ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

School of Science
Department of Computer Science and Engineering
Degree in Computer Science

QUANTUM CIRCUIT SIZE ESTIMATION VIA HOARE LOGIC

Supervisor:
Prof. Ugo Dal Lago

Submitted by:
Leonardo Venturi

Academic Year 2023/2024

Contents

Introduction

Quantum computing represents one of the most exciting and rapidly evolving frontiers in modern computing. Quantum computing leverages unique quantum phenomena, such as superposition, entanglement, and interference, to process information using quantum bits, or qubits, which differ fundamentally from classical bits in their ability to exist in multiple states simultaneously. This emerging area has profound implications for fields such as cryptography, optimization, materials science, machine learning, and more.

Within this field, programming languages play a critical role in the design, implementation, and testing of quantum algorithms. These programming languages are specifically designed to interface with quantum circuits which are the core computational models of quantum computers. Over time, a variety of quantum programming languages have been developed to address the unique requirements of programming quantum hardware. Among these, Qiskit stands out as one of the most prominent and widely adopted frameworks.

However, as quantum computing technologies continue to mature, the field of imperative quantum programming languages faces challenges. Despite their potential, these languages, like Qiskit, lack a comprehensive formal framework that integrates semantics and logical reasoning. The absence of such formalization poses significant obstacles for verifying the correctness, reliability, and scalability of quantum algorithms and circuits.

Formal methods can provide systematic ways to reason about program correctness, allowing developers and researchers to ensure that quantum algorithms are free from errors or unintended behavior. One such formal approach is Hoare logic, a formal system widely used in the verification of classical software programs.

This thesis will propose a formal method for verifying the size of circuits using Hoare logic. In doing so we will also develop a formalized language derived from Qiskit, called Proto-Qiskit and on that we will build a type system and a logic for reasoning on programs written in this language.

- In Chapter 1 we first introduce the major concepts in quantum computing. We discuss what a qubit is and how it can be represented, we then give the notion of quantum gates and what operations we are able to do in quantum computing. We then continue by explaining the principal model for representing computation: the quantum circuit. Additionally, we introduce Qiskit. We discuss its workflow and we provide examples on how to build circuits.

- In the second chapter we introduce the theory of programming languages, we explore what is the syntax and the semantics of them, we then go deeper into the explanation of two particular types of semantics: operational semantics and Hoare logic. In the part about Operational semantics, we will explain its major concepts like the state, the configuration and the transition, ending this part with a few examples of rules that defines the semantics. We will then end the chapter by explaining what a Hoare logic is and how it can be used to prove the correctness of programs, we will give rules for a simple imperative language.

- In the third chapter, we introduce Proto-Qiskit, a formalized representation of Qiskit. We will first outline its syntax and explain its key components, focusing on its primary constructs: expressions and commands. Subsequently, we will explore its semantics using a big-step semantics approach.

- The fourth chapter will examine the logic underpinning Proto-Qiskit. We will begin by introducing the concept of a type system, defining core ideas such as judgements and rules, and presenting the type system specific to Proto-Qiskit. Next, we will shift our focus to the logical framework, establishing a logic to reason about preconditions and postconditions of a program. The chapter will end with the definition of a Hoare logic and its derivation rules tailred to the verification of the size of circuits.

Chapter 1

Quantum Computing

Quantum computing is an interdisciplinary field which combines aspects of information theory, computer science and quantum physics. It leverages the principles of quantum mechanics to solve certain problems more efficiently than classical methods. This has the effect of making previously presumed infeasible problems, tractable. Notable quantum algorithms, such as Grover's algorithm for unstructured database search and Shor's factorization algorithm, exemplify the potential of quantum computing.

In this chapter we are going to discuss the main components of quantum computing: qubits, quantum gates and quantum circuits along with their classical counterparts. In the final section we will introduce Qiskit: a well known framework, tailored to the development of quantum circuits.

1.1 Qubits

The *qubit*, which is a shorthand for *quantum bit*, is the fundamental unit of information in quantum computing and plays a role analogous to that of the bit in classical computing. However, unlike a bit, which can only exist in one of two states, either 0 or 1, a qubit can exist not only in the states $|0\rangle$ and $|1\rangle$, but also in a *superposition* of both states simultaneously. Mathematically, the state of a qubit can be represented as a linear combination of two basis states, orthonormal vectors chosen within a two dimensional space called the Hilbert space. The two standard basis vectors, often called the computational basis, are:

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

A generic qubit $|\psi\rangle$ can therefore be expressed as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle .$$

Here, α and β are complex numbers known as *amplitudes*.

Example 1.1.1
An example of a physical system that can realize a qubit is:

- *An electron, where the two levels are given by the spin states* $|\uparrow\rangle$ *and* $|\downarrow\rangle$
- *A photon. Here the two levels are given by independent polarization states, for example horizontal and vertical:* $|H\rangle$ *and* $|V\rangle$*.*
- *An atom where the ground state and the first excited level are close to one another and relatively far from the others.*

A defining characteristic of a qubit is that its measure is both probabilistic and destructive. When measured, the qubit will collapse into one of its basis state, either $|0\rangle$ or $|1\rangle$, and will remain in that state afterward. The result $|0\rangle$ occurs with probability $|\alpha|^2$ and the outcome $|1\rangle$ occurs with probability $|\beta|^2$. Since we are dealing with probabilities, and $|\alpha|^2$, $|\beta|^2$ account for all possible outcomes, it must be that: $|\alpha|^2 + |\beta|^2 = 1$. This is called the normalization condition.

Taking into account the normalization condition, a single qubit state $|\psi\rangle$ can be expressed as:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle \quad \text{where} \quad \theta \in [0,\pi[\,, \varphi \in [0, 2\pi[$$

The parameters θ and φ define a point on the surface of a three-dimensional unit sphere. Component θ determines the angle relative to the ẑ-axis. The parameter φ on the other hand controls the angle of rotation, relative to the x̂-axis, in a counterclockwise manner. This second rotation is also called the *phase* of a qubit, where the factor of which it rotates is $e^{i\varphi}$.

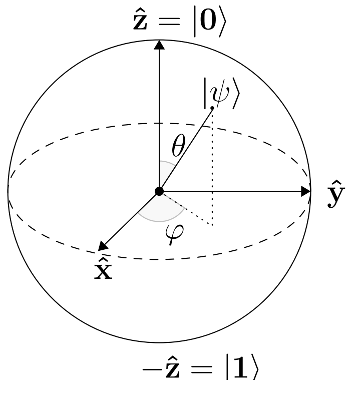
The aforementioned sphere, called the Bloch sphere, provides a geometric visualization for a qubit state. This representation however, does not scale, and we do not have an intuitive way to visualize multi-qubit states without losing information.

1.1.1 The Bloch sphere

The Bloch sphere, Figure 1.1, has specific points of interest. The ẑ-axis intercept the sphere in two points: the north pole ($\theta = 0$), representing $|0\rangle$ and the south pole ($\theta = \pi$), corresponding to $|1\rangle$. Similarly, the other two axes (x̂ and ŷ) define two points each. Therefore each one of them define an orthonormal basis for the qubit's Hilbert space.

1.1.2 Multiple qubits

So far we have covered only single qubit states. Extending the concept to two qubits, we observe that in the classical version, a two bit system can be in four distinct states:



$\hat{\mathbf{z}} = |\mathbf{0}\rangle$
$|\psi\rangle$
θ
$\hat{\mathbf{y}}$
φ
$\hat{\mathbf{x}}$
$-\hat{\mathbf{z}} = |\mathbf{1}\rangle$

Figure 1.1: Bloch Sphere.

00, 01, 10, 11. Similarly to the one bit case, the two qubit system can be in a linear combination of four basis states:

$$|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

where the probability of finding the system in any given state is given by $|\alpha_{ij}|^2$. The probabilities must satisfy the normalization condition, thus:

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$$

Similarly, for n qubit systems, the state can be written as:

$$|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x\,|x\rangle$$

Where $\{0,1\}^n$ denotes all the possible strings of zeros and ones of length n and $|\alpha_x|^2$ represents the probability of measuring the state $|x\rangle$.

1.1.3 Entanglement

When dealing with multi-qubit systems, a peculiar property called entanglement arises. This feature provides a way to have an interconnection between multiple qubits, no matter how far apart they are. In fact, the qubits involved exhibit a perfect correlation, meaning that the measurement outcome of one qubit will instantaneously determine the measurement of the other. Entanglement occurs when the state of one qubit cannot be described independently from the state of another. Formally, a multi-qubit state is entangled if it cannot be expressed as a product of single qubit states.

Example 1.1.2
The state

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle + |11\rangle\right)$$

is not entangled, since it can be written as:

$$|\psi_0\rangle = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes |1\rangle$$

Conversely, the state:

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right)$$

is entangled, as it cannot be factored as a product of single qubit's states.

A well known example of entangled states are the set of so-called Bell states. There exist four of those:

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right) \qquad |\Psi^-\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle - |10\rangle\right)$$

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) \qquad |\Phi^-\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right)$$

These pairs are an example of *maximally* entangled states, where maximally means that measuring just one qubit, determines the outcome of all the others. Bell pairs are also the simplest system to show this property, hence they provide a method to experiment with more ease on entanglement.

1.2 Quantum Gates

As logic gates in classical computers are the core components when building operations, quantum gates enable manipulation of qubit states in quantum computers. However, unlike their classical counterparts, all quantum gates are reversible, meaning that every operation performed on qubits (except for measurement) can be reversed. This is expressed by the mathematical representation of gates as unitary matrices. A unitary matrix is a complex matrix U, that satisfy the property $UU^\dagger = \mathbb{I}$, where $U^\dagger$ is the conjugate transpose of the matrix.

When a quantum gate is applied to a state, it alters the state according to the corresponding unitary matrix U. This transformation can be mathematically expressed as:

$$|\psi'\rangle = U\,|\psi\rangle$$

where $U\,|\psi\rangle$ stands for a matrix multiplication and $|\psi'\rangle$ is the resulting state.

Some of the most notable single qubit gates include:

Pauli Matrices:

Since these gates operate on single qubits, they are represented as 2×2 matrices.

