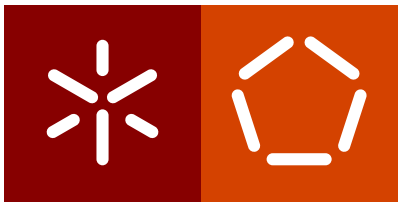
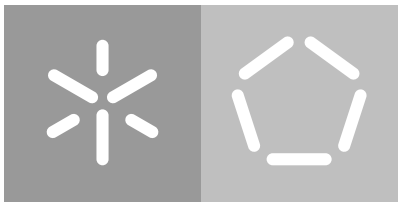


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tomás Barros Carneiro

iQbricks: Integration of a fully-featured quantum language in the framework Qbricks

May 2023



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tomás Barros Carneiro

iQbricks: Integration of a fully-featured quantum language in the framework Qbricks

Master dissertation
Integrated Master Degree in Information Physics

Dissertation supervised by
Renato Neves
Pedro Rangel Henriques

May 2023

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

ACKNOWLEDGEMENTS

I would like to express my heartfelt appreciation to my supervisors, Pedro Henriques and Renato Neves, for their invaluable patience, and feedback. My journey would not have been possible without Christophe Chareton, who provided me with generous guidance, knowledge, and expertise throughout my internship. I am also deeply grateful to CEA for their generous support, which made this incredible opportunity possible.

I would like to extend my thanks to the rest of the Qbricks team for their constant help, feedback, and moral support. I am also thankful to the researchers and work colleagues I met during talks and seminars, who have impacted and inspired me.

Lastly, I want to express my sincere gratitude to my family and friends. Their unwavering belief in me has kept my spirits and motivation high during this process.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Tomás Barros Carneiro

_____

ABSTRACT

Quantum Computing has noticeably grown over the last two decades, making it a revolutionary field of investigation in the current era of technological research.
Such a growth has been leading to an increasing demand in research by several big enterprises such as *IBM*, *Google* and *Microsoft*, paving the way for a richer ecosystem and untold benefits among the Quantum Computing community.
Verification is a crucial aspect of software development, as it ensures that a program performs as intended and reduces the risk of introducing errors. This is especially important in the field of Quantum Computing, where the complexity of programs is high and the behavior of quantum systems is often counterintuitive. Verification of quantum programs can help detect errors that may lead to incorrect results, which is of utmost importance when dealing with quantum algorithms and quantum simulations. As a result, having a formal verification framework for quantum programs can greatly benefit the development of reliable and accurate quantum software. Qbricks is a *verification framework* for building *quantum programs*, and corresponds to the framework on which this project has been integrated. During the course of this thesis, iQbricks – an intuitive and user-friendly language to build and formally verify quantum programs – was developed, along with a framework to translate and generate verifiable Qbricks programs from iQbricks.
This project's main achievements were: (1) the design and implementation of a high-level programming language for describing quantum circuits in an intuitive and user-friendly way and (2) the implementation of a translator, embedded in Qbricks' framework, that converts iQbricks programs to Qbricks ones.
The developed framework was evaluated against two different quantum algorithms: the *Quantum Fourier Transform* and *Grover's algorithm*.
This project was accompanied by an internship at the *Commissariat à l'énergie atomique et aux énergies alternatives (CEA) - LSL*, where this implementation was developed in direct involvement with Qbricks' team of investigators.

Keywords: Quantum Computing, Qbricks, formal verification, quantum programming language, *Quantum Fourier Transform*, *Grover's algorithm*

RESUMO

A Computação Quântica cresceu de forma notável nas últimas duas décadas, tornando-se um campo revolucionário de investigação na atual era da investigação tecnológica. Tal crescimento tem levado a uma crescente procura na investigação por parte de várias grandes empresas como a *IBM*, *Google* e *Microsoft*, abrindo caminho para um ecossistema mais rico e benefícios inéditos entre a comunidade da Computação Quântica.

A verificação é um aspecto crucial no desenvolvimento de software, pois garante que um programa execute conforme o previsto e reduz o risco de introduzir erros. Isso é especialmente importante no campo da computação quântica, onde a complexidade dos programas é alta e o comportamento dos sistemas quânticos é frequentemente contra-intuitivo. A verificação de programas quânticos pode ajudar a detectar erros que possam levar a resultados incorretos, o que é de extrema importância ao lidar com algoritmos quânticos e simulações quânticas. Como resultado, ter um framework de verificação formal para programas quânticos pode beneficiar grandemente o desenvolvimento de software quântico confiável e preciso. Qbricks é uma estrutura de verificação para a construção de programas quânticos, e corresponde à estrutura sobre a qual este projecto foi integrado. Durante o curso desta tese, foi desenvolvida iQbricks - uma linguagem intuitiva e de fácil utilização para construir e verificar formalmente programas quânticos - juntamente com uma ferramenta para traduzir e gerar programas Qbricks verificáveis a partir de iQbricks.

As principais concretizações deste projecto foram: (1) a concepção e implementação de uma linguagem de programação de alto nível para descrição de circuitos quânticos de uma forma intuitiva e de fácil utilização e (2) a implementação de um tradutor, embutido em Qbricks, que converte programas iQbricks para Qbricks. A ferramenta desenvolvida foi testada utilizando dois algoritmos quânticos diferentes: o Transformada de Fourier Quântica *(QFT)* e algoritmo de *Grover*.

Este projecto foi acompanhado por um estágio no *Commissariat à l'énergie atomique et aux énergies alternatives (CEA) - LSL*, onde esta implementação foi desenvolvida em colaboração direta com a equipa de investigadores de Qbricks.

Palavras-chave: Computação Quântica, Qbricks, verificação formal, linguagem para programação quântica, Transformada de Fourier Quântica, algoritmo de *Grover*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

This dissertation presents and discusses a Master's Project in Physics Engineering, Information Physics branch. The current chapter showcases: the main goals of this project, contributions brought by the work done and the structure of this document.

1.1 MOTIVATION AND CONTEXT

From coherently handling only a few qubits to dozens of them currently, viable quantum computers seem closer to us than ever. Actually there is a general belief that we are on the verge of a '*second quantum revolution*', as signaled for example by the quantum computers Sycamore and Jiuzhang which provided for the first time experimental evidence of the so-called '*quantum computational advantage*' (Arute et al., 2019; Zhong et al., 2020). In addition, 2022's Nobel prize in physics was awarded to three of the pioneers in quantum information science, for experiments with entangled photons, establishing the violation of Bell inequalities. Such a revolution raises the hope that many quantum algorithms will soon move from the scientist's or engineer's notebook into actual quantum computers, which would revolutionise different areas of society, from medicine to metrology (Preskill, 2018).

In this context, the overarching goal of the project is to provide mechanisms for checking whether a given quantum algorithm will behave as intended when running on a quantum computer. This is both interesting and challenging: many of the methods used in classical programming for checking that algorithms behave as intended simply break when subjected to quantum laws. For example, debugging is not applicable because to inspect the state of a quantum program at the middle of its execution ruins its end result. On the other hand, the well-known method of *deductive verification* is still applicable. Formal methods and deductive verification offer a wide range of techniques aiming at proving the correctness of a system with absolute, mathematical guarantee -— reasoning over all possible inputs, with methods drawn from logic, automated reasoning and program analysis (Chareton et al., 2021). The advantages of having a *formally verified* program are numerous. First, it ensures that the program is free from errors that could lead to unintended behavior or security vulnerabilities. Additionally, it can increase confidence in the correctness of the program, which is especially

important for critical systems that must operate correctly in high-stakes environments, such as quantum computers. Finally, it can serve as a form of documentation, making it easier to understand the program's behavior and ensuring that it remains consistent as the program evolves over time. Thus even if more expensive, formal verification offers an alternative to testing, it enables parametric proof certificates that guarantee program correctness.

In deductive verification, programs are typically *annotated* with *logical assertions*, such as *pre* and *post*-conditions for operations or *loop invariants*. Then *proof obligations* are (semi-) automatically generated, in such a way that proving them ensures that the logical assertions hold along any execution of the program (Chareton et al., 2021). A deductive verification framework for quantum algorithms, called Qbricks, was proposed in (Chareton et al., 2021) – and remarkably the latter was already used for successfully verifying important quantum algorithms, such as quantum phase estimation (QPE) and Grover's search. Another similar solution for formally proved correct quantum programs is the language Sqir (Hietala et al., 2021b,a) which is being developed concurrently with Qbricks.
Qbricks adopts the *circuit formalism* as the quantum computational model. In this sense, it is a formal framework for developing quantum programs in the form of circuits, from their design to their verification. This framework serves as the foundation for the the main contributions of this dissertation as detailed next.

1.2 Contributions

Qbricks' quantum programming language is still minimal (*i.e.* not rich in features) and it is thus neither user-friendly nor close to the way quantum algorithms are typically described. This hinders the design of complex quantum algorithms within the framework and its wide adoption by the everyday quantum programmer. The goal of the project is to integrate in the framework Qbricks a more feature-rich and user-friendly quantum programming language, in the spirit of IBM's famous language Qiskit Cross (2018). By introducing a quantum programming language which allows for deductive verification, in addition to offering an alternative to testing, it has in principle the decisive additional advantages to both enable parametric proof certificates and offer once-for-all absolute guarantees for the correctness of quantum programs.
The primary contributions of this project are:

1. The design and implementation of a high-level programming language for describing quantum circuits in an intuitive and user-friendly way.

2. An embedded layer in Qbricks' framework that allows users to write a quantum program using the newly designed iQbricks and translate it to Qbricks' language which can then be verified via *SMT* solvers such as *Z3* and *Alt-Ergo*.

3. An AST for quantum circuits, it (1) roots the connection between QBRICKS and IQBRICKS, (2) enables to connect the circuit object language with a specification language and (3) opens way for point 4.

4. The framework which results from the development of points 1 and 2 can be expanded to other quantum programming languages in the future, by reusing the developed data structures in the translation process.

The full implementation for this project can be consulted in Carneiro (2023).

During the course of this thesis, an internship was conducted for the duration of six months at *CEA Paris-Saclay*, where this project was developed in collaboration with QBRICKS' research group. This allowed for a more overall optimised implementation, resulting from the daily-basis communication and interaction with developers working on different layers of QBRICKS' framework. As different steps were being taken during this implementation, they were thoroughly discussed amongst the research group, whose feedback always contributed to the development process.

1.3 DOCUMENT'S STRUCTURE

The current chapter just gave a general overview of this M.Sc. project. The subsequent chapters provide a more detailed report, specifically:

1. Chapter 2 briefly introduces the topics that serve as basis of this project: namely, quantum computation, the QBRICKS framework, and language processing;

2. Chapter 3 introduces the new language and provides a thorough comparison against other quantum programming languages;

3. Chapter 4 describes the general architecture of the project's implementation. Data manipulation processes involving the creation of different *AST*'s are also thoroughly covered;

4. Chapter 5 illustrates our implementation at work. Specifically it presents two well-known quantum algorithms/subroutines written in our language – *Quantum Fourier Transform* and *Grover algorithm* – and the verification process concerning some of their properties via the translation implemented in this project;

5. Chapter 6 discusses future work prospects and concludes.

2

STATE OF THE ART

2.1 QUANTUM COMPUTING

Modern technology is present in most parts of our lives and typically recurs to different aspects of computation - in fact, from simple devices to more complex ones, there is almost always some form of (classical) computation involved. At the most fundamental level, such devices store digital information in the form of bits - 0's and 1's - which are then processed by different algorithms with a certain task at hand. During the last few decades, there has been tremendous progress in computational performance, allowing computers to perform operations faster and more efficiently. However, when faced with highly complex problems - with lots of variables interacting in complicated ways - scientists and engineers still need to turn to supercomputers - very large classical devices, often with thousands of classical CPU and GPU cores. Even then, supercomputers struggle to solve certain kinds of problems due to the *fundamental nature of classical computing*. Not only this, quantum effects are beginning to interfere in the functioning of electronic devices as they are made smaller and smaller. There are several leading fields of study in which quantum computing can have a positive impact, such as the **simulation** of chemical reactions – contributing to drug manufacturing, molecular research, **optimisation** of computational processes – applicable to the financial industry or any organization dealing with logistics optimization – and quantum **machine learning** – with countless applications in areas like image recognition, training neural networks and fraud detection.

