenation in the sequence of instructions).

Pay particular attention to how the context is modified in the conclusion of the rules: $\mathcal{I}$ remains the same as before the application of `ifdl`, ensuring that only the quantum operation inside the body of `ifdl` can be conditioned on the values introduced in the guard of the `ifdl`.

Next, we want to highlight the command that allows the application of a unitary operation $U\texttt{(}qb_0, \ldots, qb_n\texttt{)}$. Its semantics involves adding a new instruction to the circuit.

$$\frac{\langle qb_0, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i_0 \qquad \ldots \qquad \langle qb_n, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow i_n}{\langle U\texttt{(}qb_0, \ldots, qb_n\texttt{)}, (\sigma, \kappa, \mathcal{I})\rangle \rightarrow (\sigma, \kappa', \mathcal{I})}\ U$$

where, if $\kappa = (In, Qb, Clb, Lv)$ then $\kappa' = (In :: (U, [i_0, \ldots, i_n], \emptyset, \mathcal{I}), Qb, Clb, Lv)$.

The last command necessary for the description of a quantum circuit is the measurement. Its semantics is the following:

$$\frac{\langle qb, (\sigma, \kappa, \mathcal{I})\rangle \to i_0 \qquad \langle clb, (\sigma, \kappa, \mathcal{I})\rangle \to i_1}{\langle \texttt{Measure(}qb, clb\texttt{)}, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma, \kappa', \mathcal{I})} \; meas$$

where, if $\kappa = (\mathcal{In}, \mathcal{Qb}, \mathcal{Clb}, \mathcal{Lv})$ then $\kappa' = (\mathcal{In} :: \{(\texttt{Measure}, [i_0], [i_1], \emptyset)\}, \mathcal{Qb}, \mathcal{Clb}, \mathcal{Lv})$.

The semantics for the remaining commands, pertaining to the classical aspects of computation, can be found in Figure 4.3.

4.2.1 An Example

We will now give an example of derivation of the code described in section 4.1.

In the beginning we have a moment of definition and assignment of qubits and classical bit.

```
c1 = Clbit()
c2 = Clbit()
c3 = Clbit()
q = Qubit()
```

After this, the context will be: $env = (\sigma[i_1/\texttt{c1}][i_2/\texttt{c2}][i_3/\texttt{c3}][i_4/\texttt{q}], \epsilon, \emptyset)$.

Next, we have the dynamic lifting of the two classical bits `c1` and `c2`.

```
l1 = LiftedValue(c1)
l2 = LiftedValue(c2)
```

To illustrate, an example of the derivation for the first lifting can be outlined as follows:

$$\frac{\overline{\langle \texttt{LiftedValue(c1)}, (\sigma[i_1/\texttt{c1}][i_2/\texttt{c2}][i_3/\texttt{c3}][i_4/\texttt{q}], \epsilon, \emptyset)\rangle \to i_1}}{\langle \texttt{l1 = LiftedValue(c1)}, env\rangle \to (\sigma[i_1/\texttt{c1}][i_2/\texttt{c2}][i_3/\texttt{c3}][i_4/\texttt{q}][i_1/\texttt{l1}], \epsilon, \emptyset)}$$

After the dynamic lifting, we add all these values into the circuit.

```
addClbit(c1)
addClbit(c2)
addClbit(c3)
addQubit(q)
addLiftedValue(l1)
addLiftedValue(l2)
```

This results in the context $(\sigma, \kappa, \emptyset)$, where:

- $\sigma = \sigma[i_1/\texttt{c1}][i_2/\texttt{c2}][i_3/\texttt{c3}][i_4/\texttt{q}][i_1/\texttt{l1}][i_2/\texttt{l2}]$
- $\kappa = (\emptyset, \{i_4\}, \{i_1, i_2, i_3\}, \{i_1, i_2\})$

Next, we have the application of the `ifdl`.

```
ifdl(l1 & l2):
    X(q)
```

The derivation proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \texttt{l1}, (\sigma, \kappa, \emptyset)\rangle \to i_1}\,var}{\langle \texttt{l1}, (\sigma, \kappa, \emptyset)\rangle \to (i_1, \epsilon)}\,lv \qquad \dfrac{\dfrac{}{\langle \texttt{l2}, (\sigma, \kappa, \emptyset)\rangle \to i_2}\,var}{\langle \texttt{l2}, (\sigma, \kappa, \emptyset)\rangle \to (i_2, \epsilon)}\,lv}{\langle \texttt{l1\&l2}, (\sigma, \kappa, \emptyset)\rangle \to (i_5, \kappa^1)}\,lv_and \qquad \dfrac{\dfrac{}{\langle \texttt{q}, (\sigma, \kappa + \kappa^1, \{i_5\})\rangle \to i_4}\,var}{\langle \texttt{X(q)}, (\sigma, \kappa + \kappa^1, \{i_5\})\rangle \to (\sigma, \kappa^2, \{i_5\})}\,U}{\langle \texttt{ifdl (l1\&l2) :\ \ X(q)}, (\sigma, \kappa, \emptyset)\rangle \to (\sigma, \kappa^2, \emptyset)}\,ifdl$$

Let us take a closer look at this derivation.