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

The operation performed by these gates can be interpreted as:

- $\boldsymbol{\sigma_x}$: A *NOT* operation applied on the computational basis.
- $\boldsymbol{\sigma_y}$: A *NOT* operation and a phase flip.
- $\boldsymbol{\sigma_z}$: A phase flip.

Example 1.2.1 (Application of a gate)
Consider the initial state:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,.$$

Now, we apply the σ_x gate:

$$|\psi'\rangle = \boldsymbol{\sigma_x}\,|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

Therefore, the final state will be:

$$|\psi'\rangle = \beta\,|0\rangle + \alpha\,|1\rangle$$

It is evident that this operation is reversible, as applying it once more will return the state to its starting form.

Hadamard gate:

Another important single operation gate is the Hadamard gate. This gate is quite peculiar, as it allows for a change of base. The representation of the gate as a matrix is:

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

The new base after the application of the gate will be

$$\mathbf{H}\,|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \quad \mathbf{H}\,|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

1.2.1 Multi-qubit gates

As we introduced single qubit operations, we can also have gates that operate on multiple qubits, akin to how classical gates like *AND* and *OR* operate on bits. The most significant multi-qubit gates are the controlled gates, which allow one (or more) qubit, known as the *control*, to influence the state of another qubit, called *target*.

Controlled-NOT gate

The most notable gate of this class is the Controlled-NOT, or CNOT gate. It works by flipping the target qubit only if the control qubit is in the state $|1\rangle$. The mathematical representation is given by the following 4×4 matrix:

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

Example 1.2.2 (CNOT operation)
Consider a two qubit state:

$$|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

We apply the CNOT gate:

$$\mathbf{CNOT}|\psi\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix} = \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{11} \\ \alpha_{10} \end{bmatrix}.$$

The final state will be:

$$\mathbf{CNOT}\,|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{11}\,|10\rangle + \alpha_{10}\,|11\rangle$$

As we can see, the amplitudes of the last two states have been swapped.

SWAP gate

The SWAP gate, as the name suggests, swaps the state of the two qubits in input. The corresponding 4×4 matrix is:

$$\mathbf{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It can be shown that the SWAP gate is decomposable into a sequence of three CNOTs.

1.3 Quantum Circuits

The quantum circuit is the natural analogue of the classical circuit. It models the operations applied to qubits, with quantum gates being its building blocks.

A quantum circuit is canonically represented as a series of horizontal wires, where time flows from left to right. Each wire correspond to a qubit (or a bit if a measurement is carried out). The operations on the qubits are annotated in three distinct ways:

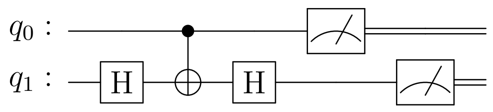
- As little boxes named with capital letters, if they involve a single qubit operation.
- As a vertical line connecting qubits, if they are a controlled operation. The control bit is denoted by a dot while the target bit can be denoted by various symbols depending on the operation.
- As a little square with a meter, if they are a measurement operation. This is usually followed by a double line representing the bit on which the measurement is stored.

Since quantum gates are reversible, there are several limitations on how they can be arranged in a circuit, the operations here described are not allowed:

- **Fan-in**: the convergence of multiple wires into one, because the process isn't reversible.
- **Fan-out**: wire duplication, due to the no-cloning theorem, which states that the exact copy of an unknown quantum state is unobtainable.
- **Loop**: the circuit must be acyclical.

The following example shows a representation of a quantum circuit as described above.

Example 1.3.1

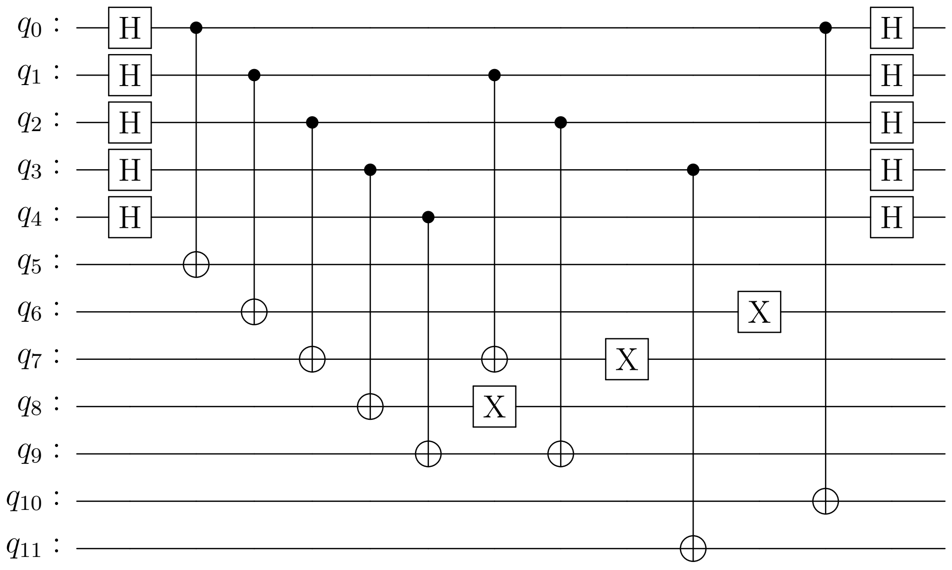


The single qubit operations are Hadamard gates, and the two qubit operation is a CNOT gate.

Two important measures are often used to evaluate the efficiency and complexity of a quantum circuit, these are its width and depth.
The width is regarded as the number of qubits of which the circuit is composed. The

depth instead is a measure of temporal complexity. It is defined as the count of time steps needed to execute all the gates in the circuit. The gates that can be executed in parallel do not contribute to the depth count.

Example 1.3.2
As an example let us examine the following circuit:



The width of this circuit is 12*, as the number of qubits. The depth, however, is not* 13*, as one would expect from the diagram, but only* 4*. This is due to the fact that most of the CNOTs can be applied simultaneously; for instance the first five CNOT gates can be executed in parallel , effectively counting as one.*

1.4 Qiskit

Qiskit is an open-source software development kit (SDK) created by IBM and written in Python. It allows users to write, simulate and execute quantum circuits. It has also the option for running circuits on IBM's quantum platform, by creating an account and selecting the appropriate provider. Since its release in 2017, Qiskit has gained a widespread popularity, becoming one of the most prominent projects in quantum computing.

The Qiskit's workflow can be broken down into four main parts:

- **Circuit Design**: create and manipulate quantum circuits.
- **Circuit Compilation**: prepare the circuit for execution on quantum hardware.

- **Execution**
- **Results Analysis**

The library is imported in the usual way:

```python
from qiskit import *
```

In the following examples we will omit the library import.

1.4.1 Circuit Design

In Qiskit, circuit design is delegated to the QuantumCircuit object.
In order to create a simple circuit we can do the following:

```python
qc = QuantumCircuit(2)
```

This line create a quantum circuit called qc that consist of two qubits. It should be noted that every qubit in Qiskit starts in the state $|0\rangle$.

A circuit can also be created with a second parameter which specifies the number of classical bits in the circuit. For instance:

```python
qc = QuantumCircuit(3,3)
```

creates a circuit of three qubits and three bits.

Gates can be added in two ways: the simplest one is to use the standard library of gates provided by Qiskit (`qiskit.circuit.library`), otherwise one can choose to use the append method to have more freedom in terms of gates choice.

```python
from qiskit.circuit.library import XGate
qc = QuantumCircuit(2)
qc.x(0)
qc.append(XGate(), [1])
```

Here we applied an X gate (σ_x) to the 0-th and 1st qubits. For controlled gates, we have to specify which qubit acts as the control and which one as the target.

```python
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1) #Qubit 0 is the control and qubit 1 is the target.
```

To obtain the state of a qubit after applying gates, we need to measure the qubit. Measurement collapses the qubit state into one of the computational basis states, and the result is stored in a classical bit.

An example involving a measurement operation is:

```python
qc = QuantumCircuit(5, 5)
qc.x(0)
qc.x(2)
qc.x(3)
qc.measure(range(5), range(5))
# Measures all qubits into the corresponding clbit.
```

As now we have designed our circuit, we might want a way to represent it. Qiskit offers plenty of ways to do so. The standard output can be obtained through the use of draw:

```python
qc = QuantumCircuit(3, 3)
qc.h(0)
qc.cx(0, 1)
qc.cx(1, 2)
qc.measure_all() #Another method for measuring all qubits
qc.draw()
```

The resulting output is in Figure 1.2.

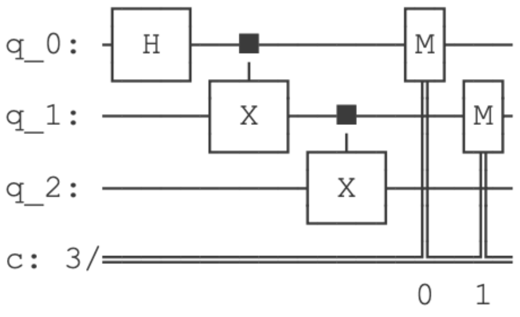


Figure 1.2: Qiskit standard output

Other possible output formats are: an image, created with the *matplotlib* library and latex code, which is produced using the *qcircuit* package.

1.4.2 Circuit Compilation

This part is comprised of various steps that aim at reducing the dimensions of the circuit. This is done because quantum hardware is noisy, due to multiple factors such as the instability of a qubit state; therefore, minimizing the number of operations and ensuring qubits are closer together when gate operations are applied, improves performance. In Qiskit this tasks are handled by the so-called transpiler. Its functionality can be understood in terms of three main stages:

- **Initialization**. In this first stage, the transpiler unrolls any custom instruction and produce a circuit made up of one and two qubit gates.
- **Quantum Hardware Mapping**. The circuit is adapted to any target quantum hardware. This process involves two steps: first, logical qubits are mapped to physical ones; second, any operation must be rewritten in terms of the ones supported by the hardware. The mapping between logical and physical qubits must satisfy the constraints dictated by the hardware, such as limited connections between qubits, while also minimizing the use of additional gates introduced by this procedure.
- **Optimization**. The transpiler optimizes the circuit using different routines that combines or eliminate gates. These method usually produces circuits which have lower widths and depths than the ones in input.