2.1.1 *Historical background*

The problems mentioned above can be tackled to some extent by the use of a new type of computation brought up to attention by Richard Feynman in Feynman (1982). Back then, following great breakthroughs in quantum physics, by scientists such as Erwin Schrödinger and Niels Bohr, it was already apparent that our universe was quantum mechanical. Facing this idea, Feynman propounded the question: *What kind of computer are we going to use to simulate physics?* His interest was to simulate the physical world but, as mentioned above, this

world was a quantum mechanical one. He came to realize that due to the hidden-variable problem (Bell, 1964), it was almost impossible to represent the results of quantum mechanics with a classical universal device. Indeed, the amount of data generated in order to simulate quantum probabilities was just too heavy to store. His suggestion was a Quantum Computer, *i.e.* a computer that fully explores the quantum mechanical laws. Feynman believed that with a suitable class of such quantum machines any quantum system could be efficiently simulated.

The fact that classical computers cannot efficiently simulate quantum ones (Feynman, 1982) suggests that the latter, at least at a theoretical level, may offer a fundamental speed advantage. In fact we know by now that this speed advantage is so remarkable that many researchers believe no conceivable amount of progress in classical computation will be able to overcome the gap between the power of a classical computer and the power of a quantum one Nielsen and Chuang (2010). David Deutsch was one of the first scientists to demonstrate this advantage, by presenting an algorithm – that combines *quantum parallelism* with a property of quantum mechanics known as *interference* (Nielsen and Chuang, 2010). Deutsch's algorithm later evolved into what is known today as *Deutsch-Jozsa*'s algorithm, mainly used in introductory courses as an illustration of 'quantum advantage'. Even though considered a "toy algorithm", it supports the belief in quantum computing's superiority for solving certain classes of problems. This belief was crystallised most notably in 1997 by Peter Shor who demonstrated that two strikingly hard computational problems - prime factorisation of an integer and the so-called *discrete logarithm* - could be solved efficiently in a quantum computer Shor (1997). Since these two problems seem to have no efficient solution on a classical computer, the breakthrough fuelled interest among the scientific community. Additionally, Lov Grover's 1996 discovery of a quadratic speedup on the unstructured database search problem using a quantum computer further confirms the potential power of quantum computing Grover (1996). Unfortunately, while there may be other problems that quantum computers can solve more efficiently than classical ones, coming up with good quantum algorithms is still a hard task.

2.1.2 *Qubit*

In classical computation information is represented by *bits*. In quantum information, a bit can be encoded using a two-dimensional property of a subatomic object (usually, a photon or an electron, or an ion). This quantum two-dimensional system is called a *quantum* bit, or *qubit* for short. The bases of a qubit can be represented using *Dirac* '$|\rangle$' notation - $|0\rangle$ and $|1\rangle$. One of the most prominent aspects of a qubit is that it can be in a state other than $|0\rangle$ or $|1\rangle$; more specifically it can be in a *superposition*, encoded in the form of a linear combination of $|0\rangle$ and $|1\rangle$. A linear combination of states is written as a vector:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where α and β are complex numbers, called probability amplitudes. In the quantum setting, we are interested on those combinations that satisfy the equation $|\alpha|^2 + |\beta|^2 = 1$. Then, the probability of observing state $|0\rangle$ as the result of measuring the qubit $|\psi\rangle$ is $|\alpha|^2$ while the probability of observing $|1\rangle$ is $|\beta|^2$ (Nielsen and Chuang, 2010).

Simply put, the state of a qubit is a unit vector in the two-dimensional complex vector space $\mathbb{C}^2$. The states $|0\rangle$ and $|1\rangle$ are known as *computational basis states*, and form an orthonormal basis for this space. In matrix from they are formally defined as:

$$|0\rangle = [1,0]^T, \quad |1\rangle = [0,1]^T \tag{2}$$

For the particular cases where $\alpha = \beta = \frac{1}{\sqrt{2}}$ and $\alpha = \frac{1}{\sqrt{2}}$ and $\beta = -\frac{1}{\sqrt{2}}$, the corresponding states are commonly denoted by $|+\rangle$ and $|-\rangle$:

$$|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \tag{3}$$

$$|-\rangle = \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle \tag{4}$$

For these amplitude values, we have $|\alpha|^2 = |\beta|^2 = \frac{1}{2}$. This means that we have a 50% probability of observing either $|0\rangle$ or $|1\rangle$ when measuring either of the above states.

If global phases are ignored, the state of a qubit can be represented geometrically on a three-dimensional unit sphere - often called *Bloch sphere* - showed in Figure 1 by rewriting Equation 1 as:

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle \tag{5}$$

where θ and ϕ are respective polar and azimuthal angles.

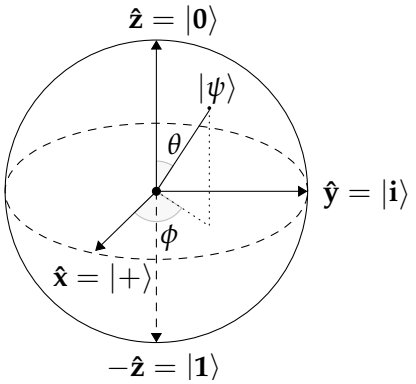


Figure 1: Bloch sphere representation of a qubit's state

By observing Figure 1, we see that a qubit can be in an infinite number of states. However, when a qubit is *measured* its state collapses from a superposition of $|0\rangle$ and $|1\rangle$ to a specific state corresponding to the measurement result, either $|0\rangle$ or $|1\rangle$ (*i.e.* one of the poles). For example, if we measure a qubit in the state $|+\rangle$ and obtain 0 as a result, the state collapses to $|0\rangle$.

2.1.3 *Quantum circuits - a model for quantum computing*

Although conceptually different, quantum computation is analogous to classical computation in the sense that both can be represented as circuits containing wires (where information passes) and gates (which compute the information). Akin to classical circuits and logical gates, information in a quantum circuit is manipulated via what are called *quantum gates*. To understand how such gates work is essential to comprehend and build different quantum algorithms, so we will briefly overview them next.
The most common notation for quantum gates is the following:

$$|\psi_{in}\rangle \text{ —} \boxed{U} \text{— } |\psi_{out}\rangle$$

where U can be any unitary gate, and the circuit's input and output are given by $|\psi_{in}\rangle$ and $|\psi_{out}\rangle$, respectively. Quantum circuits are represented as a sequence of gates, read from the left to the right, and each wire represents a single qubit. Unitary gates are used to represent unitary operators, which perform unitary transformations on qubits. An operator U is defined as *unitary* if $U^\dagger U = I$ (Nielsen and Chuang, 2010), where $U^\dagger$ denotes the Hermitian conjugate of U and I denotes the identity matrix. The importance of unitary operators lies in the fact that they preserve inner products between vectors, which is essential for the preservation of norms of states that are encoded as unit vectors (as given by Eq.1). Therefore, it is essential that gates preserve norms, and as a result, single quantum gates (*i.e.* gates that work on one qubit) are described as 2×2 unitary matrices. One of the most important single quantum gates is the *Hadamard* gate H represented by matrix 6:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{6}$$

When the *Hadamard* gate is applied to an element of the computational basis it creates a superposition:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) = |+\rangle \tag{7}$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) = |-\rangle \tag{8}$$

In circuit language:

Other essential single quantum gates are given by the *Pauli* matrices:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{9}$$

In particular, the X gate is usually referred to as *NOT* gate, because its application to a quantum state results in:

$$X(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle \tag{10}$$

The Z gate leaves $|0\rangle$ unchanged and flips the sign of $|1\rangle$ to give $-|1\rangle$. By using a linear combination of Pauli matrices a group of gates called *rotation gates*, generally defined as:

$$R_u(\theta) = e^{\frac{-i\theta u}{2}}, \quad u \in \{X, Y, Z\} \tag{11}$$

The application of one of these gates to a qubit results in a rotation of its state around one of *Bloch sphere's* (Figure 1) axis – $\hat{x}$ if the gate is $R_X(\theta)$, $\hat{y}$ if it is $R_Y(\theta)$ or $\hat{z}$ if it is $R_Z(\theta)$ – by an angle of θ:

$$R_X(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \; R_Y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \; R_Z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix} \tag{12}$$

Another gate worth mentioning is the *phase shift gate* $Ph(\theta)$, which shifts the global phase of a qubit by θ if its quantum state is $|1\rangle$. From this gate, we can obtain the S and T gates, which correspond to the phase shift gate with $\theta = \frac{\pi}{2}$ and $\theta = \frac{\pi}{4}$, respectively:

$$Ph(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} \tag{13}$$

2.1.4 *Multi-qubit systems*

When dealing with multiple qubits, the space of corresponding states arises from the tensor product ($\otimes$). Specifically, for n-qubits the state space of the system is $\mathbb{C}^{2^n}$. If we consider a multi-qubit system $|\psi\rangle$ with n qubits, each described by a state $|\psi_i\rangle$, then we note that the composite state is given by:

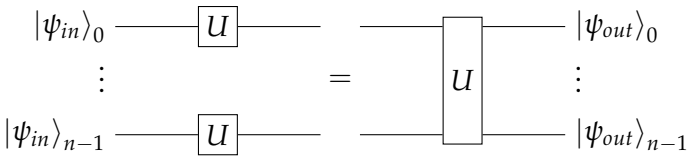
$$|\psi\rangle = |\psi_0\rangle \otimes |\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle = \bigotimes_{i=0}^{n} |\psi_i\rangle \tag{14}$$

For a more compact representation, $\bigotimes_{i=0}^{n} |\psi_i\rangle$ will be typically represented as:

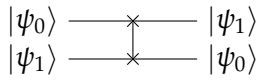
$$|\psi_0 \psi_1 \dots \psi_{n-1}\rangle \tag{15}$$

An example of a multi-qubit gate is the tensor product of n single-qubit gates U (for example, H or X):

$$U^{\otimes n} \bigotimes_{i=0}^{n} |\psi_i\rangle = \bigotimes_{i=0}^{n} U |\psi_i\rangle \,,\; U \in \{H, X, Y, Z, \dots\} \tag{16}$$



Swap gates, as their name suggests, swap the position of two qubits:

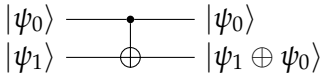


Another useful multi-qubit quantum gate is the *controlled-NOT* gate (*C-NOT* for short). This gate receives two input qubits, designated as *control* and *target* respectively. If the control qubit – marked with a dot – is set to $|1\rangle$, the target qubit – marked with a "$\oplus$" – is flipped – as if an X gate were applied to it. Otherwise if the control qubit is $|0\rangle$, the target qubit stays the same. Thus we obtain:

$$\begin{aligned} &\text{C-NOT}\,|00\rangle \mapsto |00\rangle\,, \quad \text{C-NOT}\,|01\rangle \mapsto |01\rangle \\ &\text{C-NOT}\,|10\rangle \mapsto |11\rangle\,, \quad \text{C-NOT}\,|11\rangle \mapsto |10\rangle \end{aligned} \tag{17}$$

The corresponding matrix and circuit are given by:

$$\text{C-NOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{18}$$



This operation is as a generalization of a classical *XOR* gate, since its action can be described as $|A, B\rangle \rightarrow |A, B \oplus A\rangle$, where $\oplus$ is addition modulo two, which is exactly what the *XOR* gate does Nielsen and Chuang (2010).