First of all, we need to evaluate the lifted value expression `l1 & l2`. Once we evaluate `l1` and `l2` individually, the evaluation of the expression is given by the function $rev_and((i_1, \epsilon), (i_2, \epsilon))$. This function implements the quantum logical operation we described in the previous chapter when talking about the implementation. We will use $\mathsf{And} \in Instr^*$ to refer to the sequence of instructions needed to implement the quantum conjunction operator[1]. The And instruction needs three

auxiliary quantum bits anc_1, anc_2, anc_3 and one classical bit, i_5, which is lifted. So the *rev_and* function returns the couple (i_5, κ^1), where

$$\kappa^1 = (\mathsf{And}, \{anc_1, anc_2, anc_3\}, \{i_5\}, \{i_5\}).$$

Next, we have the application of a X in the body of the `ifdl`. The context we use to evaluate `X(q)` has to incorporate the result of the lifted value expression in the guard. To do so, we add the lifted value i_5 to the set of active lifted values, and we add the subcircuit κ^1 to the circuit already in the context. The rule for the application of the unitary operators adds the instruction

$$\mathsf{X}_\mathsf{q}^{\mathsf{i5}} = (\mathtt{X}, [q], \emptyset, [i_5]) \in \mathit{Instr}$$

to the circuit. So, the evaluation of the configuration $\langle \mathtt{X(q)}, (\sigma, \kappa + \kappa^1, \cup\{i_5\})\rangle$ results in the context $(\sigma, \kappa^2, \{i_5\})$ where

$$\kappa^2 = (\mathsf{And} :: \mathsf{X}, \{i_4, anc_1, anc_2, anc_3\}, \{i_1, i_2, i_3, i_5\}, \{i_1, i_2, i_5\}).$$

Let us move on to the next command which is a simple application of a Hadamard gate and a measurement:

```
H(q)

Measure(q,c3)
```

They can be derived with the *U* and *meas* rules respectively, adding to the circuit the instructions:

$$\mathsf{H}_\mathsf{q} = (\mathtt{H}, [q], \emptyset, \emptyset) \in \mathit{Instr}$$
$$\mathsf{Meas}_\mathsf{q} = (\mathtt{Measure}, [q], [c3], \emptyset) \in \mathit{Instr} \quad .$$

At the end of the whole derivation, the result will be the context $(\sigma, \kappa, \emptyset)$ where:

- $\sigma = \sigma[i_1/\mathtt{c1}][i_2/\mathtt{c2}][i_3/\mathtt{c3}][i_4/\mathtt{q}][i_1/\mathtt{l1}][i_2/\mathtt{l2}]$
- $\kappa = (\mathsf{And} :: \mathsf{X} :: \mathsf{H} :: \mathsf{Meas}, \{i_4, anc_1, anc_2, anc_3\}, \{i_1, i_2, i_3, i_5\}, \{i_1, i_2, i_5\})$

$$\frac{}{\langle n,(\sigma,\kappa,\mathcal{I})\rangle \to n} \qquad \frac{\langle a,(\sigma,\kappa,\mathcal{I})\rangle \to n}{\langle \texttt{-}a,(\sigma,\kappa,\mathcal{I})\rangle \to -n} \qquad \frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{+}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0+n_1}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{-}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0-n_1} \qquad \frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{*}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 n_1}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{/}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0/n_1} \qquad \frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{\%}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0\ mod\ n_1}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{//}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to \lfloor n_0/n_1 \rfloor} \qquad \frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{**}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_0^{n_1}}$$

$$\frac{}{\langle \texttt{True},(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}} \qquad \frac{}{\langle \texttt{False},(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{==}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 = n_1 \\ \texttt{False if } n_0 \neq n_1 \end{cases}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{!=}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 \neq n_1 \\ \texttt{False if } n_0 = n_1 \end{cases}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{<}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 < n_1 \\ \texttt{False if } n_0 \nless n_1 \end{cases}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{>}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 > n_1 \\ \texttt{False if } n_0 \ngtr n_1 \end{cases}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{<=}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 \leq n_1 \\ \texttt{False if } n_0 \nleq n_1 \end{cases}$$

$$\frac{\langle a_0,(\sigma,\kappa,\mathcal{I})\rangle \to n_0 \qquad \langle a_1,(\sigma,\kappa,\mathcal{I})\rangle \to n_1}{\langle a_0\texttt{>=}a_1,(\sigma,\kappa,\mathcal{I})\rangle \to t} \quad \text{where } t = \begin{cases} \texttt{True if } n_0 \geq n_1 \\ \texttt{False if } n_0 \ngeq n_1 \end{cases}$$

$$\frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True} \qquad \langle b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}{\langle b_0 \texttt{ and } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}} \qquad \frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True} \qquad \langle b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}{\langle b_0 \texttt{ and } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}$$

$$\frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}{\langle b_0 \texttt{ and } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}} \qquad \frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False} \qquad \langle b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}{\langle b_0 \texttt{ or } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}$$

$$\frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False} \qquad \langle b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}{\langle b_0 \texttt{ or } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}} \qquad \frac{\langle b_0,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}{\langle b_0 \texttt{ or } b_1,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}$$

$$\frac{\langle b,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}{\langle \texttt{ not } b,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}} \qquad \frac{\langle b,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{False}}{\langle \texttt{ not } b,(\sigma,\kappa,\mathcal{I})\rangle \to \texttt{True}}$$

Figure 4.2: Arithmetic and Boolean expressions operational semantic rules.

$$\frac{\langle c_0, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma'', \kappa'', \mathcal{I}) \qquad \langle c_1, (\sigma'', \kappa'', \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}{\langle c_0 \;\; c_1, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}$$

$$\frac{\langle b, (\sigma, \kappa, \mathcal{I})\rangle \to \texttt{True} \qquad \langle c_0, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}{\langle \texttt{if}\; b\texttt{:} \quad c_0 \;\; \texttt{else :} \quad c_1, (\sigma, \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}$$

$$\frac{\langle b, (\sigma, \kappa, \mathcal{I})\rangle \to \texttt{False} \qquad \langle c_1, (\sigma, \kappa, \mathcal{I})\rangle \to \sigma'}{\langle \texttt{if}\; b\texttt{:} \quad c_0 \;\; \texttt{else :} \quad c_1, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}$$

$$\frac{\langle b, (\sigma, \kappa, \mathcal{I})\rangle \to \texttt{False}}{\langle \texttt{while}\; b \;\texttt{:} \quad c, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma, \kappa, \mathcal{I})}$$

$$\frac{\begin{array}{c}\langle b, (\sigma, \kappa, \mathcal{I})\rangle \to \texttt{True} \\ \langle c, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma'', \kappa'', \mathcal{I}) \qquad \langle \texttt{while}\; b \;\texttt{:} \quad c, (\sigma'', \kappa'', \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})\end{array}}{\langle \texttt{while}\; b \;\texttt{:} \quad c, (\sigma, \kappa, \mathcal{I})\rangle \to (\sigma', \kappa', \mathcal{I})}$$

Figure 4.3: Classical commands operational semantic rules.