The stage we are interested in is the optimization stage. A very basic example of what Qiskit's optimization can do is the following:

```python
qc = QuantumCircuit(3)
qc.cx(0, 1)
qc.h([0,0])
qc.rx(np.pi/2, 1) # rotation about the X axis
qc.rx(-np.pi/2, 1)
qc.cx(0, 1)
qc.cx(0, 2)
qc.draw("mpl")
```

The graphical representation is in Figure 1.3

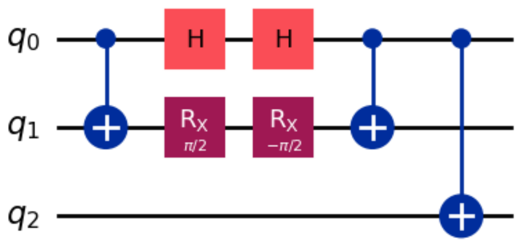


Figure 1.3: Initial circuit before compilation

The transpiled circuit is obtained with the following code:

```python
qc_compiled = transpile(qc)
qc_compiled.draw("mpl")
```

The output is displayed in Figure 1.4. We note that Qiskit has correctly identified as removable the sequence of gates which composed yield the identity. The depth decreased from five to one, and the number of qubits involved also decreased to two.

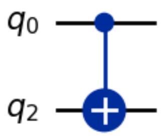


Figure 1.4: Transpiled circuit

1.4.3 Execution and Results Analysis

As for the execution of the circuit, we will use the simulator called AerSimulator, which has to be imported from *qiskit_aer*. The circuit is created as before, we will use a simple circuit with an X operation on the first qubit and an Hadamard operation on both qubits.

```
qc = QuantumCircuit(2,2)
qc.x(0)
qc.h([0,1])
qc.measure_all()
```

We can then use the simulator and obtain the results.

```
from qiskit_aer import AerSimulator
backend = AerSimulator()
qc_compiled = transpile(qc, backend)
job_sim = backend.run(qc_compiled, shots=1000)
result_sim = job_sim.result()
counts = result_sim.get_counts(qc_compiled)
```

The circuit is first transpiled for the specific backend. The *shots = 1000* refers to the number of times the circuit is being simulated, the default would be 1024. The results are finally retrieved using the *result* object with its *get_counts* method. To visualize these results we can import and use the *plot_histogram* function:

```
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

The resulting histogram of the simulation is in Figure 1.5

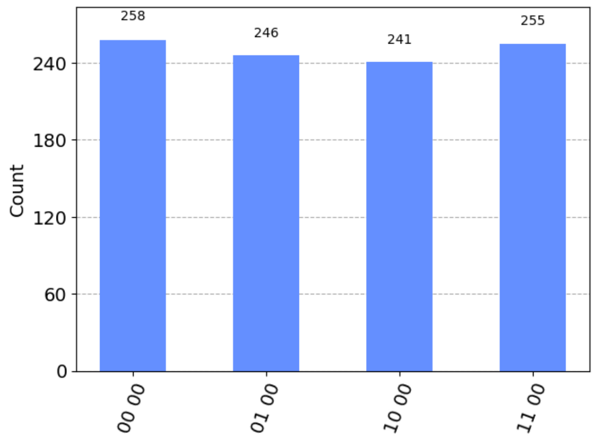


Figure 1.5: Histogram with results

Chapter 2

Programming Language Theory

Programming languages are at the core of modern computing, enabling developers to create and innovate in various ways. Many programming languages exist, each with its unique features and design choices, and many more are continually being developed. Given the vast diversity and the ever increasing number, mastering all of them is neither practical nor necessary. Instead, we can focus on understanding their underlying theory. The theory focuses on understanding the structures and principles that govern how languages are created, how they work and how they can be used to write software effectively. The fundamental aspect we are interested in is the behavior of programs, this can be useful both for defining what the program must do but also for verifying that the program satisfies certain properties. This is done through the use of a semantics. A key concept is a paradigm, which is a way to design and structure the implementation of a program.

In this chapter we will first discuss what the syntax and semantics of a programming language are, and how they are represented. Then we will focus on two types of semantics: Operational semantics and Floyd-Hoare logic (also called Axiomatic semantics).

2.1 Description of a language

A programming language can be described using three components:

- **The Syntax**: Delineates the valid combinations of symbols that can appear in a program. Namely the morphology of valid program sentences.
- **The Semantics**: Assigns meaning to the syntactic constructs, ensuring that they act as intended.
- **The Pragmatics**: Relates to how the language is used and how its features are employed to build programs. For example, the practice of using variables, defined

in a loop header, only within the loop's scope in order to prevent undesirable side effects.

2.1.1 Syntax

In order to understand the syntax, we first define what symbols and tokens (words) can appear in a programming language. These aspects are respectively resolved by specifying the alphabet and the vocabulary of the language. Additionally, syntax defines how valid phrases can be constructed using tokens.

All of these aspects are formally represented using a grammar, often defined using the Backus-Naur form (**BNF**). In this notation, the grammar rules are called production rules and they state how a non-terminal symbol can be replaced by a sequence of terminal and/or non-terminal symbols. A **BNF** production rule has the following format:

$$\text{symbol} \ ::= \ \text{alternative1} \mid \text{alternative2} \ \ldots$$

Where:

- "::=" means that the symbol on the left must be replaced with one of the alternatives on the right.
- "|" separates two alternatives.

Example 2.1.1
Consider a simple grammar for arithmetic sums in base two:

$$\begin{aligned} E &::= E + N \mid N \\ N &::= D \mid DN \\ D &::= 0 \mid 1 \end{aligned}$$

In this small example the alphabet is $A = \{0,1,+\}$. The non-terminals E, N and D represents respectively expressions, numbers and digits. Furthermore, DN represents the concatenation of a digit and a number, creating multiple digit numbers.

2.1.2 Semantics

Once the syntax is established, we can assign meaning to its constructs, ensuring that a program's behaviour aligns with its intended purpose.

There are three different approaches when giving semantics to a programming language:

- **Operational Semantics**: assigns meaning to programs by showing their execution steps, often modeled using transition systems.

- **Denotational Semantics**: maps mathematical objects, such as functions or sets, to language constructs, giving them meaning.
- **Axiomatic Semantics**: gives meaning to a program by describing its effect on assertions about the program's state.

Each of these approaches serves different purposes and highlights different aspects of a programming language.

2.2 Operational Semantics

Operational semantics is a way to give meaning to a programming language by specifying how its constructs execute. It specifies the behaviour of an abstract machine, which can be loosely defined as an abstraction of a physical computer. The execution is dealt using a transition system. In order to detail what a transition is, we first have to explain the concepts of state and configuration.

Program State

A state, often represented as σ, models the program's memory at a specific point, during execution. It is defined as a partial function, mapping variable names to their values. The mapping is deemed partial to account for variables that may be undefined in a certain part of a program.

Example 2.2.1

$$\sigma = \{(x, 5), (y, 8)\}$$

Here, the state indicated that the variable x has a value of 5 *and the variable y has a value of* 8.

Two basic operations can be performed on a state:

- **Modification**: The value of a variable is updated. This is denoted as:

$$\sigma[\text{new value/variable name}]$$

- **Lookup**: Retrieving the current value of a variable. This is represented as:

$$\sigma(x)$$

A configuration is a pair, consisting of a command (or an expression), and a state. Therefore, it encapsulate the current state and the remaining computation. It is often displayed as:

$$\langle c, \sigma \rangle$$

Transition

Finally, a transition expresses a single computational step, and shows how the execution of a command modifies the program's state. It can be seen as a mapping between configurations, and it's written as:

$$\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$$

This means that when executing the command c on the state σ, it transition the program to a new configuration, where c' is the updated command and σ' is the updated state. In many cases, commands are reduced until there is nothing left to compute. This is signified by the **skip** command. A final configuration:

$$\langle \textbf{skip}, \sigma \rangle$$

indicates that the computation is complete.

Example 2.2.2 (Variable assignment)
Consider the command $x \leftarrow x + 1$ and an initial state $\sigma = \{(x, 5)\}$. A corresponding computation can be:

$$\langle x \leftarrow x + 1, \sigma \rangle \rightarrow^* \langle \textbf{skip}, \sigma' \rangle$$

Where σ' is the new state, $\{(x, 6)\}$ and the command is reduced to **skip***, indicating that the computation has ended. The $*$ symbol denotes the Kleene star, so multiple transitions can be involved in obtaining the result.*

In many programming constructs, the transition can be conditional. These kinds of transitions are captured using an inference rule. A typical example is the rule for sequencing:

Example 2.2.3

$$\frac{\langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma' \rangle}{\langle c_1 \ ; \ c_2, \sigma \rangle \rightarrow \langle c_1' \ ; \ c_2, \sigma' \rangle}$$

This rule states that if the first command c_1 transitions to c_1' in state σ, then the sequence $c_1; c_2$ transitions to $c_1'; c_2$ in the same state.

Types of Operational Semantics

There exists two main approaches to operational semantics:

- **Small-step**: Execution is broken down into minimal steps. This allows for a precise control over the semantics and is usually denoted as $\rightarrow$.

- **Big-step**: Contrarily, the big-step approach, characterizes execution in terms of the final result. Each rule directly relates initial and final states, providing a simpler syntax and a more high-level view. We will represent it as $\Rightarrow$.

Example 2.2.4
As an example, we can evaluate using these two styles, the expression $2 + (7 * 3)$ *in a state, using:*

- *Small-step semantics:*

$$\langle 2 + (7 * 3), \sigma \rangle \rightarrow \langle 2 + 21, \sigma \rangle \rightarrow \langle x := 23, \sigma \rangle$$

- *Big-step semantics:*

$$\langle x := 2 + (7 * 3), \sigma \rangle \Rightarrow \langle x := 23, \sigma \rangle$$

Furthermore, small-step semantics can also be extended in many ways. A typical extension is the one with errors, where we introduce special states, such as **error**, to handle unexpected conditions. Other extensions can be also made to account for non-deterministic or parallel execution.

2.3 Floyd-Hoare Logic

Hoare logic, also known as axiomatic semantics, is a formal system with a set of logical rules for reasoning about the correctness of a program. It was proposed by C.A.R. Hoare in 1969, however, the foundational ideas were introduced earlier by Robert W. Floyd in a 1967 paper. It consists of two main parts: a language for stating assertions about programs, and rules for establishing the truth of assertions. Many logics can be used to encode assertions, for the sake of simplicity we will focus on first order logic.

2.3.1 Assertions

Assertions describe conditions about the state of a program at specific points during its execution. These assertions are expressed using arithmetic relations and are combined using first order logic.

Example 2.3.1
We can build various examples of assertions:

- $x = 1$*: Asserts that the variable* x *equals to 1.*
- $y < 3 \land x = 2$*: States that* x *equals to 2 and* y *is less than 3.*
- $\forall z.z > 0 \rightarrow x \geq z$*: Asserts that if* z *is a positive number,* x *is greater or equal than* z*.*

2.3.2 Hoare Triple

The fundamental concept of this formal system is a Hoare triple; it has the following general form:

$$\{P\}\, c\, \{Q\}$$

The triple state that: "if the formula P holds before the program execution, then, after the execution of c, the formula Q will hold". In this context, P is called precondition, Q is called postcondition and c is the program of interest. In this logic, correctness is typically discussed in terms of partial correctness. This guarantees that the program meets its specification *if* it terminates. Therefore, termination isn't ensured.

Partial correctness can also be given with respect to operational semantics, defining what a valid triple is:

Definition 2.3.1. A Hoare triple $\{P\}\, c\, \{Q\}$ is considered valid, if for any state σ, σ', such that σ satisfies P and $\langle c, \sigma\rangle \rightarrow^* \langle \mathbf{skip}, \sigma'\rangle$, then σ' satisfies Q. We will write this as $\vDash \{P\}\, c\, \{Q\}$

Example 2.3.2
A simple example can be stated as:

$$\{x = 1\}\, y \leftarrow x + 1\, \{y = 2\}$$

This statement asserts that if $x = 1$ *holds before the execution of the assignment, then* $y = 2$ *will hold afterwards.*

Example 2.3.3
Partial correctness ensures that the program computes correctly if it terminates, thus we can devise a correct triple as:

```
{$x = x_0 \wedge y = 3$}
 while $x \neq 0$ :
        $x \leftarrow x - 1$;
        $y \leftarrow y + 1$
 {$x = 0 \wedge y = x_0 + 3$}
```

Where x_0 *is a fixed value. We can distinguish between* $x_0 \geq 0$*, as the execution ends, and* $x_0 < 0$ *as the execution does not terminate. The triple is valid in both cases for partial correctness.*

2.3.3 Rules

The proof system defines how to derive triples, using inference rules. We will write $\vdash \{P\}\, c\, \{Q\}$ to indicate that the triple is derivable using the rules. These rules are given for the specific simple language considered in the previous sections, whose semantics is also simple. They include:

Skip

$$\frac{}{\{P\}\, \textbf{skip}\, \{P\}}$$

This rule asserts that the **skip** command does not change the state of the program, so the precondition and postcondition are identical.

Assignment

$$\frac{}{\{P[e/x]\}\, x \leftarrow e\, \{P\}}$$

The assignment rules states that, after an assignment, any predicate that was true for the right side of the assignment, now holds for the variable. The substitution $P[e/x]$ means that each free occurence of x has been replaced by e in P.

Example 2.3.4
An example of this kind of triple is: $\{x + 1 \leq N\}\, x \leftarrow x + 1\, \{x \leq N\}$

Composition

$$\frac{\{P\}\, c_1\, \{R\} \qquad \{R\}\, c_2\, \{Q\}}{\{P\}\, c_1\ ;\ c_2\, \{Q\}}$$

If executing the commands c_1 and c_2 sequentially results in the postcondition R being verified, then R will be the postcondition for the entire program.

Conditional

$$\frac{\{b \wedge P\}\, c_1\, \{Q\} \qquad \{\neg b \wedge P\}\, c_2\, \{Q\}}{\{P\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}}$$

This rule states that if the postcondition Q is common to both branches, it will also be a postcondition for the whole statement.

Loop

$$\frac{\{P \land b\}\, c\, \{P\}}{\{P\}\, \texttt{while}\, b : c\, \{\neg b \land P\}}$$

When dealing with loops, P is called loop invariant. It is an assertion that holds both before and after each iteration of the loop.

Consequence

$$\frac{P_1 \rightarrow P_2 \qquad \{P_2\}\, c\, \{Q_2\} \qquad Q_2 \rightarrow Q_1}{\{P_1\}\, c\, \{Q_1\}}$$

This rule is quite important, as it allows to strenghten the precondition P_2 and/or weaken the postcondition Q_2, while preserving its validity.

Example 2.3.5
Consider the triple:

$$\{y > 0\}\, x \leftarrow y\, \{x > 0\}$$

We can strenghten its precondition, making it more precise, using $y = 5 \Rightarrow y > 0$, *producing:*

$$\{y = 5\}\, x \leftarrow y\, \{x > 0\}$$

It can be proven that a system that comprises these rules is sound and complete with respect to validity. This is denoted as: $\vDash \{P\}\, c\, \{Q\} \Leftrightarrow \vdash \{P\}\, c\, \{Q\}$. Meaning that everything which is semantically valid is also syntactically provable and vice versa.

Chapter 3

Proto-Qiskit

In the preceding chapters, we examined two major concepts: quantum mechanics and the theory of programming languages. Our objective is now to establish a theoretical foundation for reasoning about quantum circuits, produced by circuit description languages, and their size.

To achieve this, we will make use of the previously introduced language library Qiskit, with the addition of the core features commonly found in programming languages, such as loops, conditional and assignments, taken from Python. Therefore, we will formally define this language, called Proto-Qiskit, which will serve as the basis for the next chapter, where we define a Hoare logic, tailored to reason about circuits' structural properties.

This chapter introduces the syntax and operational semantics of Proto-Qiskit.

3.1 Syntax

As `Proto-Qiskit` is designed as an abstraction of `Qiskit`, and therefore of the `Python` programming language, its syntax is similar to it. In this section we will outline it. We define two types of major constructs: *expressions* and *commands*.

3.1.1 Expressions

Expressions, denoted as e, represent the entities which are evaluated to produce values. These include:

- **Constants** (k): these are literal values, comprehensive of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$ and boolean values $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$.
- **Variables** ($\texttt{x}$): variables denote references to values. They are drawn from the set $\mathbb{V}$ and conform with Python's conventions. Specifically, variables consists of

non-empty strings containing alphanumeric characters and underscores. Notably, they cannot begin with a number.

- **Functions** (f): functions include standard operators, such as *Arithmetic*, *Boolean* and *Comparison* ones. Examples of these can be: the sum operator (+), the not operator (*not*) and the equality operator (==). Each operator is characterized by an arity $n \in \mathbb{N}$, denoting the number of inputs it accepts.

- **Quantum Circuits** (`QuantumCircuit(`e_1,e_2`)`): this expression introduces a quantum circuit, where e_1 is the number of qubits it is composed of and e_2 represents the number of bits.

In Proto-Qiskit, a quantum circuit is depicted as a triple (*Instr*, *Qb*, *Clb*), where *Qb* and *Clb* are natural numbers, and *Instr* is a sequence of instructions. An instruction is itself a triple from the set $\mathit{String} \times \mathbb{N} \times \mathbb{N}$ or the name of the instruction and the qubits and bits on which it operates.

The evaluation of an expression occurs with respect to a *state*, which is defined as a mapping:

$$\sigma : \mathbb{V} \longrightarrow \textbf{Value}, \text{ where } \textbf{Value} = \mathbb{N} + \mathbb{B} + \textbf{CIRC}$$

The set **CIRC** is the set which contains all circuits, therefore $\textbf{CIRC} = \mathit{Instr} \times \mathbb{N} \times \mathbb{N}$.

3.1.2 Commands

Commands are executable statements that, unlike expressions, can alter the program's state. The basic commands are the parallel of Python's, thus they include operations such as assignment, sequential execution, conditional statements and while loops. However, sequential execution is indicated by a semicolon, instead of relying on a new line.

In addition, specialized commands that allow us to manipulate quantum circuits are introduced. These are:

- **Unitary operation** `x.`$U(e_1 \dots e_n)$: this command applies a unitary transformation to the qubits referenced by $e_1, \dots, e_n$.

- **Measure** `x.measure(`e_1, e_2`)`: measures the qubit referenced by e_1 in circuit `x` and stores the result in bit e_2.

- **Composition** `x.compose(`e_1`)`: this operation concatenates the circuit in `x` with the one in e_1.

- **Adding of Qubits and Bits** `x.addQubits(`e_1`)` `x.addBits(`e_1`)`: these two commands allow us to add respectively e_1 qubits or bits, to the circuit in `x`.

Expressions:

e ::= k | `x` | $f^n(e_1 \ldots e_n)$ | `QuantumCircuit`(e_0, e_1)

Commands:

c ::= `x` $\leftarrow e$ | c_0 ; c_1 | `if` e `then` c_0 `else` c_1 | `while` $e : c$
  | `x.`$U(e_1, \ldots, e_n)$ | `x.compose`(e_1) | `x.measure`(e_1, e_2)
  | `x.addQubits`(e_1) | `x.addBits`(e_1)

Figure 3.1: Formal syntax of Proto-Qiskit

The complete syntax for the language is given in Figure 3.1

Example 3.1.1. Consider the following Qiskit code:

```
qc = QuantumCircuit(3, 3)
qc.h(0)
qc.cx(0, 1)
qc.cx(1, 2)
qc.measure_all()
```

Using Proto-Qiskit's syntax, it becomes:

```
qc ← QuantumCircuit(3, 3);
qc.H(0);
qc.CX(0, 1);
qc.CX(1, 2);
qc.measure(0,0);
qc.measure(1,1);
qc.measure(2,2)
```

Which is nearly identical to the original syntax, except for the use of the semicolon between commands and the arrow as assignment.

3.2 Operational Semantics

The operational semantics described here, is presented using the big-step style, as it provides a clearer and more concise representation. To formally express a transition in our big-step semantics, we use the notation: $\Rightarrow$.

3.2.1 Expressions

We start by describing the evaluation of expressions. Expressions do not modify the state, therefore a relation between a configuration and a resulting value can be defined as $\langle e, \sigma \rangle \Rightarrow v$, which means that the expression e evaluated in state σ, yields the value v.

As an example, we show the inference rule for the evaluation of a function:

$$\frac{\langle e_1, \sigma \rangle \rightarrow v_1 \qquad \dots \qquad \langle e_n, \sigma \rangle \rightarrow v_n}{\langle f(e_1, \dots, e_n), \sigma \rangle \rightarrow v}$$

where v is the result of the operator f applied to the values $v_1 \dots v_n$. This rule simply states that whenever the n terms above are evaluated, the function f is applied to their values and yields a result.

Example 3.2.1
Let us consider the expression `QuantumCircuit`(5,6 + 2). *Its evaluation proceed as follows: first, the constants are evaluated using the* CONST *rule, as:*

$$\frac{}{\langle 5, \sigma \rangle \Rightarrow 5}$$

Then the FUNCTION *rule is applied, to determine the result of* $6 + 2$*:*

$$\frac{\langle 6, \sigma \rangle \Rightarrow 6 \qquad \langle 2, \sigma \rangle \Rightarrow 2}{\langle 6 + 2, \sigma \rangle \Rightarrow 8}$$

Finally we can retrieve the value of ***`QuantumCircuit`*** *using its corresponding rule. The complete derivation tree is shown:*

$$\frac{\langle 5, \sigma \rangle \Rightarrow 5 \qquad \dfrac{\langle 6, \sigma \rangle \Rightarrow 6 \qquad \langle 2, \sigma \rangle \Rightarrow 2}{\langle 6 + 2, \sigma \rangle \Rightarrow 8}}{\langle \texttt{QuantumCircuit}(5{,}6 + 2), \sigma \rangle \Rightarrow (\epsilon, 5, 8)}$$

3.2.2 Commands

Commands modify the state, hence we have to define the relation between a configuration and a state, in order to represent how the state evolves. In the big-step semantics, this is formalized as: $\langle c, \sigma_1 \rangle \Rightarrow \sigma_2$, meaning that the command c evaluated in the state σ_1 will produce the state σ_2.

We will now present some of the most notable big-step inference rules for commands:

Assignment

The ASSIGN rule specifies how a variable is updated in the program state:

$$\frac{\langle e, \sigma \rangle \to v}{\langle \mathtt{x} \leftarrow e, \sigma \rangle \to \sigma[v/\mathtt{x}]}$$

It states that when an expression e gets evaluated to a value v, we update the variable in the state σ with the computed value. This is done using the notation $\sigma[v/\mathtt{x}]$

Example 3.2.2
Continuing from our earlier example, an assignment of the quantum circuit can be performed as:

$$\frac{\dfrac{\ldots}{\langle \mathtt{QuantumCircuit}(5{,}6+2), \sigma \rangle \Rightarrow (\epsilon, 5, 8)}}{\langle \mathtt{x} \leftarrow \mathtt{QuantumCircuit}(5{,}6+2), \sigma \rangle \Rightarrow \sigma[(\epsilon, 5, 8)/\mathtt{x}]}$$

Unitary

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathcal{In}, \mathcal{Qb}, \mathcal{Clb}) \qquad \langle e_1, \sigma \rangle \to m_1 \qquad \ldots \qquad \langle e_n, \sigma \rangle \to m_n}{\langle \mathtt{x}.U(e_1, \ldots, e_n), \sigma \rangle \to \sigma[(\mathcal{In}; (U(m_1, \ldots, m_n), \emptyset), qb, clb)/\mathtt{x}]}$$

This rules applies a unitary operation U to the specified qubits $m_1 \ldots m_n$, appending the operation to the circuit's instruction sequence *In*. We note that every m_i must be less or equal to the number of qubits in the circuit, in order to reference a valid qubit.

Composition

The composition rule specifies how two circuits are combined:

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathcal{In}_1, \mathcal{Qb}_1, \mathcal{Clb}_1) \qquad \langle e, \sigma \rangle \to (\mathcal{In}_2, \mathcal{Qb}_2, \mathcal{Clb}_2)}{\langle \mathtt{x.compose}(e), \sigma \rangle \to \sigma[(\mathcal{In}_1 :: \mathcal{In}_2, \mathcal{Qb}_1, \mathcal{Clb}_1)/\mathtt{x}]}$$

Here we require the first circuit being composed to have less or equal the number of qubits of the one referenced by e. The result is a circuit where the instructions are concatenated and that is referenced by `x`.

Measure

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathcal{In}, \mathcal{Qb}, \mathcal{Clb}) \qquad \langle e_1, \sigma \rangle \to n_1 \qquad \langle e_2, \sigma \rangle \to n_2}{\langle \mathtt{x.measure}(e_1, e_2), \sigma \rangle \to \sigma[(\mathcal{In}; (measure(n_1, n_2)), \mathcal{Qb}, \mathcal{Clb})/\mathtt{x}]}$$

The measurement operation is appended to the instructions and it specifies which qubit and bit are involved. Here n_1 and n_2 must reference valid qubits.

Adding Qubits

$$\frac{\langle \mathtt{x}, \sigma\rangle \rightarrow (\mathcal{In}, \mathcal{Qb}, \mathcal{Clb}) \qquad \langle e_1, \sigma\rangle \rightarrow n}{\langle \mathtt{x.addBits}(e_1), \sigma\rangle \rightarrow \sigma[(\mathcal{In}, \mathcal{Qb} + n, \mathcal{Clb})/\mathtt{x}]}$$

Finally, this rule increases the number of qubits in the circuit in x.

The complete big-step semantics is given in Figure 3.2.

Example 3.2.3

Let us expand our previous example by adding a Hadamard gate and a measure operation on the zeroth qubit. The Hadamard gate can be added using the unitary operation rule as:

$$\frac{\langle \mathtt{x}, \sigma\rangle \rightarrow (\epsilon, 5, 8) \qquad \langle 0, \sigma\rangle \rightarrow 0}{\langle \mathtt{x.H(0)}, \sigma\rangle \rightarrow \sigma[(H(0), 5, 8)/\mathtt{x}]}$$

The measure operation can be added using the measure rule:

$$\frac{\langle \mathtt{x}, \sigma\rangle \rightarrow (H(0), 5, 8) \qquad \langle 0, \sigma\rangle \rightarrow 0 \qquad \langle 1, \sigma\rangle \rightarrow 1}{\langle \mathtt{x.measure}(0, 1), \sigma\rangle \rightarrow \sigma[(H(0) :: \mathit{measure}(0, 1), 5, 8)/\mathtt{x}]}$$

VAR

$$\frac{}{\langle \mathtt{x}, \sigma \rangle \to \sigma(\mathtt{x})}$$

CONST

$$\frac{}{\langle k, \sigma \rangle \to k}$$

CIRCUIT

$$\frac{\langle e_1, \sigma \rangle \to n_1 \qquad \langle e_2, \sigma \rangle \to n_2}{\langle \mathtt{QuantumCircuit}(e_1, e_2), \sigma \rangle \to (\epsilon, n_1, n_2)}$$

FUNCTION

$$\frac{\langle e_1, \sigma \rangle \to v_1 \qquad \dots \qquad \langle e_n, \sigma \rangle \to v_n}{\langle f(e_1, \dots, e_n), \sigma \rangle \to v}$$

ASSIGN

$$\frac{\langle e, \sigma \rangle \to v}{\langle \mathtt{x} \leftarrow e, \sigma \rangle \to \sigma[v/\mathtt{x}]}$$

SEQUENCE

$$\frac{\langle c_1, \sigma_1 \rangle \to \sigma_2 \qquad \langle c_2, \sigma_2 \rangle \to \sigma_3}{\langle c_1 \ ; \ c_2, \sigma_1 \rangle \to \sigma_3}$$

IF-TRUE

$$\frac{\langle e, \sigma_1 \rangle \to \mathtt{True} \qquad \langle c_0, \sigma \rangle \to \sigma_2}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_0\ \mathtt{else}\ c_1, \sigma_1 \rangle \to \sigma_2}$$

IF-FALSE

$$\frac{\langle e, \sigma_1 \rangle \to \mathtt{False} \qquad \langle c_1, \sigma \rangle \to \sigma_2}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_0\ \mathtt{else}\ c_1, \sigma_1 \rangle \to \sigma_2}$$

WHILE-FALSE

$$\frac{\langle e, \sigma_1 \rangle \to \mathtt{False}}{\langle \mathtt{while}\ e : c, \sigma_1 \rangle \to \sigma_1}$$

WHILE-TRUE

$$\frac{\langle e, \sigma_1 \rangle \to \mathtt{True} \qquad \langle c, \sigma_1 \rangle \to \sigma_2 \qquad \langle \mathtt{while}\ e : c, \sigma_2 \rangle \to \sigma_3}{\langle \mathtt{while}\ e : c, \sigma_1 \rangle \to \sigma_3}$$

UNITARY

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathit{In}, \mathit{Qb}, \mathit{Clb}) \qquad \langle e_1, \sigma \rangle \to m_1 \qquad \dots \qquad \langle e_n, \sigma \rangle \to m_n}{\langle \mathtt{x}.U(e_1, \dots, e_n), \sigma \rangle \to \sigma[(\mathit{In} :: U(m_1, \dots, m_n), \emptyset, \mathit{Qb}, \mathit{Clb})/\mathtt{x}]}$$

COMPOSITION

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathit{In}_1, \mathit{Qb}_1, \mathit{Clb}_1) \qquad \langle e_1, \sigma \rangle \to (\mathit{In}_2, \mathit{Qb}_2, \mathit{Clb}_2)}{\langle \mathtt{x.compose}(e_1), \sigma \rangle \to \sigma[(\mathit{In}_1 :: \mathit{In}_2, \mathit{Qb}_1, \mathit{Clb}_1)/\mathtt{x}]}$$

MEASURE

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathit{In}, \mathit{Qb}, \mathit{Clb}) \qquad \langle e_1, \sigma \rangle \to n_1 \qquad \langle e_2, \sigma \rangle \to n_2}{\langle \mathtt{x.measure}(e_1, e_2), \sigma \rangle \to \sigma[(\mathit{In} :: \mathit{measure}(n_1)(n_2), \mathit{Qb}, \mathit{Clb})/\mathtt{x}]}$$

ADD-QUBITS

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathit{In}, \mathit{Qb}, \mathit{Clb}) \qquad \langle e_1, \sigma \rangle \to n}{\langle \mathtt{x.addQubits}(e_1), \sigma \rangle \to \sigma[(\mathit{In}, \mathit{Qb} + n, \mathit{Clb})/\mathtt{x}]}$$

ADD-BITS

$$\frac{\langle \mathtt{x}, \sigma \rangle \to (\mathit{In}, \mathit{Qb}, \mathit{Clb}) \qquad \langle e_1, \sigma \rangle \to n}{\langle \mathtt{x.addBits}(e_1), \sigma \rangle \to \sigma[(\mathit{In}, \mathit{Qb}, \mathit{Clb} + n)/\mathtt{x}]}$$

Figure 3.2: Big-step semantics for Proto-Qiskit

Chapter 4

Circuit Size Estimation using Proto-Qiskit's Hoare Logic

Building upon the formalization of Proto-Qiskit given in the last chapter, we now focus on the verification of circuits dimensions. To achieve this, we introduce an Hoare logic, designed to capture how circuits depth and width evolve as operations are added.

This approach necessitates of a formal logic that is able to express preconditions and postconditions. Consequently, this logic requires a type system for Proto-Qiskit, which allows us to reason about different types of values in a program.

There exist two approaches when giving a type system: static and dynamic typing. The main difference being the moment in which *type checks* are performed. Static typing ensures that the correct types are bounded at compile-time, while dynamic typing performs type checks at runtime. Proto-Qiskit will be statically typed, this is opposite to what Python does. It is done because we want to be able to reason about the types we are working with in the logic. Developing techniques to statically handle dynamic typing also introduces significant complexity while still depending on static typing to some extent.

In this chapter we provide a concise introduction to type systems and a description of the one used in Proto-Qiskit. Subsequently we focus on defining the logic for expressing assertions, we will then express the Hoare logic and finally we will end the chapter with an example of how it can be used to reason on circuits' metrics.