Suppose that instead of the X gate, which flips the qubit given as input, we now have an arbitrary single qubit gate U. A *controlled-U* operation 2 will follow the same logic as the *C-NOT* gate: there will be a control and a target qubit, and only if the control qubit is set to $|1\rangle$ the U gate will be applied to the target qubit; otherwise this qubit is left unchanged:

$$\text{C-U}\,|\psi_0, \psi_1\rangle \mapsto |\psi_0\rangle\, U^{\psi_0}\,|\psi_1\rangle \tag{19}$$

Note that since our control qubit $|\psi_0\rangle$ will be in a state $|0\rangle$ or $|1\rangle$, the notation U^{ψ_0} means that the gate U will either be applied to the target qubit $|\psi_1\rangle$, or act as an identity gate $\mathbb{I}$, which maps a state to itself: $\mathbb{I}\,|\psi\rangle \mapsto |\psi\rangle$.

Note that the U gate can be any unitary gate, meaning that it can also represent a *multi-qubit* gate.

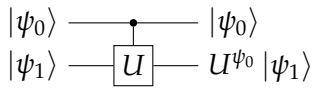
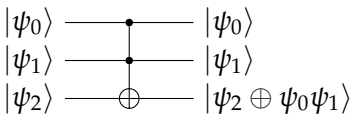


Figure 2: *Controlled-U* gate

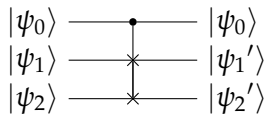
Another very useful multi-qubit gate is the *Toffoli* gate, which is basically a generalized *controlled-NOT* gate. It receives 3 qubits as input - two control qubits and a target one - and only flips the target qubit if both the control qubits are set to $|1\rangle$:



Note that if $|\psi_2\rangle$ is set to $|1\rangle$, its output state will correspond to the result of a *NAND* gate acting on the first two qubits – state $|0\rangle$ is obtained only when both $|\psi_0\rangle$ and $|\psi_1\rangle$ are set to $|1\rangle$:

$$\begin{aligned}
&\text{Toffoli}\,|000\rangle \mapsto |000\rangle\,,\ \text{Toffoli}\,|001\rangle \mapsto |001\rangle \\
&\text{Toffoli}\,|010\rangle \mapsto |010\rangle\,,\ \text{Toffoli}\,|100\rangle \mapsto |100\rangle \\
&\text{Toffoli}\,|110\rangle \mapsto |111\rangle\,,\ \text{Toffoli}\,|111\rangle \mapsto |110\rangle
\end{aligned} \tag{20}$$

A less common, but still useful, multi-qubit gate is the *Fredkin* gate, also known as "*CSWAP*", which acts as a controlled *swap* gate:



2.1.5 *Universal quantum gates*

In quantum computing, a set of gates is said to be *universal* if any unitary operation may be approximated to arbitrary accuracy by a quantum circuit involving only those gates. Nielsen and Chuang (2010) describes three different *universality constructions* for quantum computing, which culminate in a proof that any unitary operation can be approximated to arbitrary accuracy using the *Hadamard*, *Phase* ($Ph(\theta)$), *C-NOT* and T gates.

2.1.6 *An example of Quantum Advantage*

It is referred to as a *quantum advantage* when a quantum computer can solve a specific class of problems more efficiently than a classical computer. As detailed in the example below, one of the significant advantages of quantum computing is the ability to perform an extremely high number of computations in parallel. Technically, this is achieved using the concept of superposition. Superposition can be achieved by simply applying n *Hadamard* gates in parallel on n qubits. For n qubits initialized at state $|0\rangle$ ($|0\rangle^{\otimes n}$), we obtain the circuit in Figure 3:

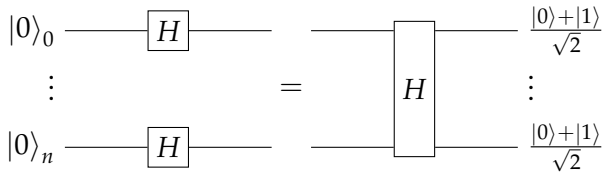


Figure 3: *Hadamard* transform on n qubits

The resulting state for this circuit is:

$$\frac{1}{\sqrt{2^n}} \sum_{x \in 2^n} |x\rangle. \tag{21}$$

Note that the state of n-qubits carries the information of 2^n complex numbers. Thus a quantum superposition state, can store 2^n inputs in n qubits simultaneously. This means that as the number of qubits grows *linearly*, the number of associated complex numbers grows *exponentially*. This is a reason why it is so hard to simulate quantum mechanical systems classically.

Suppose there is a function $f : \{0,1\} \to \{0,1\}$. It receives the state $|x,y\rangle$ as input, where x is a superposition: $|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Now, assume there's an appropriate quantum circuit, which will be called U_f, defined by a *black-box* (for the purpose of this example) that performs the transformation $|x,y\rangle \mapsto |x, y \oplus f(x)\rangle$ – similarly to a *C-NOT* gate (see 17). With a little thought, if $y = 0$, the final state of the second qubit will simply be the value $f(x)$. With this assumption, the circuit in Figure 4 will result in the state:

$$\frac{|0,f(0)\rangle + |1,f(1)\rangle}{\sqrt{2}} \tag{22}$$

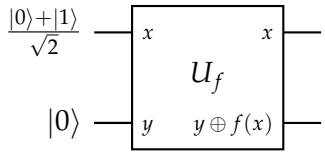


Figure 4: Quantum circuit for simultaneously evaluating $f(0)$ and $f(1)$

which contains information about both $f(0)$ and $f(1)$, as if f was evaluated for $|0\rangle$ and $|1\rangle$ simultaneously. This phenomenon is referred to as *quantum parallelism*, and is a fundamental feature of many quantum algorithms Nielsen and Chuang (2010).

Despite the fact that the function f appears to be evaluated only once, quantum parallelism allows for the evaluation of f for all potential inputs at once. This is not necessarily useful, since it is only possible to extract one value of f from a superposition state such as $\sum_x |x, f(x)\rangle$. Nevertheless, simple modifications to the quantum circuit in Figure 4 enable it to outperform classical ones. The resulting circuit is presented in Figure 5:

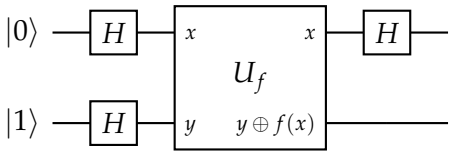


Figure 5: Quantum circuit respective to *Deutsch's algorithm*

The input state $|01\rangle$ is sent through two *Hadamard* gates in parallel, and the circuit U_f is applied to the resulting state. Then a final *Hadamard* gate is applied to the first qubit, which results in the state:

$$\begin{cases} \pm|0\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) = f(1) \\ \pm|1\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) \neq f(1) \end{cases} \tag{23}$$

It can be rewritten more concisely as

$$|\psi\rangle = \pm\,|f(0) \oplus f(1)\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right], \tag{24}$$

So if the first qubit is measured, the result $f(0) \oplus f(1)$ – which represents a global property of $f(x)$ – can be determined using only one evaluation of f, which, in the classical case, would require at least *two* evaluations. This example demonstrates a property of quantum mechanics known as *interference*. Intuitively, U_f introduces *interference* patterns which, after applying the *Hadamard* gate, will give away information about a **global property** of the function f, namely the value $f(0) \oplus f(1)$.
Deutsch's algorithm is a simplified version of the more general *Deutsch-Jozsa algorithm* [Nielsen and Chuang (2010)]. Although this algorithm is often referred to as a *toy algorithm* – since it has no known applications – it provides an **exponential** speedup, and it makes use of unique quantum mechanics' properties in order to efficiently manipulate information.

2.1.7 *Quantum Computing in our days*

Current generation quantum computers **are not** yet capable of replacing classical computers for commercially relevant applications. However, as quantum technology advances very rapidly, quantum applications are expected to become viable during the next decade (Finke, 2022).

The main obstacle in the path to such applications is the fact that the state of a qubit is extremely sensitive to disturbances from the outside environment, which raises a critical challenge in quantum computing technology. It is foreseen that while the community tries to overcome this obstacle, quantum computing will fall into two phases:

1. the *NISQ* phase: standing for *Noisy Intermediate Scale Quantum*, in which noise in quantum gates cannot be completely eliminated. This generally limits the size of quantum circuits that can be executed reliably and promotes the design of near-term, noise-resilient quantum algorithms. *Intermediate scale* refers to quantum computers ranging from 50 to around 100 qubits (Preskill, 2018).

2. *Fault Tolerant Quantum Computing* phase: through the application of quantum error correction algorithms, a threshold that limits the probability of error can be imposed, mitigating the effect of noise in the underlying hardware Nielsen and Chuang (2010).

2.2 QBRICKS FRAMEWORK

Quantum algorithms are usually described as a series of operations which, when composed with each other, transform an initial state into the desired state. This state is then usually measured to retrieve classical information. A quantum algorithm is thus described as a sequence of operations typically given in the form of a quantum circuit or a quantum programming language. It is also common to attach a specification to an algorithm, *i.e.* the corresponding *global memory-state* transformation typically described in logical form. A major challenge is then to verify that the circuit generated by the code written as an implementation of a given algorithm *indeed corresponds* to the intended algorithm, and that the transformations occurring along the circuit *indeed* result in the intended *global state* (Chareton et al., 2021).

2.2.1 *An overview of* QBRICKS

QBRICKS (Chareton et al., 2021) is a framework for semi-automated formal verification of quantum programs. Specifically it supports the description of quantum circuits using a circuit description language and semi-automatic proof support concerning program specifications. It thus lowers the amount of labor-intensive effort needed to create verified quantum programs.

In more detail, QBRICKS is equipped with:

- a *domain specific language* (QBRICKS-DSL) for building parameterised quantum circuits;
- a dedicated *logical specification language* (QBRICKS-SPEC);
- a flexible path-sum (Amy, 2019) *symbolic representation* for reasoning about quantum states - named *parameterised path-sum semantics* (*PPS*);
- a *Hoare-style logic*, called *Hybrid Quantum Hoare Logic (HQHL)*, for deductive verification of quantum programs.

Overall the framework can be boiled down to two key aspects:

QUANTUM CIRCUIT REPRESENTATION is achieved through the *domain specific language* (QBRICKS-DSL) - which makes use of data constructors which represent elementary gates, sequential and parallel composition, and ancilla creation Chareton et al. (2021). One particular aspect of QBRICKS-DSL is that *measurement* is out of its scope. Nonetheless it is possible to *reason* about probabilistic outputs of circuits, as if the result of a circuit was measured Chareton et al. (2021).