Conclusions

In conclusion, this thesis has embarked on a journey into the realm of quantum computing, focusing on the integration of dynamic lifting into the widely used Qiskit framework. The exploration of dynamic lifting, a concept borrowed from other quantum languages like Quipper, has not only enriched the functionality of Qiskit but has also laid the groundwork for a formalized understanding of the resulting quantum language. This work follows the steps of others like [8][12][15] in giving a formal understanding of dynamic lifting, adapting it in a new environment.

As we reflect on the motivations behind this research, it becomes evident that the formalization of quantum programming languages is a critical step forward in the advancement of quantum computing. As a prominent quantum programming framework, Qiskit has been the focal point of our investigation, and the successful implementation of dynamic lifting offers a versatile and adaptable dimension to its capabilities. The formalization of this extension provides a structured foundation, facilitating a more comprehensive understanding of quantum algorithms developed within the Qiskit environment.

As we look towards the future, the findings and formalizations presented in this thesis lay the groundwork for further exploration and refinement of quantum programming languages, in particular the possibility of construing a new independent language based on Qiskit, further improving Qiskit's ergonomics.

The field of quantum computing is in need of ongoing research and innovation, and the contributions made here mark a modest advancement in this dynamic journey.

Appendix A

Code

```
from qiskit.circuit.classicalregister import Clbit
from qiskit import ClassicalRegister, AncillaRegister
from qiskit import QuantumCircuit
from qiskit.circuit.library.boolean_logic import AND, OR
from DynamicLifting import gc_global



class LiftedValue(Clbit):

    __slots__ = {'_clbit'}

    def __init__(self, clbit : Clbit):

        self._clbit = clbit

        super().__init__(register=self._clbit.register, index=self._clbit.index)

    def __and__(self, other: 'LiftedValue') -> 'LiftedValue': # symbol &
        return self._logic_binaryop(other, AND(2))


    def __or__(self, other: 'LiftedValue') -> 'LiftedValue': # symbol |
        return self._logic_binaryop(other, OR(2))

    def __invert__(self) -> 'LiftedValue':
        return self._logic_notop()

    def __enter__(self):
```

```
30          gc_global.dl_flag = True
31          gc_global.lifted_values.append(self)
32
33      def __exit__(self, exc_type, exc_value, traceback):
34          gc_global.lifted_values.pop()
35          if len(gc_global.lifted_values) == 0:
36              gc_global.dl_flag = False
37
38      def _logic_notop(self):
39          input_reg = ClassicalRegister(bits=[self._clbit])
40          output_reg = ClassicalRegister(1)
41          qreg = AncillaRegister(1)
42
43          temp_circ = QuantumCircuit(qreg, input_reg, output_reg)
44
45          temp_circ.x(0).c_if(input_reg[0], 0)
46
47          temp_circ.measure(qreg[0], output_reg[0])
48
49          gc_global.global_qc.add_register(input_reg)
50          gc_global.global_qc.add_register(output_reg)
51          gc_global.global_qc.add_register(qreg)
52
53          gc_global.global_qc.compose(
54              temp_circ, [qreg[0]],
55              [input_reg[0], output_reg[0]], inplace=True)
56
57          return LiftedValue(output_reg[0])
58
59      def _logic_binaryop(self, other, bool_op):
60          input_reg = ClassicalRegister(bits=[self._clbit, other._clbit])
61          output_reg = ClassicalRegister(1)
62          qreg = AncillaRegister(3)
63
64          temp_circ = QuantumCircuit(qreg, input_reg, output_reg)
65
66          temp_circ.x(0).c_if(input_reg[0], 1)
67          temp_circ.x(1).c_if(input_reg[1], 1)
68
69          temp_circ.compose(
70              bool_op, [qreg[0], qreg[1], qreg[2]], inplace=True)
71
72          temp_circ.measure(qreg[2], output_reg[0])
73
74          gc_global.global_qc.add_register(input_reg)
75          gc_global.global_qc.add_register(output_reg)
76          gc_global.global_qc.add_register(qreg)
```

```python

        gc_global.global_qc.compose(
            temp_circ, [qreg[0], qreg[1], qreg[2]],
            [input_reg[0], input_reg[1], output_reg[0]], inplace=True)

        return LiftedValue(output_reg[0])
```

```python
from qiskit import QuantumCircuit

global_qc = QuantumCircuit()

lifted_values = []
dl_flag = False
```

```python
from functools import reduce
import operator
import DynamicLifting.lifted_value as lv
from DynamicLifting import gc_global



def get_condition_register() -> lv.LiftedValue:
    global lifted_values

    if len(gc_global.lifted_values) <= 0:
        return
    elif len(gc_global.lifted_values) == 1:
        return gc_global.lifted_values[0]
    else:
        return reduce(operator.and_, gc_global.lifted_values[1:], gc_global.lifted_values[0])
```

```python
from qiskit import *
from DynamicLifting.gc_global import global_qc
from DynamicLifting import gc_global
from DynamicLifting import global_circuit
from qiskit.circuit.instruction import Instruction
from qiskit.circuit.instructionset import InstructionSet
from qiskit.circuit.register import Register
from qiskit.circuit.bit import Bit
from qiskit.circuit.gate import Gate
from qiskit.circuit.quantumregister import Qubit
from qiskit.circuit.classicalregister import Clbit
```

```
12 from qiskit.circuit.operation import Operation
13 from qiskit.circuit.quantumcircuitdata import CircuitInstruction
14 from qiskit.circuit.parameter import Parameter
15 from qiskit.circuit.parameterexpression import ParameterValueType
16 from qiskit.circuit.quantumcircuit import QubitSpecifier, ClbitSpecifier
17 from typing import (
18     Union,
19     Optional,
20     Sequence,
21     Iterable,
22     Dict,
23     Type,
24     Mapping
25 )
26 import typing
27
28
29
30 def makeGlobal(qc : QuantumCircuit):
31     """Create a new global circuit from a given circuit"""
32     gc_global.global_qc = QuantumCircuit()
33
34     for qreg in qc.qregs:
35         gc_global.global_qc.add_register(qreg)
36     for creg in qc.cregs:
37         gc_global.global_qc.add_register(creg)
38     for instruction in qc:
39         gc_global.global_qc.append(instruction)