4.1 Type System Definition

A type system decribes the types that can be used to annotate programs and the relationship between programs and types. It also establishes rules for checking the consistency of programs and ensures that operations are applied to compatible data types. For example, consider a function f with type

$$f : \mathbf{Nat} \to \mathbf{Bool}$$

this type annotation means that it accepts a natural number as input and outputs a boolean value. Such typing ensures that the operation, designed to work on numbers doesn't end up processing other types of input.

Types in Proto-Qiskit

In Proto-Qiskit we have three possible types, which also represent all possible values in our language:

$$\mathbf{TYPES} = \{\mathsf{Nat}, \mathsf{Bool}, \mathsf{Circ}\}$$

where Nat is the type of natural numbers, Bool is the type for booleans and Circ is the type for quantum circuits.

We assume that each constant k of the language is associated to a type $\tau \in \mathbf{TYPES}$. Furthermore, functions comes with an associated type of the form $\tau_1 \ldots \tau_n \to \tau$, which reflects that the function has an arity of n and each of the input has a type τ_i.

4.1.1 Judgements

The behaviour of a type system is formally described by what is called a *judgement.* A typical judgement has the form $\Gamma \vdash t : \tau$. This means that the term t has type τ under environment Γ. The typing environment Γ is a mapping between variables and their associated types.

Judgements in Proto-Qiskit

In Proto-Qiskit, two types of judgements are defined:

- **Typing Judgement** ($\Gamma \vdash e : \tau$): this judgment states that the expression e has type τ in the environment Γ. For example it could be that: $\Gamma \vdash 0 : \mathbf{Nat}$, which means that the number 0 has type natural numbers.
- **Well Typed Judgement** ($\Gamma \vdash c$): this expresses that the command c is well-typed under the environment Γ.

4.1.2 Typing Rules

Typing rules define the validity of certain judgements on the basis of other judgements, adhering to the structure of inference rules. These rules describe how to derive more complex statements from simpler ones. Below we explain two rules, as the others are quite similar and do not require an explanation. The complete list of typing rules for Proto-Qiskit can be found in Figure 4.1.

Variable Typing Rule

$$\frac{\Gamma \vdash \mathtt{x} : \tau}{\mathtt{x} : \tau \in \Gamma}$$

This rule asserts that if a variable x is associated with a type τ in the environment Γ, then Γ is able to type x with the type τ. In other words, if the environment defines a type for the variable, than that variable is typed accordingly.

Assignment Typing Rule

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \mathtt{x} : \tau}{\Gamma \vdash \mathtt{x} \leftarrow e}$$

This rule states that if an expression is typed as τ in our environment, and also a variable x has the same type, then the variable can take the value of the expression. This is intuitively correct as a variable which type is Circ can only be assigned a circuit.

4.2 A Logic for Assertions

As we discussed before, a logic is needed to reason about the preconditions and postconditions of Hoare triples. This logic must be able to reason on Proto-Qiskit's typed expressions and commands. Specifically, the appropriate logic for this purpose is what's called a many-sorted first order logic. This logic is a generalization of traditional first order logic, which is designed to support reasoning about different types of objects. In fact, in many-sorted logic, the domain is partitioned into different *sorts*, each being a set that classifies objects into different types.

4.2.1 Many-sorted Logic

The logic we employ has as sorts the base types of Proto-Qiskit presented before. From now on, in order to distinguish them, we will identify *program* variables, which appear in Proto-Qiskit's programs, as $\mathtt{x}, \mathtt{y}, \mathtt{z}$ and *logic* variables as x, y, z. As this is a first order logic, its syntax is standard and it is given in Figure 4.2.

While the syntax remains consistent with first order logic, the *signature* is what changes.

The signature, written as $\mathcal{S}$, describes the non-logical symbols, we will write $\mathcal{S}$ with an apex to identify which element of it we are describing.

$$\textsc{T-Const}\ \frac{}{\Gamma \vdash k : \tau} \qquad \textsc{T-Var}\ \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau} \qquad \textsc{T-Function}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f^n(e_0, \dots, e_n) : \tau}$$

$$\textsc{T-Circuit}\ \frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{QuantumCircuit}(e_1,e_2) : \mathsf{Circ}}$$

$$\textsc{T-Assign}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \mathtt{x} : \tau}{\Gamma \vdash \mathtt{x} \leftarrow e} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1 \ ;\ c_2} \qquad \textsc{T-Conditional}\ \frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2}$$

$$\textsc{T-While}\ \frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash c}{\Gamma \vdash \mathtt{while}\ e : c} \qquad \textsc{T-Unitary}\ \frac{\Gamma \vdash \mathtt{x} : \mathsf{Circ} \qquad \Gamma \vdash e_1 : \mathsf{Nat} \quad \dots \quad \Gamma \vdash e_n : \mathsf{Nat}}{\Gamma \vdash \mathtt{x}.U(e_1, \dots, e_n)}$$

$$\textsc{T-Compose}\ \frac{\Gamma \vdash \mathtt{x} : \mathsf{Circ} \qquad \Gamma \vdash e : \mathsf{Circ}}{\Gamma \vdash \mathtt{x.compose}(e)} \qquad \textsc{T-Measure}\ \frac{\Gamma \vdash \mathtt{x} : \mathsf{Circ} \qquad \Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{x.measure}(e_1, e_2)}$$

$$\textsc{T-Add-Qubits}\ \frac{\Gamma \vdash \mathtt{x} : \mathsf{Circ} \qquad \Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathtt{x.addQubits}(e)} \qquad \textsc{T-Add-Bits}\ \frac{\Gamma \vdash \mathtt{x} : \mathsf{Circ} \qquad \Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathtt{x.addBits}(e)}$$

Figure 4.1: Typing rules of Proto-Qiskit

$$\begin{aligned}
\phi &::= P^n(e_1, \dots, e_n) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall y : \tau.\, \phi \mid \exists y : \tau.\, \phi \\
e &::= y \mid \mathtt{x} \mid k \mid f^n(e_1, \dots, e_n).
\end{aligned}$$

Figure 4.2: Many-sorted first order logic

- The set $\mathcal{S}^{\mathcal{C}}$ contains the Proto-Qiskit's language constants k as natural numbers and boolean values. In order to identify them, we assume each constant to come with a defined type τ.
- The set $\mathcal{S}^{\mathcal{P}}$ include an equality symbol $=_\tau$ for each type τ. A predicate comes with a type of the form $\tau_1 \times \cdots \times \tau_n$, where $\tau_1 \dots \tau_n \in$ **TYPES**.
- The set $\mathcal{S}^{\mathcal{F}}$ include symbols for Proto-Qiskit's functions, but also some functions symbols for the circuits metric we are interested in. Each function f^n comes with a type of the form $\tau_1 \times \cdots \times \tau_n \to \tau$ for $\tau_1 \dots \tau_n, \tau \in$ **TYPES**

The functions we are more interested in are the ones that operates on quantum circuits. Some of them correspond to the results of the methods in Proto-Qiskit. These are:

- $\mathit{QuantumCircuit}$: $\mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Circ}$, which correspond to the circuit constructor in Proto-Qiskit.
- append^U: $\mathsf{Circ} \times \mathsf{Nat}^n \to \mathsf{Circ}$, there exist one per unitary operation U with n arguments.
- $\mathit{measure}$: $\mathsf{Circ} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Circ}$.
- $\mathit{compose}$: $\mathsf{Circ} \times \mathsf{Circ} \to \mathsf{Circ}$.
- $\mathit{addQubits}$: $\mathsf{Circ} \times \mathsf{Nat} \to \mathsf{Circ}$.
- $\mathit{addBits}$: $\mathsf{Circ} \times \mathsf{Nat} \to \mathsf{Circ}$.

Other than these functions, we define two more, which correspond to the metrics we are interested in, those are:

- **width**: $\mathsf{Circ} \to \mathsf{Nat}$, which returns the width of a circuit.
- **gatecount**: $\mathsf{Circ} \to \mathsf{Nat}$, which returns the number of gates in a circuit.
- **depth**: $\mathsf{Circ} \times \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$, which returns the depth of the circuit. The three inputs correspond respectively to the circuit being analyzed, and the input and output qubit from which to compute the depth.

4.2.2 The Logic Type System

As we gave a notion of typing for Proto-Qiskit's terms, we can also define a notion of typing for the logic we gave. In order to do so, we employ the use of two typing contexts to distinguish between the program variables and the logic variables.

Similarly to what we did before we also have two types of judgements:

- **Typing judgement** ($\Gamma; \Delta \vdash e : \tau$): which states that e is given type τ under program typing context Γ and logic typing context Δ.
- **Well-typed judgment** ($\Gamma; \Delta \vdash \phi$): which asserts that the formula ϕ is well typed under typing environment Γ and logic context Δ.

The complete set of typing rules for the logic is given in Figure 4.3. These rules for typing expressions in the logic are designed to be more general than the rules for typing expressions in Proto-Qiskit. This is intentional. Since the logic is meant to analyse and reason about Proto-Qiskit's programs, it should be possible to derive types for Qiskit expressions.

$$\text{L-CONST}\quad \frac{}{\Gamma; \Delta \vdash k : \tau_k} \qquad \text{L-PROGRAM-VAR}\quad \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma; \Delta \vdash \mathtt{x} : \tau} \qquad \text{L-VAR}\quad \frac{y : \tau \in \Delta}{\Gamma; \Delta \vdash y : \tau}$$

$$\text{L-FUNCTION}\quad \frac{f^n : \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma; \Delta \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma; \Delta \vdash e_n : \tau_n}{\Gamma; \Delta \vdash f^n(e_1, \ldots, e_n) : \tau}$$

$$\text{L-TOP}\quad \Gamma; \Delta \vdash \top \qquad \text{L-BOTTOM}\quad \Gamma; \Delta \vdash \bot$$

$$\text{L-PREDICATE}\quad \frac{P^n : \tau_1 \times \cdots \times \tau_n \qquad \Gamma; \Delta \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma; \Delta \vdash e_n : \tau_n}{\Gamma; \Delta \vdash P^n(e_1, \ldots, e_n)}$$

$$\text{L-CONJUNCTION}\quad \frac{\Gamma; \Delta \vdash \phi_0 \qquad \Gamma; \Delta \vdash \phi_1}{\Gamma; \Delta \vdash \phi_0 \wedge \phi_1} \qquad \text{L-DISJUNCTION}\quad \frac{\Gamma; \Delta \vdash \phi_0 \qquad \Gamma; \Delta \vdash \phi_1}{\Gamma; \Delta \vdash \phi_0 \vee \phi_1} \qquad \text{L-NEGATION}\quad \frac{\Gamma; \Delta \vdash \phi}{\Gamma; \Delta \vdash \neg\phi}$$