PROGRAM SPECIFICATION AND VERIFICATION is accomplished by using *PPS* symbolic representation as a specification mechanism for quantum programs. Figure 6 illustrates how a program implemented using QBRICKS framework is processed. A program in QBRICKS will consist of two parts: a circuit description, accomplished using QBRICKS-DSL, and a formal specification, using QBRICKS-SPEC and *PPS* representation. Along with QBRICKS-SPEC specification language and *PPS*, the so-called *Hybrid Quantum Hoare Logic (HQHL)* engine produces proof-obligations which are then validated (mainly by SMT solvers) Chareton et al. (2021).

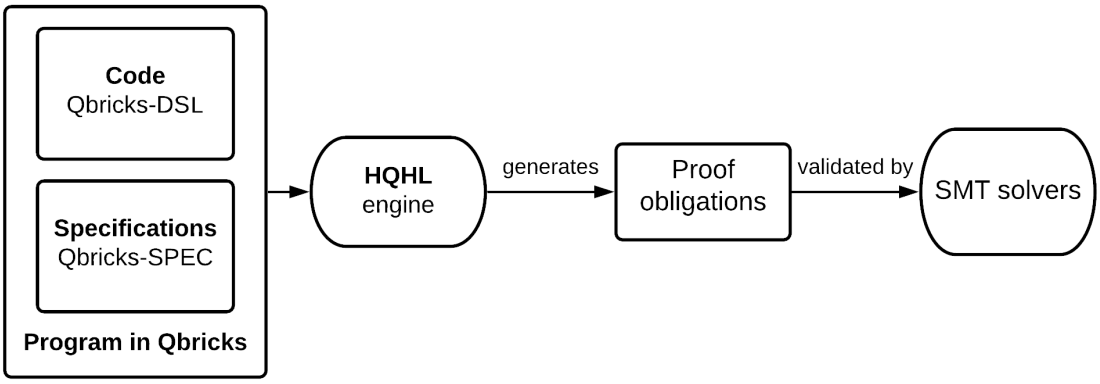


Figure 6: Overview of QBRICKS verification process

2.2.2 *Circuit representation*

Quantum circuits in QBRICKS are represented using QBRICKS-DSL, a *first-order*, *functional* language. Unlike *e.g. Quipper* or *Qwire*, a quantum circuit in QBRICKS is not a function acting on qubits: it is a simple, static object. A circuit is a compositional structure consisting of gates and circuit combinators, such as sequential composition, parallel composition, control and inversion Chareton et al. (2021).

QBRICKS is embedded in the Why3 (Filliâtre and Paskevich, 2013) deductive verification framework. Programs are written in ML language (Leroy et al., 2022) with an (opaque) datatype `circ` as the medium to build and manipulate circuits. On top of `circ`, the type system of QBRICKS-DSL features the integer (`int`) and arrow types. This type system is not exhaustive and is meant to be extended with usual constructs such as Booleans, pairs, lists, and other user-defined inductive datatypes: its embedding into `WhyML` makes it easy to use such types. The syntax for QBRICKS-DSL is expressed in Figure 7.

The core language constructs for QBRICKS-DSL are divided in two main parts:

$$
\begin{array}{rrcl}
\textit{Types} & A, B & ::= & \texttt{circ} \mid \texttt{int} \mid A \to B \mid \ldots \\
\textit{Circuit Terms} & C & ::= & Ph(\theta) \mid H \mid \texttt{CNOT} \mid R_z(\theta) \mid \texttt{parallel}(M, N) \mid \texttt{sequence}(M, N) \mid \\
 & & & \texttt{invert}(M) \mid \texttt{control}(M) \mid \texttt{ancilla}(M) \mid \texttt{size}(M) \\
\textit{Terms} & M, N & ::= & C \mid n \mid \theta \mid x \mid \lambda x.M \mid M\,N \mid \texttt{let rec}\ f\ x = M\ \texttt{in}\ N
\end{array}
$$

Figure 7: Syntax for Qbricks-Dsl

- Circuit Terms, which consist in basic elementary gates, high-level combinators and a *size* construct:
  - **Basic elementary gates:** H for the Hadamard gate, $R_z(\theta)$ for the rotation gate around z-axis, $Ph(\theta)$ for the global phase shift gate on one qubit and *CNOT* for the controlled *NOT* gate. The chosen gates represent a *universal set of gates*, meaning that any quantum unitary matrix can be approached arbitrarily close via the sequence or parallel composition of these gates. Other, more convenient gates, can be defined as macros on top of these. If one aims at using Qbricks inside a verification tool-chain, these macros can for instance be the gates of the targeted architecture.
  - **High-level circuit combinators:** the purpose of these constructors is to allow different circuits to be arranged together: `parallel` for parallel composition of circuits, `sequence` for sequential composition of circuits with the same `size`, `invert` for inverting circuits, `control` for controlling circuits, and finally `ancilla`(M) for adding an auxiliary qubit initialized to $|0\rangle$ as the last wire of M which is discarded afterwards. Provided that their input is of type `circ`, the output of all combinators is `circ`.
  - ***Size* construct:** `size`(M) returns the number of wires of M. Provided that its input is of type `circ`, the output is `int`.
- Regular ML-like constructs:
  - Integer constants $(n, k, \ldots)$, term variables $(x, f, \ldots)$, lambda terms $(\lambda x.M)$, application $(M\,N)$ and *let-rec* construction to permit recursion.

2.2.3 *Specification and verification*

Specification and logical reasoning in Qbricks is supported by `Qbricks-Spec` specification language – aimed at expressing and proving specification properties of quantum programs (written via Qbricks-DSL) – along with a *Hybrid Quantum Hoare Logic (HQHL)* which follows Hoare's *pre* and *post* condition logical sequents (Hoare, 1969).

In order to interpret circuit description functions, Qbricks uses parameterised path-sums (*PPS*). *PPS* is an extension of *path-sums* (Amy, 2019), a way of specifying quantum operations and verifying the equivalence between quantum circuits.
In Qbricks, a path-sum P is an object with **four** parameters:

1. The *width* of the target circuit `pps_width`(P), with type `int`
2. The *range*, meaning that the output sum of kets has a term for each bit vector $\vec{y}$ of length `pps_range`(P), with type `int`
3. Function `pps_angle`(p) that for any input bit vector $\vec{x}$ of length `pps_width`(P) (standing for a basis ket input to the target circuit) and for any index bit vector $\vec{y}$ of length `pps_range`(P), defines a *real scalar* (standing for a basis ket output to the target circuit)
4. Function `pps_ket`(P), that for any input bit vector $\vec{x}$ of length `pps_width`(P) and for any index bit vector $\vec{y}$ of length `pps_range`(P), defines a *bit vector* of length `pps_width`(P) (standing for a basis ket output to the target circuit)

Table 1: PPS accessors and types

Identifier	Type	Id abbreviation
`pps_width`	`int`	p_w
`pps_range`	`int`	p_r
`pps_angle`	`bitvector` $\rightarrow$ `bitvector` $\rightarrow$ `real`	p_a
`pps_ket`	`bitvector` $\rightarrow$ `bitvector` $\rightarrow$ `bitvector`	p_k

Then for any bit vector $\vec{x}$ of size $p_w(P)$, the expression

$$Ps(h, |\vec{x}\rangle) = \frac{1}{\sqrt{2^{p_r(P)}}} \sum_{\vec{y} \in \mathrm{BV}_{p_r(P)}} e^{2 \cdot \pi i . p_a(P)(\vec{x},\vec{y})} \, |p_k(P)(\vec{x},\vec{y})\rangle_{p_w(P)} \tag{25}$$

combines these different elements together to define a linear application for quantum state vectors.

In Qbricks-Spec, a generalization of *path-sums* is given by parameterised path-sums (`pps`). A `pps` will act as a function which takes a set of parameters as input and returns a path-sum as output. Thus for each possible value of its parameters, this function outputs a *family* of *path-sums*, that describes the transformations applied in a family of quantum circuits.

This makes the `pps` representation well-fitted for the specification of parameterised quantum algorithms, allowing for the compositional combination of accessors (Table 1) along with circuit constructors from Qbricks-Dsl. As a result, it enables reasoning about parameterised quantum circuits and their semantics without requiring the manipulation of sum terms or other higher-order objects. With the aid of this tool, the automatic generation of proof obligations for Qbricks specifications only yields first-order formulas, allowing for a high degree of automation when given as input to SMT-solvers (Chareton et al., 2021).

Since both Qbricks-Dsl and Qbricks-Spec ultimately express families of quantum circuits, it is possible to access Qbricks-Dsl object terms with Qbricks-Spec, by making use of the `circ_to_pps`(C) function, where C represents a quantum circuit family in Qbricks-Dsl. Since this function outputs a `pps`, it is possible to reason about circuit constructors' properties by using the accessors listed in Table 2.

Table 2: Function `circ_to_pps` and accessors

Accessor
`pps_width(circ_to_pps(`C`))`
`pps_range(circ_to_pps(`C`))`
`pps_angle(circ_to_pps(`C`))`
`pps_ket(circ_to_pps(`C`))`

Combined with Qbricks-Dsl, Qbricks-Spec makes it possible to express properties regarding Qbricks-Dsl circuit constructors, by making use of *HQHL* expressions in the form of *pre* and *post* conditions for arbitrary parameterised circuits. Hoare logical expressions have the form: $\{\Psi\}\ e\ \{\Phi\}$. This notation states that whenever Ψ is satisfied, running e ensures that Φ is satisfied. By including `pps` specifications in a program, one can specify *pre*-conditions on function inputs and *post*-conditions ensuring properties on function outputs. These *pre* and *post* conditions can be chained along the composition of functions in order to generate *proof-obligations* that are sent to the proof engine.

The formulas expressing semantical constraints used in Qbricks-Spec are of the form:

$$\forall \vec{x} : \vec{A} \cdot P(\vec{x}) \longrightarrow (M : Exp_1 \mapsto Exp_2) \tag{26}$$

meaning "For all typed variables $x_1 : A_1, \ldots, x_n : A_n$ of Qbricks-Dsl, provided that the property $P(\vec{x})$ is satisfied, then the circuit corresponding to the term M (of type `circ`) maps the algebraic expression Exp_1 to Exp_2". In particular, M is an open term of Qbricks-Dsl (of type `circ`) with free variables contained in $\vec{x}$; $P(\vec{x})$ is a logical property that should be satisfied by any substitution of $\vec{x}$; finally, Exp_1 and Exp_2 are algebraic expressions parameterised by $\vec{x}$ (Chareton et al., 2021).

For example, programs that represent parallel and sequential composition and their specifications are given in Tables 3 and 4. M_1 and M_2 represent two arbitrary `circ` terms. Expressions "pre" and "post" refer to *pre-* and *post*-conditions for these compositions. The variable `result` contains the `circ` term which results from the application of each combinator.

1. Concerning the composition in Table 3, it is guaranteed that the `width` of `result` is equal to the sum of the widths of M_1 and M_2, given by n_1 and n_2, respectively. Additionally, it states that the application of `result` to the tensor product of input vectors $|\vec{x}\rangle$ and $|\vec{y}\rangle$ results in the tensor product of their respective PPS (this notation is the same as in Subsection 2.1.4).

2. Concerning the composition in Table 4, it must be true that the `width` of M_1, M_2 and `result` is the same. Finally, it is also guaranteed that PPS_3 – corresponding to the *path-sum* which results from applying both M_1 and M_2 (sequentially) to input state $|\vec{x}\rangle$ – is obtained by applying `result` to the initial PPS_1.