40
41
42 # For easy testing on jupiter notebook
43 def regenerateGlobal():
44     gc_global.global_qc = QuantumCircuit()
45     gc_global.lifted_values = []
46
47
48 def takeGlobal():
49     '''Returns the global circuit so that it can be used as a local circuit'''
50     return gc_global.global_qc.copy()
51
52
53
54 def check_empty_global(qbits: QubitSpecifier = None, cbits: ClbitSpecifier = None):
55     """add the proper number of bits and qubits if the global circuit is empty"""
56     if isinstance(qbits, int):
57         if gc_global.global_qc.num_qubits < (qbits + 1):
58             gc_global.global_qc.add_register(QuantumRegister((qbits + 1) -
```

```
                (gc_global.global_qc.num_qubits)))
59      if isinstance(cbits, int):
60          if gc_global.global_qc.num_clbits < (cbits + 1):
61              gc_global.global_qc.add_register(ClassicalRegister(
62                  (cbits + 1) - (gc_global.global_qc.num_clbits)))
63
64
65  def addRegisterGlobal(*regs: Union[Register, int, Sequence[Bit]]):
66      gc_global.global_qc.add_register(*regs)
67
68  def qregsGlobal():
69      return gc_global.global_qc.qregs
70
71
72  def cregsGlobal():
73      return gc_global.global_qc.cregs
74
75  def qubitsGlobal():
76      return gc_global.global_qc.qubits
77
78
79  def clbitsGlobal():
80      return gc_global.global_qc.clbits
81
82
83
84  def dynamic_lifting(gate: InstructionSet):
85      if gc_global.dl_flag:
86          condition = global_circuit.get_condition_register()
87          return gate.c_if(condition._clbit, 1)
88
89      else:
90         return gate
91
92  def H(qubit: QubitSpecifier):
93      return dynamic_lifting(gc_global.global_qc.h(qubit))
94
95
96  def X(qubit: QubitSpecifier, label: Optional[str] = None):
97      dynamic_lifting(gc_global.global_qc.x(qubit, label))
98
99
100 def draw_global(
101     output: Optional[str] = None,
102     scale: Optional[float] = None,
103     filename: Optional[str] = None,
104     style: Optional[Union[dict, str]] = None,
```

```
105        interactive: bool = False,
106        plot_barriers: bool = True,
107        reverse_bits: bool = False,
108        justify: Optional[str] = None,
109        vertical_compression: Optional[str] = "medium",
110        idle_wires: bool = True,
111        with_layout: bool = True,
112        fold: Optional[int] = None,
113        # The type of ax is matplotlib.axes.Axes, but this is not a fixed dependency, so cannot be
114        # safely forward-referenced.
115        ax: Optional[typing.Any] = None,
116        initial_state: bool = False,
117        cregbundle: bool = None,
118        wire_order: list = None,):
119
120        return gc_global.global_qc.draw(
121            output,
122            scale,
123            filename,
124            style,
125            interactive,
126            plot_barriers,
127            reverse_bits,
128            justify,
129            vertical_compression,
130            idle_wires,
131            with_layout,
132            fold,
133            ax,
134            initial_state,
135            cregbundle,
136            wire_order)
137
138
139
140    def dataGlobal():
141        '''Calls data proprety of global circuit'''
142        return gc_global.global_qc.data
143
144    def op_start_timeGlobal():
145        '''Calls op_start_times proprety of global circuit'''
146        return gc_global.global_qc.op_start_times
147
148
149
150    def calibrationsGlobal():
151        '''Calls calibrations proprety of global circuit'''
```

```
152     return gc_global.global_qc.calibrations
153
154
155 def has_calibration_forGlobal(instr_context: tuple):
156     '''Calls has_calibration_for of global circuit'''
157     return gc_global.global_qc.has_calibration_for(instr_context)
158
159 def metadataGlobal():
160     '''Calls metadata proprety of global circuit'''
161     return gc_global.global_qc.metadata
162
163
164 def stringGlobal():
165     '''Returns the global circuit as a string'''
166     return str(gc_global.global_qc.draw(output="text"))
167
168 def equalGlobal(other):
169     if not isinstance(other, QuantumCircuit):
170         return False
171
172     from qiskit.converters import circuit_to_dag
173
174     return circuit_to_dag(gc_global.global_qc) == circuit_to_dag(other)
175
176 def has_registerGlobal(register: Register):
177     '''Calls has_register on global circuit'''
178     return gc_global.global_qc.has_register(register)
179
180 def revers_opsGlobal():
181     '''Calls reverse_ops on global circuit'''
182     return gc_global.global_qc.reverse_ops()
183
184 def reverse_bitsGlobal():
185     '''Calls reverse_bits on global circuit'''
186     return gc_global.global_qc.reverse_bits()
187
188 def inverseGlobal():
189     '''Calls inverse on global circuit'''
190     return gc_global.global_qc.inverse()
191
192 def repeatGlobal(reps: int):
193     '''Returns the global circuit repeated reps times'''
194     return gc_global.global_qc.repeat(reps)
195
196 def my_repeatGlobal(reps: int):
197     '''The global circuit is sostituted by the same circuit but repeated reps times'''
198     new_qc = gc_global.global_qc.repeat(reps)
```

```
199     makeGlobal(new_qc)
200
201 def powerGlobal(power: float, matrix_power: bool = False):
202     '''Raise global circuit to the power of ‘‘power‘‘ and returns it'''
203     return gc_global.global_qc.power(power, matrix_power)
204
205
206 def my_powerGlobal(power: float, matrix_power: bool = False):
207     '''Raise global circuit to the power of ‘‘power‘‘ '''
208     new_qc = gc_global.global_qc.power(power, matrix_power)
209     makeGlobal(new_qc)
210
211 def controlGlobal(
212     num_ctrl_qubits: int = 1,
213     label: Optional[str] = None,
214     ctrl_state: Optional[Union[str, int]] = None):
215     '''Returns controlled version of global circuit'''
216     return gc_global.global_qc.control(num_ctrl_qubits, label, ctrl_state)
217
218
219 def composeGlobal(
220     other: Union["QuantumCircuit", Instruction],
221     qubits: Optional[Sequence[Union[Qubit, int]]] = None,
222     clbits: Optional[Sequence[Union[Clbit, int]]] = None,
223     front: bool = False,
224     inplace: bool = False,
225     wrap: bool = False):
226     '''Compose global circuit with ‘other‘ circuit or instruction'''