$$\text{L-UNIVERSAL}\quad \frac{\Gamma; \Delta, y : \tau \vdash \phi}{\Gamma; \Delta \vdash \forall y : \tau .\, \phi} \qquad \text{L-EXISTENTIAL}\quad \frac{\Gamma; \Delta, y : \tau \vdash \phi}{\Gamma; \Delta \vdash \exists y : \tau .\, \phi}$$

Figure 4.3: Typing rules for the many-sorted logic

4.2.3 The Semantics of the Logic

Given the syntax for the logic and its typing system, we have to now define a semantics for the logic. This is done to establish the meaning of its components, especially the ones which do not have a counterpart in Proto-Qiskit but are native to the logic. We will give this semantics using the symbol $[\![\cdot]\!]$. These brackets, so-called semantic evaluation brackets, maps an expression to its denotation, or semantic value.

We start by giving an interpretation for types and for both the program typing context and the logic typing context.

Types

Base types are interpreted as the sets of values they represent, while functions and predicate types are interpreted as sets of relations and functions respectively.

$$[\![\mathsf{Nat}]\!] = \mathbb{N} \qquad [\![\mathsf{Bool}]\!] = \mathbb{B} \qquad [\![\mathsf{Circ}]\!] = \mathsf{Circ}$$

$$[\![\tau_0 \times \cdots \times \tau_n]\!] = [\![\tau_0]\!] \times \cdots \times [\![\tau_n]\!] \qquad [\![\tau_0 \times \cdots \times \tau_n \to \tau]\!] = [\![\tau]\!]^{[\![\tau_0]\!] \times \cdots \times [\![\tau_n]\!]}$$

Typing Contexts

A context environment Γ holds variables and their corresponding types. An interpretation can be given as:

$$[\![\Gamma]\!] = \{\sigma \mid \forall \mathtt{x} : \tau \in \Gamma .\, \sigma(\mathtt{x}) \in [\![\tau]\!]\}$$

Which means that the environment Γ interpretation correspond to the set of all states in which every variable of a type τ is mapped to a value in the semantic interpretation of that type.

Similarly, we define the interpretation of a logic typing context Δ as a set of *logic variable substitutions*. This means that a value is substituted to a logical variable, which turns an abstract logical statement into one that can be evaluated. The interpretation become:

$$[\![\Delta]\!] = \{\delta \mid \forall y : \tau \in \Delta .\, \delta(y) \in [\![\tau]\!]\}$$

Which means that the interpretation of the logic typing context Δ, correspond to the set of all substitutions δ in which every logical variable of type τ in the envorinment Δ when mapped to a value, this value has to be in the interpretation of type τ.

We now are able to define the semantics for both logic expressions and logic formulae.

Interpretation of Expressions and Formulae

A logic expression, is a Proto-Qiskit expression, which now may contain a logic variables. A formula ϕ instead, talks about the current state of program and logic variables. We only interpret well-typed expressions and formulae, which means that their interpretation is defined on typing judgments. This interpretation relies on the assignment of domain elements to both program variables (represented in the store σ) and logic variables (represented in the store δ).

Definition 4.2.1 (Semantics of Logic Expressions)**.** We define the interpretation of a well typed expression $\Gamma; \Delta \vdash e : \tau$ as a function $[\![\Gamma; \Delta \vdash e : \tau]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to [\![\tau]\!]$.

Specifically:

$$\begin{aligned}
[\![\Gamma;\Delta \vdash k : \tau]\!]_{\sigma,\delta} &= [\![k]\!] \\
[\![\Gamma;\Delta \vdash \mathtt{x} : \tau]\!]_{\sigma,\delta} &= \sigma(\mathtt{x}) \\
[\![\Gamma;\Delta \vdash y : \tau]\!]_{\sigma,\delta} &= \delta(y) \\
[\![\Gamma;\Delta \vdash f^n(e_1,\dots,e_n) : \tau]\!]_{\sigma,\delta} &= [\![f^n]\!]([\![\Gamma;\Delta \vdash e_1 : \tau_1]\!]_{\sigma,\delta},\dots,[\![\Gamma;\Delta \vdash e_n : \tau_n]\!]_{\sigma,\delta})
\end{aligned}$$

This is intuitive, as the interpretation is the same that we gave in Proto-Qiskit's semantics.

A formula ϕ, can be satisfied or not by a store and a logic substitution, therefore we define the interpretation of a formula as the set of pairs (σ, δ) that satisfy it.

Definition 4.2.2 (Semantics of Logic Formulae)**.** We define the interpretation of a well-formed formula $\Gamma;\Delta \vdash \phi$ as a relation $[\![\Gamma;\Delta \vdash \phi]\!] \subseteq [\![\Gamma]\!] \times [\![\Delta]\!]$. It is defined as:

$$\begin{aligned}
[\![\Gamma;\Delta \vdash \top]\!] &= [\![\Gamma]\!] \times [\![\Delta]\!] \\
[\![\Gamma;\Delta \vdash \bot]\!] &= \emptyset \\
[\![\Gamma;\Delta \vdash P^n(e_1,\dots,e_n)]\!] &= \{(\sigma,\delta) \mid ([\![\Gamma;\Delta \vdash e_1 : \tau_1]\!]_{(\sigma,\Delta)} \dots [\![\Gamma;\Delta \vdash e_n : \tau_n]\!]_{(\sigma,\Delta)}) \in [\![P^n]\!]\} \\
[\![\Gamma;\Delta \vdash \phi \wedge \psi]\!] &= [\![\Gamma;\Delta \vdash \phi :]\!] \cap [\![\Gamma;\Delta \vdash \psi :]\!] \\
[\![\Gamma;\Delta \vdash \phi \vee \psi]\!] &= [\![\Gamma;\Delta \vdash \phi :]\!] \cup [\![\Gamma;\Delta \vdash \psi :]\!] \\
[\![\Gamma;\Delta \vdash \neg\phi]\!] &= [\![\Gamma]\!] \times [\![\Delta]\!] \diagdown [\![\Gamma;\Delta \vdash \phi :]\!] \\
[\![\Gamma;\Delta \vdash \forall y : \tau.\, \phi]\!] &= \{(\sigma,\delta) \mid \forall v \in [\![\tau]\!].(\sigma,\delta[y \mapsto v])\} \\
[\![\Gamma;\Delta \vdash \exists y : \tau.\, \phi]\!] &= \{(\sigma,\delta) \mid \exists v \in [\![\tau]\!].(\sigma,\delta[y \mapsto v])\}
\end{aligned}$$

Function Symbols

When describing the semantics of symbols at the logic level, we can omit the symbols which are interpreted in the same way as in Proto-Qiskit semantics, and the ones that are intuitive, such as the *Arithmetic*, *Boolean* and *Comparison* operators. Other function symbols interpretation however, deserve to be discussed. We start with the symbols for circuit building functions:

$$\begin{aligned}
[\![\mathit{QuantumCircuit}]\!](qb, cb) &= (\epsilon, qb, cb) \\
[\![\mathit{append}^U]\!]((\mathit{Instr}, qb, cb), q_1,\dots,q_n) &= (\mathit{Instr} :: (U(q_1,\dots,q_n)), qb, cb) \\
[\![\mathit{measure}]\!]((\mathit{Instr}, qb, cb), q, c) &= (\mathit{Instr} :: (\mathit{measure}(q,c)), qb, cb) \\
[\![\mathit{compose}]\!]((\mathit{Instr}, q_1, c_1), (\mathit{Instr}, q_2, c_2)) &= (\mathit{Instr}_1 :: \mathit{Instr}_2, q_1, c_1) \\
[\![\mathit{addQubits}]\!]((\mathit{Instr}, q, c), n) &= (\mathit{Instr}, q+n, c) \\
[\![\mathit{addBits}]\!]((\mathit{Instr}, q, c), n) &= (\mathit{Instr}, q, c+n)
\end{aligned}$$

Where with q and c we identify single qubits on which operations are performed, and with qb, cb the number of qubits and classical bits in the circuit.

We can now give a definition for the size operations introduced before, as:

$$\begin{aligned}
[\![\mathit{width}]\!]((\mathit{Instr}, qb, cb)) &= qb + cb \\
[\![\mathit{gatecount}]\!]((\mathit{Instr}, qb, cb)) &= |\mathit{Instr}|
\end{aligned}$$

$$\begin{aligned}
[\![\mathit{depth}]\!]((\epsilon, qb, cb), q_{in}, q_{out}) &= 0 \\
[\![\mathit{depth}]\!]((\mathit{Instr} :: (op(q_1 \dots q_n)(\dots)), qb, cb), q_{in}, q_{out}) &= \begin{cases} 1 + \hat{D} & \text{if } q_{out} \in \{q_1, \dots, q_n\} \\ D & \text{otherwise} \end{cases}
\end{aligned}$$

where $|\mathit{Instr}|$ represent the lenght of a sequence of instructions, and we define

$$\begin{aligned}
&\hat{D} = \max\{[\![\mathit{depth}]\!]((\mathit{Instr}, qs, cs), q_{in}, q_{mid}) \mid q_{mid} \in \{q_1, \dots, q_n\}\} \\
&D = [\![\mathit{depth}]\!]((\mathit{Instr}, qs, cs), q_{in}, q_{out})
\end{aligned}$$

As the width is similar to what we have defined, depth seems completely different. Depth is defined recursively based on the qubits involved in the computation. If the circuit is composed of only itself, the depth is zero. If the sequence of instruction however is not empty, the depth is defined in terms of what qubits are interested by the computation. A qubit which is, will have depth of one greater than the maximum depth of the operation's inputs and a qubit which is not will have instead the same depth as before.

4.3 A Hoare Logic for Proto-Qiskit

Finally, we can devise a Hoare Logic for analyzing Proto-Qiskit's programs.

We start by defining what are the triples of this logic. A triple has the form

$$\{\phi\}\, p\, \{\psi\}$$

where ϕ and ψ are formulae in the first order logic we defined, and p is a Proto-Qiskit program. By using the many-sorted logic, ϕ and ψ reason about circuits and their measures, therefore deriving this kind of triples allow us to reason on the dimension of the circuit built in p.

The derivability of a triple is given as:

$$\vdash_\Gamma \{\phi\}\, p\, \{\psi\}$$

Note that we keep the typing context as some rules require it in order to know the type of variables that appear in p. One of these rules is the CONSEQUENCE rule, where we need to check the validity of the implication in order to apply it.

The complete set of derivation rules for Hoare triples can be found in Figure 4.4

$$\textsc{Hoare-Assign}\quad \frac{}{\vdash_\Gamma \{\phi[e/\mathtt{x}]\}\, \mathtt{x} \leftarrow e\, \{\phi\}}$$

$$\textsc{Hoare-Seq}\quad \frac{\vdash_\Gamma \{\phi\}\, c_1\, \{\xi\} \qquad \vdash_\Gamma \{\xi\}\, c_2\, \{\psi\}}{\vdash_\Gamma \{\phi\}\, c_1 \mathbin{;} c_2\, \{\psi\}}$$

$$\textsc{Hoare-Conditional}\quad \frac{\vdash_\Gamma \{\phi \wedge e\}\, c_1\, \{\psi\} \qquad \vdash_\Gamma \{\phi \wedge \neg e\}\, c_2\, \{\psi\}}{\vdash_\Gamma \{\phi\}\, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\, \{\psi\}}$$

$$\textsc{Hoare-While}\quad \frac{\vdash_\Gamma \{\phi \wedge e\}\, c\, \{\phi\}}{\vdash_\Gamma \{\phi\}\, \mathtt{while}\ e : c\, \{\phi \wedge \neg e\}}$$

$$\textsc{Hoare-Unitary}\quad \frac{}{\vdash_\Gamma \{\phi[\mathit{append}^U(\mathtt{x}, e_1, \ldots, e_n)/\mathtt{x}]\}\, \mathtt{x}.U(e_1, \ldots, e_n)\, \{\phi\}}$$

$$\textsc{Hoare-Compose}\quad \frac{}{\vdash_\Gamma \{\phi[\mathit{compose}(\mathtt{x}, e)/\mathtt{x}]\}\, \mathtt{x.compose}(e)\, \{\phi\}}$$

$$\textsc{Hoare-Measure}\quad \frac{}{\vdash_\Gamma \{\phi[\mathit{measure}(\mathtt{x}, e_1, e_2)/\mathtt{x}]\}\, \mathtt{x.measure}(e_1, e_2)\, \{\phi\}}$$

$$\textsc{Hoare-Add-Qubits}\quad \frac{}{\vdash_\Gamma \{\phi[\mathit{addQubits}(\mathtt{x}, e)/\mathtt{x}]\}\, \mathtt{x.addQubits}(e)\, \{\phi\}}$$

$$\textsc{Hoare-Add-Bits}\quad \frac{}{\vdash_\Gamma \{\phi[\mathit{addBits}(\mathtt{x}, e)/\mathtt{x}]\}\, \mathtt{x.addBits}(e)\, \{\phi\}}$$

$$\textsc{Hoare-Consequence}\quad \frac{\vDash_{\Gamma;\emptyset} \phi \Rightarrow \phi' \qquad \vdash_\Gamma \{\phi'\}\, c\, \{\psi'\} \qquad \vDash_{\Gamma;\emptyset} \psi' \Rightarrow \psi}{\vdash_\Gamma \{\phi\}\, c\, \{\psi\}}$$

Figure 4.4: Derivation rules for Proto-Qiskit's Hoare triples

4.3.1 An Example

We now demonstrate how to perform the structural analysis of properties on a circuit, using the defined Hoare logic. For this purpouse we will use a simple circuit, written in Proto-Qiskit's syntax.

Consider the program:

```
qc ← QuantumCircuit(3, 3);
qc.H(0);
qc.CX(0, 1);
qc.H(0);
qc.X(2);
qc.CX(1,2);
qc.measure(0,0);
qc.measure(1,1);
```

Let us refer to it as p. We aim to derive the following triple:

$$\{\top\}\, p\, \{gatecount(qc) = 7\}$$

Where qc is the quantum circuit constructed by p.

We start by using the HOARE-SEQ rule to split the last statement of the program from the rest.This yields:

$$\frac{\vdash_{qc:\mathsf{Circ}} \{\top\}\, p'\, \{gatecount(measure(qc,1,1)) = 7\} \qquad \vdash_{qc:\mathsf{Circ}} \{gatecount(measure(qc,1,1))\}\, c.measure(1,1)\, \{gatecount(qc) = 7\}}{\vdash_{qc:\mathsf{Circ}} \{\top\}\, p'\ ;\ c.measure(1,1)\, \{gatecount(qc) = 7\}}$$

This indicates that in order to prove that qc has a gatecount of 7, we must prove that the gatecount of qc in p' plus the effect of the last statement, is still 7.

In order to prove the second of these triple, we can make use of the HOARE-MEASURE rule.

The same reasoning can be applied iteratively, separating each statement in p and justifying the second triple at each step. This process stops when we reach the circuit'initialization. There we will need to prove that

$$\{\top\}\, qc \leftarrow QuantumCircuit(3,3)\, \{gatecount(e) = 7\}$$

where e is :

$$e = measure(measure(append^{CX}(append^{X}(append^{H}(\\ append^{CX}(append^{H}(qc,0),0,1),0),2),1,2)0,0),1,1)$$

The critical step is to now show that $\top \Rightarrow gatecount(e[QuantumCircuit(3,3)/qc]) = 7$, that is, showing that the formula $gatecount(e[QuantumCircuit(3,3)/qc]) = 7$ is valid. By evaluating the expression $e[QuantumCircuit(3,3)/qc]$, we obtain the Proto-Qiskit circuit where:

$$Instr = \texttt{hadamard(0); cnot(0,1); hadamard(0); x(2);} \\ \texttt{cnot(1,2); measure(0,0); measure(0,0)}$$

The number of qubits and bits in the circuit are both 3. Since the gatecount is expressed as $|Instr|$, and the number of instruction is 7, we can conclude that we were able to derive the initial triple. Similarly we could produce proofs for triples regarding the *width* and *depth* of the circuit.

Conclusion

In this thesis we ventured in the realm of quantum computing, with a particular emphasis on verifying structural properties of quantum circuits. The exploration revolved around the integration of formal methods, specifically Hoare logic for the analysis of quantum circuits.

We began by successfully defining a formalization for Qiskit, one of the most prominent quantum imperative programming environment. This formalization was not limited to the semantics of the language but was extended with types. This provide Proto-Qiskit with a rigorous system to prove properties like type safety and soundess.

The succsessful definition of a Hoare logic instead marks a milestone in this thesis. Our framework enables the derivation of meaningful triples that verifies the size of quantum circuits. This contribution lays the groundwork for a deeper and more structured understanding of quantum algorithms developed within the Qiskit ecosystem. It represents a concrete step towards bridging the gap between theoretical formalisms and practical quantum programming tools.

Looking ahead this work opens several promising ways for further exploration. One of these include formally analyzing the Qiskit's optimizations using a Hoare logic. This would allow us to understand how the size of circuits varies in relation to their initial dimensions.

In summary, this thesis represents a small step towards addressing the complexities of quantum circuit verification. By contributing to this field, we aim to support the development of more robust and formally defined quantum computing technologies.

Bibliography

[Bec] Olga Becci. *Integrating Dynamic Lifting into Qiskit.* PhD thesis.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[Dev] Qiskit Developers. Qiskit documentation. https://docs.quantum-computing.ibm.com/. Online, Accessed: 10/12/2024.

[GM10] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., USA, 1992.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[Plo04] Gordon Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 07 2004.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.