Table 3: Parallel composition with *pre* and *post* conditions

`result = parallel`(M_1, M_2)
pre:
$\texttt{width}(M_1) = n_1$
$\texttt{width}(M_2) = n_2$
$\forall \vec{x} \cdot (M_1 \cdot \|\vec{x}\rangle \mapsto \text{PPS}_1(\vec{x}))$
$\forall \vec{y} \cdot (M_2 \cdot \|\vec{y}\rangle \mapsto \text{PPS}_2(\vec{y}))$
post:
$\texttt{width}(\textit{result}) = n_1 + n_2$
$\forall \vec{x} \cdot \forall \vec{y} \cdot (\texttt{result} \cdot \|\vec{x}\rangle \otimes \|\vec{y}\rangle \mapsto \text{PPS}_1(\vec{x}) \otimes \text{PPS}_2(\vec{y}))$

Table 4: Sequential composition with *pre* and *post* conditions

`result = sequence`(M_1, M_2)
pre:
$\texttt{width}(M_1) = n$
$\texttt{width}(M_2) = n$
$\forall \vec{x} \cdot (M_1 \cdot \text{PPS}_1(\vec{x}) \mapsto \text{PPS}_2(\vec{x}))$
$\forall \vec{x} \cdot (M_2 \cdot \text{PPS}_2(\vec{x}) \mapsto \text{PPS}_3(\vec{x}))$
post:
$\texttt{width}(\textit{result}) = n$
$\forall \vec{x} \cdot (\texttt{result} \cdot \text{PPS}_1(\vec{x}) \mapsto \text{PPS}_3(\vec{x}))$

2.2.4 *Probabilistic reasoning*

As mentioned in Subsection 2.2.1, Qbricks-Dsl lacks a construct for *measuring* quantum registers. However, Qbricks-Spec offers *reasoning* tools that can in some way overcome this limitation. The probability of obtaining a result by a measurement is correlated with the amplitudes of the corresponding ket-basis vectors in the quantum state in memory. In particular, function `proba_measure`:

$$\texttt{proba\_measure}(C, |v\rangle_n, j) = |(Ps(\texttt{circ\_to\_pps}, |v\rangle_n, j))|^2 \tag{27}$$

receives as input a circuit C, a quantum data register $|v\rangle_n$ and an index $j \in [0, 2^n[$; and it outputs the **probability of one measuring** j in the quantum register that results from the application of circuit C to $|v\rangle$.

Qbricks' framework has been tested against different quantum algorithms such as Quantum Fourier Transform (*QFT*), Grover's search algorithm, Quantum Phase Estimation (*QPE*), Shor order-finding, among others (Chareton et al., 2021).

2.3 LANGUAGE PROCESSING

A language is a set of valid sentences, a sentence is made up of phrases, and a phrase is made up of vocabulary symbols, called *Terminal* symbols. Terminal symbols shall appear in a *sequence* and comply to a *correct order* to form a valid language *sentence*. In order to implement a processor able to cope with language sentences, an application that reads sentences and reacts appropriately to the phrases and input symbols must be built. The main contribution of this M.Sc. project is to convert "sentences" from the language that we implemented to another language (QBRICKS).

2.3.1 *Fundamentals*

Programs that recognize the structure of a language's sentences are called *parsers*. ANTLR tool (Parr, 2013) translates grammars to parsers that look remarkably similar to what an experienced programmer might build by hand. This structural analysis usually boils down to two tasks:

LEXICAL ANALYSIS the process of using a *lexer* for grouping characters into words or symbols (*tokens*), using *Regular Expressions* to formally specify each token;

PARSING the actual *parser* that feeds off of these *tokens* to recognize the sentence structure, using a grammar as formal specification of the valid sentences.

The diagram shown in Figure 8 illustrates the basic *data flow* to process the syntactic structure of a sentence. Input text is converted into *tokens* by the *lexer*. These *tokens* are then analysed by the *parser* which generates a structure (usually a *parse-tree*), that represents how input text matches the grammar.

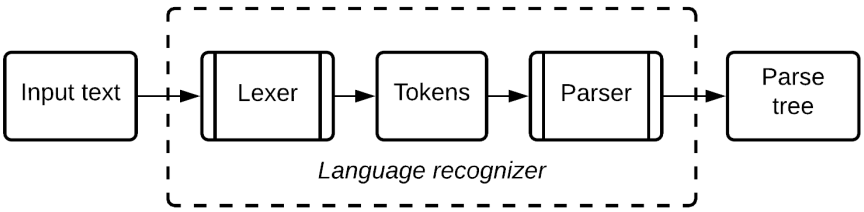


Figure 8: *Data flow* of a language recognizer

The most common formalism for describing languages are *context-free grammars (CFG)*, which are formally defined as a 4-tuple:

$$CFG = \langle T, N, S, P \rangle \tag{28}$$

where

T is the set of all **terminal symbols** of the language

N is the set of all **non-terminal symbols** of the grammar

$S \in N$ is the grammar's **start symbol** or **axiom**

P is the set of **productions** of the grammar

Each production $p \in P$ has the form $p : N \to (N \cup T)^*$, where '$*$' comes from the *EBNF (Extended Backus-Naur Form)* notation, and indicates *zero or more* repetitions of a part.

2.3.2 *Designing a language with ANTLR*

ANTLR (Parr, 2013) is a powerful parser generator for recognising and processing structured text written according to the grammar of a specific language. It is widely used in academia and industry to build all sorts of languages, tools, and frameworks. From a *formally* defined *grammar*, ANTLR generates a *parser* for that language that can automatically build *parse trees*, also known as *concrete syntax trees* - which represent how input text matches the grammar - while also generating tree walkers that can be used to visit the nodes of those *trees* to execute application-specific code (Parr, 2013). Parse-tree walkers can be in the form of *listener* and *visitor* pattern implementations, thus deemphasizing embedding actions (code) in the grammar. This decouples grammars from application code, nicely encapsulating an application inside the visitors, instead of fracturing it and dispersing the pieces across a grammar (Parr, 2013).
Given its flexibility and simplicity, ANTLR makes it much easier to build language-based applications, in particular it provides a great environment for developing new programming languages.

2.3.3 *The Abstract Syntax Tree structure*

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of text written in a formal language. Each node of the tree typically denotes a construct occurring in the text. Nodes contain information about different language block-builders (or constructors) that appear throughout a program. A node can also contain *children-nodes*, thus the denomination '*tree*'.

The syntax is "abstract" in the sense that it only represents the structural or content-related aspects rather than every detail found in the actual syntax. This distinguishes abstract syntax trees from *concrete syntax trees*, or *parse trees*, mentioned in the previous Subsection 2.3.2, which contain *terminal* symbols typically with no semantic relevance, such as brackets or parentheses, commas or semi-colons to disambiguate expressions. Once a *parse tree* is built (from a source file), an AST can be generated by *visiting* this *parse tree* and extracting relevant information about the program. The generated AST can then be edited and adapted, in order to represent different *abstract structures* which will ultimately represent a program's information.

Consider the example in Listing 2.1.

```
while (a > 0) :
   if (b == 1) :
      b = a * 3
```

Listing 2.1: Code example

After processing the lines of code in Listing 2.1, the *parse-tree* in Figure 9 is generated by a parser. A corresponding *Abstract Syntax Tree* can be seen in Figure 10. It is noticeable that some of the nodes in Figure 9 have been removed. This is due to the subsequent processing that occurs after generating the *parse-tree*. When the *AST* is generated, variables `a` and `b` are identified and locally stored, *integer* numbers are also identified and labelled, and *expressions* are *unfolded*. It is also possible to generate intermediary *ASTs*, enabling a more flexible and modular framework. *AST* generation is a vital tool for compiler design, since it allows for a convenient representation of programs, making subsequent processing and contextual analysis an easier and more intuitive task.

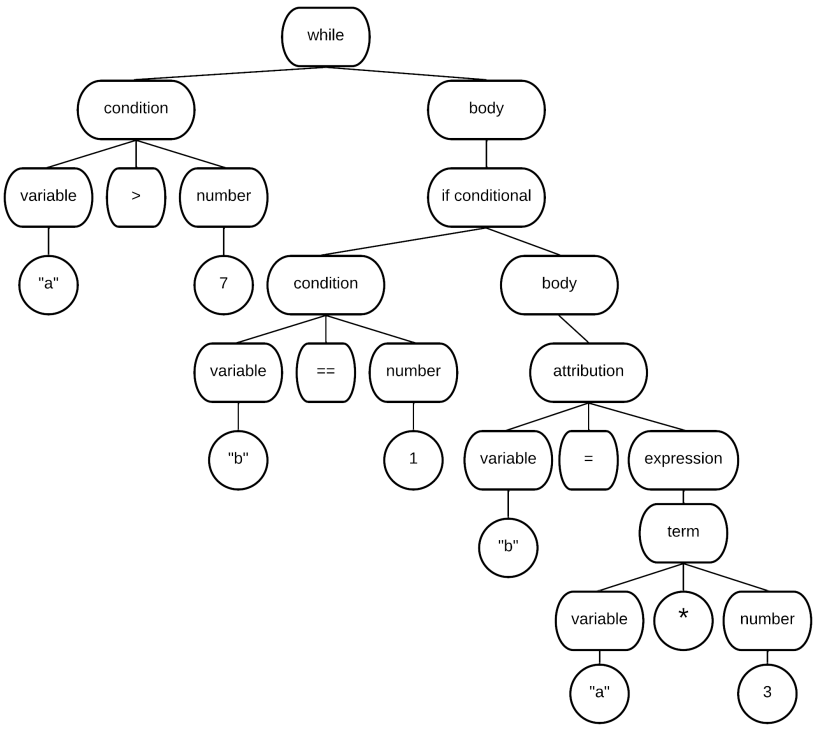


Figure 9: *Parse-tree* generated from code example

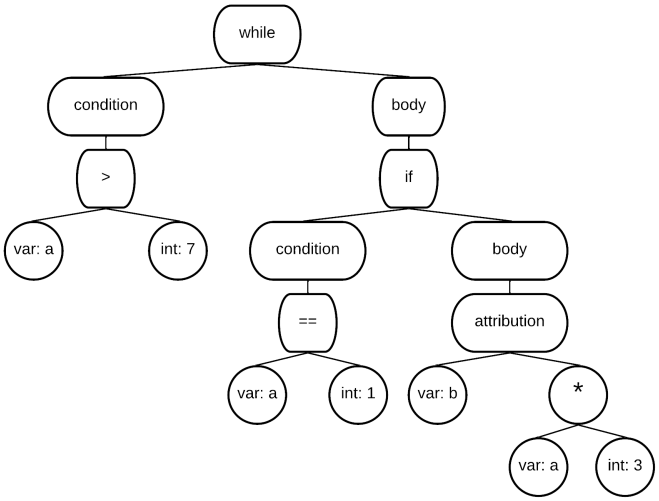


Figure 10: *AST* generated from *Parse-tree* in Figure 9

3

IQBRICKS LANGUAGE

This chapter introduces the iQbricks language. Its main features are presented and illustrated with different examples. iQbricks is also compared to other quantum programming languages using two different quantum circuit implementations. We then demonstrate how programs can be *annotated* with *specifications* in Qbricks-Spec. Finally, the language's *syntax analyser* is thoroughly explained along with the corresponding *grammar*.