227     return gc_global.global_qc.compose(other, qubits, clbits, front, inplace, wrap)
228
229
230 def tensorGlobal(other: "QuantumCircuit", inplace: bool = False):
231     '''Calls tensor function for global circuit'''
232     return gc_global.global_qc.tensor(other, inplace)
233
234 def ancillasGlobal():
235     '''Returns a list of ancilla bits in global circuit'''
236     return gc_global.global_qc.ancillas
237
238
239
240 def lenGlobal():
241     '''Returns number of instructiond in global circuit'''
242     return gc_global.global_qc.__len__
243
244
245
```

```
246 def appendGlobal(
247     instruction: Union[Operation, CircuitInstruction],
248     qargs: Optional[Sequence[QubitSpecifier]] = None,
249     cargs: Optional[Sequence[ClbitSpecifier]] = None,):
250     '''Append one or more instructions to global circuit'''
251     return gc_global.global_qc.append(instruction, qargs, cargs)
252
253
254 def add_bitsGlobal(bits: Iterable[Bit]):
255     '''Add bits to global register'''
256     gc_global.global_qc.add_bits(bits)
257
258 def find_bitGlobal(bit : Bit):
259     return gc_global.global_qc.find_bit(bit)
260
261 def to_instructionGlobal(
262     parameter_map: Optional[Dict[Parameter, ParameterValueType]] = None,
263     label: Optional[str] = None):
264       '''Create an Instruction out of global circuit'''
265       return gc_global.global_qc.to_instruction(parameter_map, label)
266
267 def to_gateGlobal(
268     parameter_map: Optional[Dict[Parameter, ParameterValueType]] = None,
269     label: Optional[str] = None):
270     '''Create a Gate out of global circuit.'''
271     return gc_global.global_qc.to_gate(parameter_map, label)
272
273
274 def decomposeGLobal(
275     gates_to_decompose: Optional[
276     Union[Type[Gate], Sequence[Type[Gate]], Sequence[str], str]
277 ] = None,
278     reps: int = 1):
279     return gc_global.global_qc.decompose(gates_to_decompose, reps)
280
281 def qasmGlobal(
282     formatted: bool = False,
283     filename: Optional[str] = None,
284     encoding: Optional[str] = None
285     ):
286     ''' Return OpenQASM string of global circuit'''
287     return gc_global.global_qc.qasm(formatted, filename, encoding)
288
289
290 def sizeGlobal(
291     filter_function: Optional[callable] = lambda x: not getattr(
292         x.operation, "_directive", False)):
```

```
293         '''Returns total number of instruction in global circuit'''
294         return gc_global.global_qc.size(filter_function)
295
296
297 def depthGlobal(
298     filter_function: Optional[callable] = lambda x: not getattr(
299     x.operation, "_directive", False)):
300     '''Return global circuit depth (i.e, length of critical path)'''
301     return gc_global.global_qc.depth(filter_function)
302
303 def widthGlobal():
304     '''Return number qubit plus cibits of global circuit'''
305     return gc_global.global_qc.width()
306
307 def num_qubitsGlobal():
308     '''Return the number of qubits in global circuit'''
309     return gc_global.global_qc.num_qubits()
310
311 def num_ancillasGlobal():
312     '''Return the number of ancilla qubits in global circuit'''
313     return gc_global.global_qc.num_ancillas()
314
315 def num_clbitsGlobal():
316     '''Return number of classical bits in global circuit'''
317     return gc_global.global_qc.clbits()
318
319 def count_opsGlobal():
320     '''Count each operation kind in the global circuit'''
321     return gc_global.global_qc.count_ops()
322
323
324 def num_nonlocal_gatesGlobal():
325     return gc_global.global_qc.num_nonlocal_gates()
326
327
328 def get_instructionGlobal(name: str):
329     return gc_global.global_qc.get_instructions(name)
330
331
332 def num_connected_componentsGlobals(unitary_only: bool = False):
333     return gc_global.global_qc.num_connected_components(unitary_only)
334
335
336 def num_unitary_factorsGlobal():
337     return gc_global.global_qc.num_unitary_factors()
338
339
```

```
340 def num_tensor_factorsGlobal():
341     return gc_global.global_qc.num_tensor_factors
342
343
344 def copyGlobal(name: Optional[str] = None):
345     '''Copy the global circuit'''
346     return gc_global.global_qc.copy(name)
347
348
349 def copy_empty_likeGlobal(name: Optional[str] = None):
350     '''Return a copy of global circuit with the same structure but empty'''
351     return gc_global.global_qc.copy_empty_like(name)
352
353 def clearGlobal():
354     '''Clear all instructions in global circuit'''
355     gc_global.global_qc.clear()
356
357
358 def resetGlobal(qubit: QubitSpecifier):
359     return gc_global.global_qc.reset(qubit)
360
361 def setQubits(params, qubits = None):
362     return gc_global.global_qc.initialize(params, qubits)
363
364
365
366 def Measure(qubit: QubitSpecifier, cbit: ClbitSpecifier):
367     '''Measure quantum bit into classical bit on global circuit'''
368     gc_global.global_qc.measure(qubit, cbit)
369
370
371 def Measure_active(inplace: bool = True):
372     '''Adds measurement to all non-idle qubits in global circuit'''
373     return gc_global.global_qc.measure_active(inplace)
374
375
376 def Measure_all(inplace: bool = True, add_bits: bool = True):
377     '''Adds measurement to all qubits in global circuit'''
378     return gc_global.global_qc.measure_all(inplace, add_bits)
379
380
381 def Remove_final_measurements(inplace: bool = True):
382     '''Removes final measurements and barriers on all qubits if they are present'''
383     return gc_global.global_qc.remove_final_measurements(inplace)
384
385
386 def global_phaseGlobal():
```

```
387     '''Return the global phase of the global circuit in radians.'''
388     return gc_global.global_qc.global_phase
389 
390 def parametersGlobal():
391     '''The parameters defined in the global circuit'''
392     return gc_global.global_qc.parameters
393 
394 
395 def num_parameters():
396     '''The number of parameter objects in the circuit'''
397     return gc_global.global_qc.num_parameters
398 
399 
400 def assign_parametersGlobal(
401     parameters: Union[Mapping[Parameter, ParameterValueType], Sequence[ParameterValueType]],
402     inplace: bool = False):
403     '''Assign parameters to new parameters or values.'''
404     return gc_global.global_qc.assign_parameters(parameters, inplace)
405 
406 
407 def bind_parametersGlobal(values: Union[Mapping[Parameter, float], Sequence[float]]):