The main goal of iQbricks language is to provide a set of symbols (vocabulary) and of *program constructors* that facilitate the description of quantum circuits. Thus throughout the design of this language, a number of features were implemented taking in account features from other *quantum circuit description oriented* languages, such as Qiskit (Cross, 2018), and Silq (Bichsel et al., 2020), aiming for an intuitive and easy-to-use quantum programming language.

3.1 MOTIVATIONAL EXAMPLES

This section introduces the main features of iQbricks and provides intuitions on how to write programs in this language. iQbricks is then compared against Qbricks, Qiskit and Silq via different quantum circuit implementations.

3.1.1 *Main features*

iQbricks circuit constructors are displayed in Figure 11. All the available gates were presented in Section 2.1. The goal for defining this set of gates *was not only* to represent a *pseudo-universal* set, but also to simplify the writing of quantum circuits: a *rich* set of circuit constructors wards off the need for complex gate compositions. Note that the *function application* gate, denoted by f, corresponds to a circuit defined by a *function* of type $B^{\otimes n} \to B^{\otimes n}$ – where B corresponds to a *Boolean* which can either be 0 or 1. iQbricks circuit constructors are expressed in Figure 11.

$$
\begin{aligned}
\textit{Function application gates} &: f \mid \textit{reverse} \\
\textit{Single-qubit gates} &: H \mid X \mid Y \mid Z \mid R_x(\theta) \mid R_y(\theta) \mid R_z(\theta) \mid Ph(\theta) \mid S \mid T \\
\textit{Multi-qubit gates} &: \textit{SWAP} \mid \textit{C-NOT} \mid \textit{Toffoli} \mid \textit{Fredkin} \mid \textit{Controlled-U}
\end{aligned}
$$

Figure 11: iQbricks circuit constructors

Single-qubit gates can be applied to quantum registers in one of three ways:

1. to a *specific index* of a register, *i.e. a single qubit*;
2. to a *whole register*;
3. to a specific *interval* of a register.

The different ways that one can apply gates avoids the need to use *for-loops* in order to *iterate* through quantum registers. Nevertheless, there are still cases where *for-loops* are useful for iterating through quantum registers. See the examples in Listings 3.1 and 3.2.

```
circ qr ->
      H(qr)
      X(qr)
```

Listing 3.1: Using the *global application* of gates to to iterate through a quantum register

```
circ qr ->
      for i in qr {
         H(qr[i])
         for a in qr[i:]{
            X(qr[a])
         }
      }
```

Listing 3.2: Using a *for-loop* to iterate through a quantum register

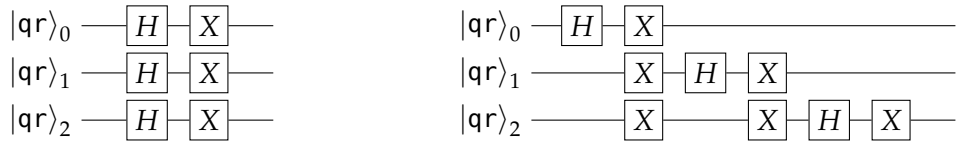
Let us analyse both examples thoroughly:

1. In the first example gates H and X are applied globally to register `qr`, of size n. This means n H gates will first be applied (in parallel), and then sequenced with n X gates (in parallel).
2. In the second example, a *nested-for* will allow the representation of a more complex circuit, by iterating through different indexes of register `qr` at each iteration[1].

For an input register of size $n = 3$, the corresponding circuits for the examples above are represented in Figures 12a and 12b.

The *controlled-U* gate (Figure 13) is a *customizable* control gate, in which

- *multiple* register indexes can be used as controls simultaneously;
- U can represent *any* circuit in iQbricks.



(a) *Global application* of gates H and X **(b)** Applying gates H and X inside a *for-loop*

Figure 12: Resulting circuits for *globally applying* gates and using *for-cycles*

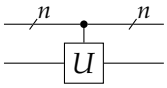


Figure 13: *Controlled-U* gate constructor

The use of the *controlled-U* gate allows for controlled operations with higher degrees of complexity to be represented with little effort. Suppose we wish to represent the circuit in Figure 14:

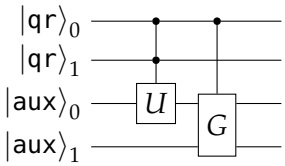


Figure 14: Example concerning *controlled* gates

Listing 3.3 exhibits a program fragment that represents the two *controlled*-operations.

```
circ qr, aux ->
     with control qr (U(aux[0]))
     with control qr[0] (G(aux))
```

Listing 3.3: Using *arbitrary control* operator to apply controlled gates

1. First, gate U is applied to the first index of register `aux` – `U(aux[0])` – using `qr` (globally) as the control.

2. Then gate G is applied (globally) to `aux` – `G(aux)` – using the first index of register `qr` – `qr[0]` – as the control.

Conjugate based circuit operations are also included in the language's constructors. A *conjugated* operation occurs when an operator U is applied before a circuit and its conjugate based operation $U^\dagger$ is applied after the circuit. In iQbricks, any gate U can be used to perform this conjugate based operation.
Consider a function f, which represents a circuit, and suppose we wish to use it as a *conjugate-based* operator. Reusing the circuit in Figure 14 and Listing 3.3, the code in Listing 3.4 would represent the circuit in Figure 15.

1 the notation `qr[i:]` in the inner *for-loop* states that the variable `a` will take values from the interval $[i, i+1, \ldots, n]$, where n is the size of register `qr`

```
circ qr, aux ->
   with conjugated (f(qr,aux)) {
      with control qr (U(aux[0]))
      with control qr[0] (G(aux))
   }
```

Listing 3.4: Using *conjugated basis* operator

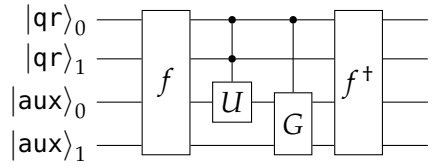


Figure 15: *Conjugated* gates circuit example

3.1.2 *Uniform superposition using Hadamard gates in parallel*

Consider a quantum circuit that creates a *uniform superposition* – *i.e.* it applies n *Hadamard* gates in parallel to $|0\rangle^{\otimes n}$ – as illustrated in Subsection 2.1.6. This is a very common operation in quantum algorithmics. The four different code examples in Listings 3.5, 3.6, 3.7 and 3.8 each implement a circuit which represents a uniform superposition in different languages: Silq (Bichsel et al., 2020), Qiskit (Cross, 2018), Qbricks and iQbricks.

```
x := 0:uint[n];
for i in [0..n) {
      x[i] := H(x[i]);
   }
```

Listing 3.5: Creating a uniform superposition in Silq

```
qc = QuantumCircuit(n)
for qubit in range(n):
   qc.h(qubit)
```

Listing 3.6: Creating a uniform superposition in Qiskit

```
let circuit = ref (m_skip n) in
for i = 0 to (n - 1) do
   circuit := !circuit -- (place hadamard i n);
done;
```

Listing 3.7: Creating a uniform superposition in Qbricks

```
circ qr[n] -> H(qr)
```

Listing 3.8: Creating a uniform superposition in iQbricks (without specification)

In the first three examples parallel application of the *Hadamard* gate is achieved through the use of *for-cycles*, where at each iteration an *Hadamard* gate is applied to a different index of the register. In the last example the *Hadamard* gate is **globally** applied to the register `qr`, which simplifies the writing and readability of the program.

3.1.3 *QFT algorithm*

The *Quantum Fourier Transform (QFT)* algorithm is the quantum implementation of the discrete *Fourier Transform* over the amplitudes of a wavefunction (Nielsen and Chuang, 2010). It is a component of several quantum algorithms, most notably *Shor*'s factoring algorithm and the *Quantum Phase Estimation (QPE)* algorithm/routine. The circuit that implements *QFT* is illustrated in Figure 16.

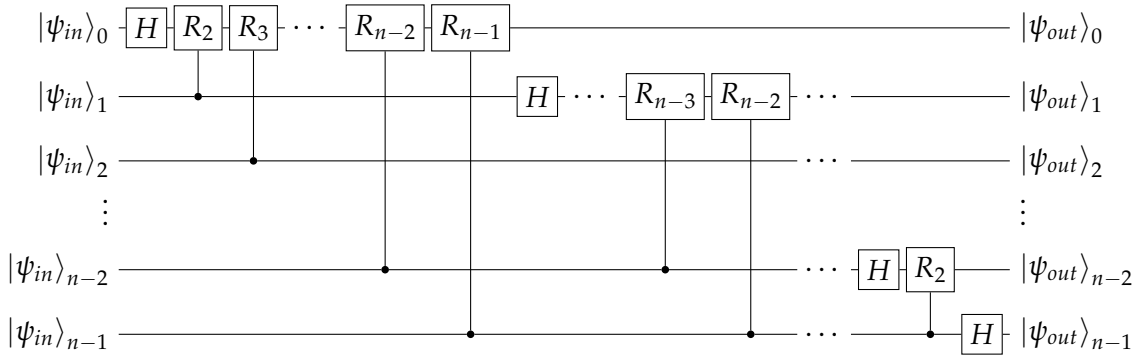


Figure 16: *QFT* circuit implementation

R_k is a rotation gate defined by the matrix:

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix} \tag{29}$$

The code in Listings 3.9, 3.10 and 3.12 corresponds to the *QFT* implementation using the languages Silq, Qiskit, Qbricks and iQbricks.

```
def qft [n:!N] (x: int[n]) mfree: int[n] {
   for k in [0..n) {
      x[k] := H(x[k]);
      for l in [k+1..n) {
         if x[l] && x[k] {
            phase(2*pi*2^(k-l-1));
         }
      }
   }
   return x;
}
```

Listing 3.9: *QFT* algorithm in Silq

Using Silq, a *nested-for* and a conditional clause are used along with *Hadamard* and *phase* rotation gates. Thus, the implementation uses 3 *control-flow* instructions and involves 2 quantum gates.

```
def qft(circuit, n):
   if n == 0:
```

```
      return circuit
   n -= 1
   circuit.h(n)
   for qubit in range(n):
      circuit.cp(pi/2**(n-qubit), qubit, n)
   qft(circuit, n)
```

Listing 3.10: *QFT* algorithm in Qiskit

Using Qiskit, the function `qft` is defined recursively, and contains one *for-loop*. The conditional clause checks the input size 'n' and decrements it if it is higher than `0`. This example has 2 *control-flow* operations, 1 recursive function-call and involves 2 quantum gates.

```
let qft (n:int) : circuit
=
begin
let q = ref 0
in let c = ref (m_skip n)
  in while (!q < n) do
      begin
         let i = ref (!q+1)
         in let cl = ref (m_skip n)
           in while (!i < n) do
               cl := !cl -- (crz !i (!q) (!i - !q+1) n );
               i := !i +1
           done;
         cl:= place_hadamard (!q) n -- !cl;
         c:= !c -- !cl;
         q := !q+1
      end
   done;
return (!c)
end
```

Listing 3.11: *QFT* algorithm in Qbricks (without specification)

Using Qbricks, a *nested-while* is used along with instructions for sequencing *Hadamard* and *controlled-Z* rotation gates to the function's circuit. Thus, the implementation uses 2 *control-flow* instructions and involves 2 quantum gates.

```
|| qft || (qreg qr)
   circ qr ->
      for q in qr {
         H(qr[q])
         for i in qr[q+1..-1] {
            with control qr[i+1] (RZ(i-q, qr[q]))
         }
      }
```

Listing 3.12: *QFT* algorithm in iQbricks (without specification)

In iQbricks, we have a *nested-for*, similarly to the Silq example in Listing 3.9. The circuit gates used in Listing 3.12 are: *Hadamard* gate and *controlled-U* gate with $U = R_z(\theta)$. Therefore, this implementation has 2 *control-flow* instructions and involves 2 quantum gates.

Concerning the different implementations of the *QFT* algorithm, iQbricks appears to be the most *readable*, and the one with less instructions.

3.2 Specification

Programs in iQbricks can be *annotated* with ***assertions***, *loop-invariants* and ***pre-*** and ***post-conditions***. The specification language used in the annotations is the aforementioned Qbricks-Spec (see Subsection 2.2.3 and Chareton et al. (2020)).
In a program, *main* and *auxiliary* functions each contain ***pre-*** and ***post-conditions*** for reasoning about input parameters and output results, respectively. Each instruction can be annotated with **assertions** after being applied. *For-cycles* are annotated with *loop-**invariants***, which is a condition that is necessarily true immediately before and immediately after each iteration of a loop.

The example in Listing 3.13 illustrates a general definition of an annotated program in iQbricks.

```
|| main || (qreg ancilla)
pre { ancilla > 0 }
   circ qr[4], ancilla -> # optional size declaration
      # Circuit description + annotations
      for i in qr {
         invariant {range circ = i}
         H(qr[i])
         assert {width circ = qr + ancilla}
      }
pos { {width result = qr + ancilla},
      {range result = qr}
   }
```

Listing 3.13: Example of a iQbricks program with specification

Firstly, a *pre-condition* for reasoning about function `main`'s inputs (`qreg ancilla`, in this case) is specified right after the declaration of the function's name and parameters[2]. Then at the beginning of the *for-loop*, a *loop-invariant* is specified – in this case, referring to the values that iterator `i` takes. By using the keyword '`circ`', we can reason about a circuit's properties. In this example, the operator '`range`' is used, which refers to the amount of created *superpositions* in a circuit. Since we apply an *Hadamard* gate at each iteration, this invariant is true throughout the loop. After applying the *Hadamard* gate, an *assertion* – referring to the size of the circuit (`width circuit`) – is used. We now refer to the `width` of `circ`, stating that it is equal to the sum of the input registers – `qr` and `ancilla`. Finally, at the end of the function, the *post-condition* is placed for reasoning about the obtained results. The keyword '`result`' is used for referring to the resulting circuit specified by the function.

2 When referring to quantum register's identifiers (in this case `ancilla`) *inside* a specification, the identifiers represent the register's **size**

Operators 'width' and 'range' are used again: the resulting circuit's *width* and *number of created superpositions* correspond to 'qr + ancilla' and 'qr', respectively.

3.3 SYNTAX ANALYSER

iQbricks is an "imperative style" language, in the sense that its programs consist of a sequence of statements that change a global state. Instructions enable the programmer to **implement** quantum circuits. The instructions available in iQbricks are the following: control-flow operations (through the use of conditional statements and 'for' loops) and basic quantum gates.
In order to formally define the syntactic rules of iQbricks, it is necessary to create a *grammar* to formally specify the set of valid sentences in this language. The tool Antlr translates this grammar to an actual parser. Then from an input file containing a program the latter builds a *parse tree* that records how the structure of this program was recognized. This was illustrated in Section 2.3. iQbricks' grammar, written in Antlr notation, is described concept by concept in the following subsections.

3.3.1 *Program definition*

A program is declared by optional file imports, a `main` function and, optionally, auxiliary functions (Listing 3.14). Functions from different files can specifically be imported at the very beginning of a program. The `main` function will represent the program's quantum circuit. Each additional auxiliary function will define a separate *quantum circuit*. These auxiliary circuits can then be used (in the `main` function) through function calls, in order to compose a more complex one.

```
program :
   (imports)? main (aux)*
   ;
```

Listing 3.14: Program definition

3.3.2 *Imports, main and auxiliary functions*

Each file to be imported to a program is identified by the *token* `IMPORT` (`IMPORT="import"`), followed by its name (`ID=[a-zA-Z][a-zA-Z_0-9]*`), which means that every `ID` must start with a letter and may contain any combination of digits and letters, with the exception of the language's *reserved words*). In order to be imported, files must also be present in the current directory.
`Main` and auxiliary functions have a structure similar to each other. Each function is identi-

fied by its `idFun`, *i.e.* its name or identifier. The characters which surround the `idFun` will allow the distinction between the `main` and auxiliary functions, these being `BB` – "double bars" `||main_id||` or `SB` – "single bars" `|aux_id|`, respectively. Additionally, each function may (note the *extended-BNF* operator "?") have **parameters** received as inputs. Finally, the function's *pre* and *post*-conditions (written according to QBRICKS-SPEC's grammar) and *circuit* – defined by the `circ` *token* (Listing 3.15) – are specified.

```
imports :
      (IMPORT file=ID)+
      ;

main :
      BB idFun BB (params)? pre circ pos #mainFun
      ;

aux :
      SB idFun SB (params)? pre circ pos #auxFun
      ;
```

Listing 3.15: Syntactic structure of main and auxiliary functions

3.3.3 *Function parameters and types*

In the case where parameters are received by a function, these will be represented in a list, separated by commas (`VG=","`), surrounded by parenthesis (`OP="("` and `CL=")"`). Each of these parameters must have a `type` and an `id`, its **name**. There are **five** different data types in iQBRICKS:

- Integer – `INT="int"`
- Float – `FLOAT="float"`
- Boolean – `BOOL="bool"`
- Circuit – `CIRC="circ"`
- Quantum register – `QREG="qreg"`

Listing 3.16 illustrates how function parameters and types are specified.

```
params : OP param (VG param)* CL #funParams
      ;

param : ptype=type id=ID #singlePar
      ;

type :
      INT #intType
      | FLOAT #floatType
```

```
      | BOOL #boolType
      | CIRC #circType
      | QREG #qrType
      ;
```

Listing 3.16: Syntactic structure of function parameters

3.3.4 *Quantum circuits*

As it was mentioned before, each function will represent a *quantum circuit* family. A circuit is identified firstly by the *token* `CIRC` (`CIRC="circ"`), and receive a list of *quantum registers* (separated by commas). Each of these registers consists in its identifier (`id`) and an optional size parameter ('`range`'), surrounded by the *tokens* `ROP="["` and `RCL="]"`, where its size may be specified. This type of quantum register is the same that will be used in gate applications and *for* loops, where the '`range`' operator may now be used to iterate through a register (Listing 3.17). There are four different ways to use the '`range`' operator:

1. as a *term* expression - which refers **to a specific index** of the register, *e.g.* `[i+1]`
2. iterating **up to** a specific index by placing a `TP` (`TP=":"`) *token* before a *term*, *e.g.* `[:-1]`
3. iterating **from** a specific index (up to the last one) by placing a `TP` *token* after a *term*, *e.g.* `[1:]`
4. iterating **over an interval** between two specific indexes by placing two `PT` (`PT="."`) *tokens* between the "`start`" and "`end`" *terms*, *e.g.* `[i..i+3]`

Note that a *term* specified by the expression "`-1`" will refer to a register's **last** index.
Finally, a circuit's body appears after the *token* `ARROW` (`ARROW="->"`).

```
circ :
      CIRC id_list (ARROW circbody=body)? #regCirc
      ;

id_list :
      qReg (VG qReg)* #qrList
      ;

qReg :
      id=ID (ROP size=range RCL)? #qr
      ;

range :
      term #termRange
      | TP term #uptoRange
      | term TP #fromRange
      | start=term PT PT end=term #intervalRange
      ;
```

Listing 3.17: Syntactic structure of circuit and quantum registers

3.3.5 *Body and Instructions*

A body consists in an **assertion** followed by a *list of* (0 or more) **instructions**.

An assertion can either be empty or identified by the *token* `ASSERT` (`ASSERT="assert"`) followed by a list of `FORMULA` tokens, separated by commas (`VG=","`) each wrapped around curly braces (`COP="{", CCL="}"`).
There are two types of instructions: *regular* and *return* instructions. *Return* instructions are identified by the *token* `RET` (`RET="return"`) and can either be empty or include an expression to be returned. They may only appear at the end of a function's description. *Regular* instructions are ensued by an assertion and take one of the following forms:

- `for` instruction
- `if/else` instruction
- `gate application` instruction
- `control` gate instruction
- `conjugated` block instruction

Listing 3.18 illustrates how the rules mentioned above are specified.

```
body : assert_ (instr)* #regBody
     ;

assert_ : #emptyAssert
     | ASSERT COP? FORMULA (VG FORMULA)* CCL? #assertSpec
     ;

FORMULA : COP ~[{}\n]+ CCL ;

instr :
     (forinst=for_ | ifinst=if_ | applyinst=apply | ctlinst=control | conjinst=
         conjugated)
        assert_ #regInst
     | RET (expr)? #retInst
     ;
```

Listing 3.18: Syntactic structure of body and instructions

3.3.6 *For-loops*

A *for loop* consists of: an iterator **variable** (identified by "`var`"), an iteration **range** (identified by "`iter`"), a loop **invariant** and a **body**. There are two types of expressions that can represent the iteration:

1. one which makes use of the "`range`" operator (`RANGE="range"`), so that the iterator will go up to the value specified by a *term* expression or a quantum register;

2. the other simply consists of a quantum register, whose size determines a specific index range.

Invariants are identified by the *token* `INVARIANT` (`INVARIANT="invariant"`) followed by a list of `FORMULA` tokens, separated by commas (`VG=","`) each wrapped around curly braces, in a similar way to the aforementioned *assert* expressions. Listing 3.19 illustrates how the rules mentioned above are specified. *Term* and *Quantum register* expressions are reviewed in Subsections 3.3.11 and 3.3.4, respectively.

```
for_ :
      FOR var=ID IN iteration=iter COP inv=invariant forbody=body CCL #forLoop
      ;

iter :
      RANGE OP (expvalue=term | qrvalue=qReg) CL #rangeIter
      | qReg #qrIter
      ;

invariant :
      INVARIANT COP? FORMULA (VG FORMULA)* CCL? #invSpec
      ;
```

Listing 3.19: Syntactic structure of the *for* loop

3.3.7 *If/else conditionals*

If statements (Listing 3.20) are defined by a **Boolean expression** (`cond`) and a **body** – surrounded by curly brackets "{" and "}" – for the case where the **Boolean expression** is true (`ifbody`). Optionally, a **body** (`elsebody`) can be placed after the reserved-word `ELSE="else"`, and corresponds to the case where the **Boolean expression** is false.

```
if_ :
      IF cond=expr
      COP ifbody=body CCL
      (ELSE COP elsebody=else_ CCL)? #condIf
      ;

else_:
      body #elsebody
      ;
```

Listing 3.20: Syntactic structure of *if/else* conditional statements

3.3.8 *Gate application instructions*

Gate application instructions are used to append quantum gates to a circuit. These gates can either be circuits defined by auxiliary functions or by one of the following primitive quantum gates:

- *Hadamard* gate
- *Rotation* gates around axis X, Y and Z
- *Pauli* gates X, Y and Z
- *Phase shift* gate
- T and S gates
- *Swap* gate