408     return gc_global.global_qc.bind_parameters(values)
409 
410 
411 def Barrier(*qargs: QubitSpecifier, label=None):
412 
413     if not qargs:
414         global_barrier = gc_global.global_qc.barrier(label=label)
415     else:
416         global_barrier = gc_global.global_qc.barrier(qargs, label)
417     return global_barrier
418 
419 def delayGlobal(
420     duration: ParameterValueType,
421     qarg: Optional[QubitSpecifier] = None,
422     unit: str = "dt"):
423     '''Calls delay on global circuit'''
424     return gc_global.global_qc.delay(duration, qarg, unit)
425 
426 
427 def CH(
428     control_qubit: QubitSpecifier,
429     target_qubit: QubitSpecifier,
430     label: Optional[str] = None,
431     ctrl_state: Optional[Union[str, int]] = None):
432     dynamic_lifting(gc_global.global_qc.ch(control_qubit, target_qubit, label, ctrl_state))
433 
```

```
434
435
436 def I(qubit: QubitSpecifier):
437     dynamic_lifting(gc_global.global_qc.i(qubit))
438
439
440
441 def ID(qubit: QubitSpecifier):
442     dynamic_lifting(gc_global.global_qc.id(qubit))
443
444
445
446 def Ms(theta: ParameterValueType, qubits: Sequence[QubitSpecifier]):
447     dynamic_lifting(gc_global.global_qc.ms(theta, qubits))
448
449
450
451 def P(theta: ParameterValueType, qubit: QubitSpecifier):
452     dynamic_lifting(gc_global.global_qc.p(theta, qubit))
453
454
455 def Cp(
456     theta: ParameterValueType,
457     control_qubit: QubitSpecifier,
458     target_qubit: QubitSpecifier,
459     label: Optional[str] = None,
460     ctrl_state: Optional[Union[str, int]] = None):
461     dynamic_lifting(gc_global.global_qc.cp(
462           theta, control_qubit, target_qubit, label, ctrl_state))
463
464
465
466 def Mcp(
467     lam: ParameterValueType,
468     control_qubits: Sequence[QubitSpecifier],
469     target_qubit: QubitSpecifier):
470     dynamic_lifting(gc_global.global_qc.mcp(lam, control_qubits, target_qubit))
471
472
473 def R(
474     theta: ParameterValueType, phi: ParameterValueType, qubit: QubitSpecifier):
475     dynamic_lifting(gc_global.global_qc.r(theta, phi, qubit))
476
477
478
479 def Rv(
480     vx: ParameterValueType,
```

```
481        vy: ParameterValueType,
482        vz: ParameterValueType,
483        qubit: QubitSpecifier):
484        dynamic_lifting(gc_global.global_qc.rv(vx, vy, vz, qubit))
485
486
487
488    def Rccx(
489        control_qubit1: QubitSpecifier,
490        control_qubit2: QubitSpecifier,
491        target_qubit: QubitSpecifier):
492        dynamic_lifting( gc_global.global_qc.rccx(control_qubit1, control_qubit2, target_qubit))
493
494    def Rcccx(
495        control_qubit1: QubitSpecifier,
496        control_qubit2: QubitSpecifier,
497        control_qubit3: QubitSpecifier,
498        target_qubit: QubitSpecifier):
499        dynamic_lifting( gc_global.global_qc.rcccx(control_qubit1, control_qubit2, control_qubit3,
               target_qubit))
500
501
502    def Rx(theta: ParameterValueType, qubit: QubitSpecifier, label: Optional[str] = None):
503        dynamic_lifting( gc_global.global_qc.rx(theta, qubit, label))
504
505
506    def Crx(
507        theta: ParameterValueType,
508        control_qubit: QubitSpecifier,
509        target_qubit: QubitSpecifier,
510        label: Optional[str] = None,
511        ctrl_state: Optional[Union[str, int]] = None):
512        dynamic_lifting( gc_global.global_qc.crx(theta, control_qubit, target_qubit, label,
               ctrl_state))
513
514
515    def Rxx(theta: ParameterValueType, qubit1: QubitSpecifier, qubit2: QubitSpecifier):
516        dynamic_lifting( gc_global.global_qc.rxx(theta, qubit1, qubit2))
517
518
519    def Ry(theta: ParameterValueType, qubit: QubitSpecifier, label: Optional[str] = None):
520        dynamic_lifting( gc_global.global_qc.ry(theta, qubit, label))
521
522
523    def Cry(theta: ParameterValueType,
524            control_qubit: QubitSpecifier,
525            target_qubit: QubitSpecifier,
```

```
526         label: Optional[str] = None,
527         ctrl_state: Optional[Union[str, int]] = None,):
528         dynamic_lifting(gc_global.global_qc(theta, control_qubit, target_qubit, label, ctrl_state))
529 
530 
531 def Ryy(theta: ParameterValueType, qubit1: QubitSpecifier, qubit2: QubitSpecifier):
532     dynamic_lifting(gc_global.global_qc(theta, qubit1, qubit2))
533 
534 
535 def Rz(phi: ParameterValueType, qubit: QubitSpecifier):
536     dynamic_lifting(gc_global.global_qc.rz(phi, qubit))
537 
538 
539 def Crz(theta: ParameterValueType,
540         control_qubit: QubitSpecifier,
541         target_qubit: QubitSpecifier,
542         label: Optional[str] = None,
543         ctrl_state: Optional[Union[str, int]] = None):
544         dynamic_lifting( gc_global.global_qc.crz(theta, control_qubit, target_qubit, label,
             ctrl_state))
545 
546 
547 def Rzx(theta: ParameterValueType, qubit1: QubitSpecifier, qubit2: QubitSpecifier):
548     dynamic_lifting( gc_global.global_qc.rzx(theta, qubit1, qubit2))
549 
550 
551 def Rzz(theta: ParameterValueType, qubit1: QubitSpecifier, qubit2: QubitSpecifier):
552     dynamic_lifting( gc_global.global_qc.rzz(theta, qubit1, qubit2))
553 
554 
555 def Ecr(qubit1: QubitSpecifier, qubit2: QubitSpecifier):
556     dynamic_lifting( gc_global.global_qc.ecr(qubit1, qubit2))
557 
558 def S(qubit: QubitSpecifier):
559     dynamic_lifting( gc_global.global_qc.s(qubit))
560 
561 def Sdg(qubit: QubitSpecifier):
562     dynamic_lifting( gc_global.global_qc.sdg(qubit))
563 
564 def Cs(
565     control_qubit: QubitSpecifier,
566     target_qubit: QubitSpecifier,
567     label: Optional[str] = None,
568     ctrl_state: Optional[Union[str, int]] = None):
569     dynamic_lifting( gc_global.global_qc.cs(control_qubit, target_qubit, label, ctrl_state))
570 
571 def Csdg(
```

```
572     control_qubit: QubitSpecifier,
573     target_qubit: QubitSpecifier,
574     label: Optional[str] = None,
575     ctrl_state: Optional[Union[str, int]] = None):
576     dynamic_lifting( gc_global.global_qc.csdg(control_qubit, target_qubit, label, ctrl_state))
577 
578 def Swap(qubit1: QubitSpecifier, qubit2: QubitSpecifier):
579     dynamic_lifting( gc_global.global_qc.swap(qubit1, qubit2))
580 
581 
582 def Iswap(qubit1: QubitSpecifier, qubit2: QubitSpecifier):
583     dynamic_lifting( gc_global.global_qc.iswap(qubit1, qubit2))
584 
585 def Cswap(
586     control_qubit: QubitSpecifier,
587     target_qubit1: QubitSpecifier,
588     target_qubit2: QubitSpecifier,
589     label: Optional[str] = None,
590     ctrl_state: Optional[Union[str, int]] = None):
591     dynamic_lifting( gc_global.global_qc.cswap(control_qubit, target_qubit1, target_qubit2,
            label, ctrl_state))
592 
593 def Fredkin(
594     control_qubit: QubitSpecifier,
595     target_qubit1: QubitSpecifier,
596     target_qubit2: QubitSpecifier):
597     dynamic_lifting(gc_global.global_qc.fredkin(control_qubit, target_qubit1, target_qubit2))
598 
599 def Sx( qubit: QubitSpecifier):
600     dynamic_lifting( gc_global.global_qc.sx(qubit))
601 
602 def Sxdg( qubit: QubitSpecifier):
603     dynamic_lifting( gc_global.global_qc.sxdg(qubit))
604 
605 def Csx(
606     control_qubit: QubitSpecifier,
607     target_qubit: QubitSpecifier,
608     label: Optional[str] = None,
609     ctrl_state: Optional[Union[str, int]] = None):
610     dynamic_lifting(gc_global.global_qc.csx(control_qubit, target_qubit, label, ctrl_state))
611 
612 def T(qubit : QubitSpecifier):
613     dynamic_lifting( gc_global.global_qc.t(qubit))
614 
615 def Tdg(qubit: QubitSpecifier):
616     dynamic_lifting( gc_global.global_qc.tdg(qubit))
617 
```

```
618 def U(
619     theta: ParameterValueType,
620     phi: ParameterValueType,
621     lam: ParameterValueType,
622     qubit: QubitSpecifier):
623     dynamic_lifting( gc_global.global_qc.u(theta, phi, lam, qubit))
624
625 def Cu(
626     theta: ParameterValueType,
627     phi: ParameterValueType,
628     lam: ParameterValueType,
629     gamma: ParameterValueType,
630     control_qubit: QubitSpecifier,
631     target_qubit: QubitSpecifier,
632     label: Optional[str] = None,
633     ctrl_state: Optional[Union[str, int]] = None):
634     dynamic_lifting( gc_global.global_qc.cu(theta, phi, lam, gamma, control_qubit, target_qubit,
            label, ctrl_state))
635
636
637 def Cx(
638     control_qubit: QubitSpecifier,
639     target_qubit: QubitSpecifier,
640     label: Optional[str] = None,
641     ctrl_state: Optional[Union[str, int]] = None):
642     dynamic_lifting( gc_global.global_qc.cx(control_qubit, target_qubit, label, ctrl_state))
643
644 def Cnot(
645     control_qubit: QubitSpecifier,
646     target_qubit: QubitSpecifier,
647     label: Optional[str] = None,
648     ctrl_state: Optional[Union[str, int]] = None):
649     dynamic_lifting( gc_global.global_qc.cnot(control_qubit, target_qubit, label, ctrl_state))
650
651 def Dcx( qubit1: QubitSpecifier, qubit2: QubitSpecifier):
652     dynamic_lifting( gc_global.global_qc.dcx(qubit1, qubit2))
653
654
655 def Ccx(control_qubit1: QubitSpecifier,
656         control_qubit2: QubitSpecifier,
657         target_qubit: QubitSpecifier,
658         ctrl_state: Optional[Union[str, int]] = None):
659         dynamic_lifting( gc_global.global_qc.ccx(control_qubit1, control_qubit2, target_qubit,
                ctrl_state))
660
661 def Toffoli(
662     control_qubit1: QubitSpecifier,
```

```
663     control_qubit2: QubitSpecifier,
664     target_qubit: QubitSpecifier):
665     dynamic_lifting( gc_global.global_qc.toffoli(control_qubit1, control_qubit2, target_qubit))
666 
667 def Mcx(
668     control_qubits: Sequence[QubitSpecifier],
669     target_qubit: QubitSpecifier,
670     ancilla_qubits: Optional[Union[QubitSpecifier,Sequence[QubitSpecifier]]] = None,
671     mode: str = "noancilla"):
672     dynamic_lifting( gc_global.global_qc.mcx(control_qubits, target_qubit, ancilla_qubits, mode))
673 
674 def Mct(
675     control_qubits: Sequence[QubitSpecifier],
676     target_qubit: QubitSpecifier,
677     ancilla_qubits: Optional[Union[QubitSpecifier,
678                             Sequence[QubitSpecifier]]] = None,
679     mode: str = "noancilla"):
680     dynamic_lifting( gc_global.global_qc.mct(control_qubits, target_qubit, ancilla_qubits, mode))
681 
682 def Y(qubit: QubitSpecifier):
683     dynamic_lifting( gc_global.global_qc.y(qubit))
684 
685 def Cy(
686     control_qubit: QubitSpecifier,
687     target_qubit: QubitSpecifier,
688     label: Optional[str] = None,
689     ctrl_state: Optional[Union[str, int]] = None):
690     dynamic_lifting( gc_global.global_qc.cy(control_qubit, target_qubit, label, ctrl_state))
691 
692 def Z(qubit: QubitSpecifier):
693     dynamic_lifting( gc_global.global_qc.z(qubit))
694 
695 def Cz(
696     control_qubit: QubitSpecifier,
697     target_qubit: QubitSpecifier,
698     label: Optional[str] = None,
699     ctrl_state: Optional[Union[str, int]] = None):
700     dynamic_lifting( gc_global.global_qc.cz(control_qubit, target_qubit, label, ctrl_state))
701 
702 def Ccz(
703     control_qubit1: QubitSpecifier,
704     control_qubit2: QubitSpecifier,
705     target_qubit: QubitSpecifier,
706     label: Optional[str] = None,
707     ctrl_state: Optional[Union[str, int]] = None):
708     dynamic_lifting( gc_global.global_qc.ccz(control_qubit1, control_qubit2, target_qubit, label,
            ctrl_state))
```

```
709
710
711 def Pauli(pauli_string: str,
712          qubits: Sequence[QubitSpecifier]):
713          dynamic_lifting( gc_global.global_qc.pauli(pauli_string, qubits))
714
715
716 def Add_calibration(
717     gate: Union[Gate, str],
718     qubits: Sequence[int],
719     # Schedule has the type ‘qiskit.pulse.Schedule‘, but ‘qiskit.pulse‘ cannot be imported
720     # while this module is, and so Sphinx will not accept a forward reference to it. Sphinx
721     # needs the types available at runtime, whereas mypy will accept it, because it handles the
722     # type checking by static analysis.
723     schedule,
724     params: Optional[Sequence[ParameterValueType]] = None):
725     gc_global.global_qc.add_calibration(gate, qubits, schedule, params)
726
727 def Qubit_start_time(*qubits: Union[Qubit, int]):
728     return gc_global.global_qc.qubit_start_time(qubits)
729
730
731 def Qubit_duration( *qubits: Union[Qubit, int]):
732     return gc_global.global_qc.qubit_duration(qubits)
733
734
735 def Qubit_start_time(*qubits: Union[Qubit, int]):
736     return gc_global.global_qc.qubit_start_time(qubits)
737
738
739 def Qubit_stop_time(*qubits: Union[Qubit, int]):
740     return gc_global.global_qc.qubit_stop_time(qubits)
```

Bibliography

[1] Ibm quantum platform. https://quantum-computing.ibm.com/. [Online; accessed 19/11/2023].

[2] The python language reference. https://docs.python.org/3/reference/compound_stmts.html#the-with-statement. [Online; accessed 15/11/2023].

[3] The python language reference. https://docs.python.org/3/reference/lexical_analysis.html#identifiers. [Online; accessed 19/11/2023].

[4] Qiskit documentation. https://docs.quantum-computing.ibm.com/. [Online; accessed 19/11/2023].

[5] Qiskit gates library. https://docs.quantum-computing.ibm.com/api/qiskit/circuit_library. [Online; accessed 19/11/2023].

[6] Quantum circuits get a dynamic upgrade with the help of concurrent classical computation. https://research.ibm.com/blog/quantum-phase-estimations. [Online; accessed 15/11/2023].

[7] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 11–20, 1993.

[8] Andrea Colledan and Ugo Dal Lago. On dynamic lifting and effect typing in circuit description languages. In *28th International Conference on Types for*

Proofs and Programs (TYPES 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[9] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[10] David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[11] David Elieser Deutsch. Quantum computational networks. *Proceedings of the royal society of London. A. mathematical and physical sciences*, 425(1868):73–90, 1989.

[12] Peng Fu, Kohei Kishida, Neil J Ross, and Peter Selinger. Proto-quipper with dynamic lifting. *Proceedings of the ACM on Programming Languages*, 7(POPL):309–334, 2023.

[13] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

[14] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[15] Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In *FSTTCS 2021*, volume 213 of *LIPIcs*, pages 51:1–51:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[16] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[17] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[18] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[19] Robert Wille and Lukas Burgholzer. Verification of quantum circuits. *Handbook of Computer Architecture*, pages 1–28, 2022.

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