Auxiliary functions can either be defined inside a program, or imported from one. Such functions can be applied as gates by using *Function* and *reverse function* operations, which take a list of arguments as inputs – corresponding to the parameters received by this function – separated by commas and surrounded by parenthesis. *Rotation* and *Phase shift* gates receive two parameters as inputs: the *angle* and the *register* where the gate will be applied. The remaining gates (H, X, Y, Z, T, S) receive one single parameter and the *swap* gate receives two, which correspond to the *registers* on which the gate will act. Listing 3.21 illustrates how the rules mentioned above are specified.

```
apply :
      fun=idFun OP (fargs=args)? CL #funApply
      | REVERSE OP fun=idFun OP (fargs=args)? CL CL #revApply
      | HAD OP qr=qReg CL #hadApply
      | XGATE OP qr=qReg CL #xApply
      | YGATE OP qr=qReg CL #yApply
      | ZGATE OP qr=qReg CL #zApply
      | TGATE OP qr=qReg CL #tApply
      | SGATE OP qr=qReg CL #sApply
      | RY OP angle=ang VG qr=qReg CL #ryApply
      | RZ OP angle=ang VG qr=qReg CL #rzApply
      | RX OP angle=ang VG qr=qReg CL #rxApply
      | PHASE OP angle=ang VG qr=qReg CL #phApply
      | SWAP OP qrL=qReg VG qrR=qReg CL #swapApply
      ;

args :
      term (VG term)* #funArgs
      ;

ang :
      term #angTerm
      ;
```

Listing 3.21: Syntactic structure of a *gate application*

3.3.9 *Controlled gate instructions*

Controlled gates are used to append controlled quantum gates to a circuit (Listing 3.22). These gates consist in:

1. The standard *C-NOT*, *Toffoli* and *Fredkin* gates, receiving the control and target registers, in this order, as arguments;

2. A "*with control*" operator, which takes a list of registers to be used as controls – separated by commas – and *any other* gate (either an *apply* or by a *control* instruction) – surrounded by parenthesis – as parameters.

```
control :
      WITHCTL ctlqrs=id_list OP ctlgate=apply CL #applyControl // with control q[0],q
          [1] (RZ(1,q[2]))
      | WITHCTL ctlqrs=id_list OP ctlgate=control CL #multiControl // with control q
          [0],q[1] (cnot(q[2],q[3]))
      | CNOT OP ctlqr=qReg VG tqr=qReg CL #cnotControl // cnot(q[0],q[2])
      | TOFF OP ctl1=qReg VG ctl2=qReg VG tg=qReg CL #toffControl // toff(q[0],q[1],q
          [2])
      | FRED OP ctl1=qReg VG tg1=qReg VG tg2=qReg CL #fredControl // fred(q[0],q[1],q
          [2])
      ;
```

Listing 3.22: Syntactic structure of *control* gates

3.3.10 *Conjugate-based gate instructions*

Conjugate-based instructions are used to apply the *conjugate-based* operator to a circuit. Such an instruction contains: a **gate application instruction** (`applyinst`) – corresponding to the gate being applied as the conjugated operator – and a **body** (`conjbody`) – which corresponds to the list of instructions applied in between the *conjugate-based* (Listing 3.23).

```
conjugated :
      WITHCJG OP? applyinst=apply CL? COP conjbody=body CCL
      ;
```

Listing 3.23: Syntactic structure of a *conjugated* instruction

3.3.11 *Expressions and atoms*

Expressions are used in *return* (Listing 3.18) and *if* (Listing 3.20) instructions and represent logical expressions. Two expressions can be compared using the logical operators: `EQ="=="`, `NEQ="!="`, `GT=">"`, `LT="<"`, `GEQ=">="` and `LEQ="<="`. An *expression* can be a *term*, which consists in an arithmetic expression where terms can be combined using arithmetic operators: `POW="^"`, `MUL="*"`, `EQ="/"`, `PLUS="+"`, `MINUS="-"`. A *term* expression can also be a unary term, with the operators `MINUS="-"`, `SQRT="sqrt"` (square-root) for *term* values and `LEN="len"` (length) for quantum *registers*. *Terms* can be surrounded by parenthesis in order to indicate the priority for their calculation. Lastly, a *term* can be defined by an *atom* which can be: a **number** value, a **variable** name or the number **Pi**.

```
expr : term #termExpr
      | left=expr op=EQ right=expr #eqExpr
      | left=expr op=GT right=expr #gtExpr
      | left=expr op=LT right=expr #ltExpr
      | left=expr op=GEQ right=expr #geqExpr
      | left=expr op=LEQ right=expr #leqExpr
      | left=expr op=NEQ right=expr #neqExpr
      | OP expr CL #parenExpr
      ;

term : atom #atomTerm
      | left=term op=POW right=term #powTerm
      | left=term op=MUL right=term #mulTerm
      | left=term op=DIV right=term #divTerm
      | left=term op=PLUS right=term #addTerm
      | left=term op=MINUS right=term #subTerm
      | OP term CL #parenTerm
      | unOp #unaryTerm
      ;

unOp : MINUS term #negUnary
      | LEN OP qReg CL #lenUnary
      | SQRT OP value=term CL #sqrtUnary
      ;

atom : value = NUM #numAtom
      | pi = PI #piAtom
      | var = ID #varAtom
      ;
```

Listing 3.24: Syntactic structure of *expressions*

4

A VERIFICATION FRAMEWORK FOR IQBRICKS

The present chapter details the implementation of the translation process from an iQbricks program to a Qbricks program. The chapter starts by presenting a solution design for the translation process. It then provides a detailed explanation of the data structures created during the translation process and the translation process itself between these structures.

4.1 SOLUTION DESIGN

The chosen architecture for the integration of iQbricks in Qbricks framework is shown in Figure 17. It can be boiled down to four stages:

1. The iQbricks program is processed by the Antlr language recognizer, which creates a concrete syntax tree (CST) representing how the input text aligns with the iQbricks grammar;
2. the generated CST is then translated to a *Java* AST structure by a *Visitor* class named "`ASTBuilder`", which extends the *base Visitor* class generated by Antlr;
3. an AST *Evaluator* takes the *Java* AST and converts it into an equivalent AST written in *Ocaml*. This process involves defining translation rules for each type of node in the *Java* AST. Essentially, the *Evaluator* provides a mapping from the *Java* AST to the *Ocaml* AST.
4. the generated *Ocaml* AST structure is finally processed by another AST *Evaluator*. This *Evaluator* maps the *Ocaml* AST to the Qbricks language by defining a translation process for each created AST data-type.

The translated program can then be tested using *Why3* (Filliâtre and Paskevich, 2013), a specification environment that generates a set of *proof obligations* for a specified program. These obligations can then be satisfied to certify the program.

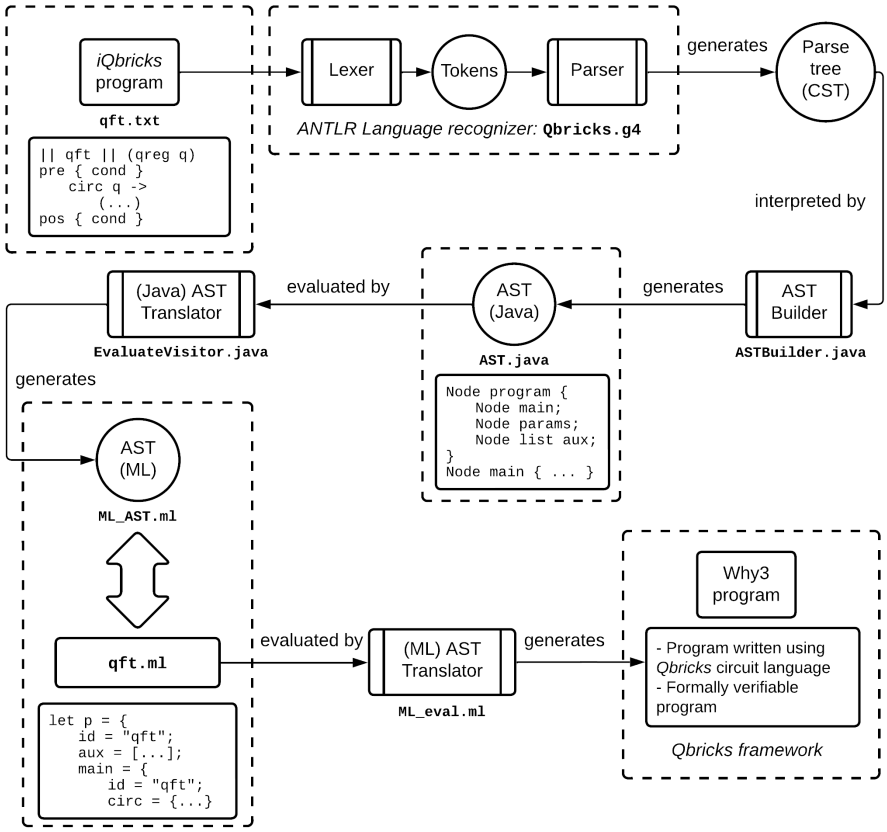


Figure 17: Chosen architecture for the implementation of iQbricks

4.2 Java AST structure

In order to build an *AST* which represents a program's information, we must first define a structure where all the possible nodes are contemplated. Along this section, we cover the process of defining the *AST* nodes. Note that the methods for *getting* and *setting* node variables are not present in the node definitions, since their implementation is trivial. Nevertheless, the full code can be seen in Carneiro (2023).

4.2.1 *AST definition*

Firstly, we create an *abstract class* which is extended by the *children* nodes. This can be done easily with the code in Listing 4.1.

```
public abstract class AST {}
```

Listing 4.1: General *AST* definition

4.2.2 *Program Node*

As it was explained in Section 3.3, a program consists of:

- optional file *imports*,
- a *main* function,
- optional *auxiliary* functions.

A *program node* has the structure presented in Listing 4.2. This node extends the previously created *AST* Node.

```
class ProgramNode extends AST {
   public ImportsNode imports;
   public MainNode main;
   public List<AuxNode> auxList;
}
```

Listing 4.2: Definition of a program node

4.2.3 *Imports Node*

In iQbricks *imports* are optional and therefore possibly non-existent. Thus one of this node's variable is a *Boolean* which expresses the existence of *imports*. If *imports* exist the corresponding files names are represented in a list of *String*. The definition for an *imports node* is shown is Listing 4.3.

```
class ImportsNode extends ProgramNode {
   private List<String> files;
   public boolean hasImports;
}
```

Listing 4.3: Definition of an import node

4.2.4 *Main and Auxiliary Nodes*

Main and *auxiliary nodes* have similar structures, since they both represent *functions*. These nodes extend the *program node*, and contain the following information:

- the function's *name*,
- a Boolean which states if the function has *parameters*, and a *parameters node* for the case where it does,
- a *circuit node*,

- *pre* and *post-condition nodes*.

Main and *auxiliary* node definitions are presented in Listings 4.4 and 4.5.

```
class MainNode extends ProgramNode {
   private String id;
   public Boolean hasParams;
   public CircNode circ;
   public ParamsNode params;
   public PreNode pre;
   public PosNode pos;
}
```

Listing 4.4: Definition of a main function node

```
class AuxNode extends ProgramNode {
   private String id;
   public CircNode circ;
   public ParamsNode params;
   public PreNode pre;
   public PosNode pos;
   public Boolean hasParams;
}
```

Listing 4.5: Definition of an auxiliary function node

4.2.5 *Parameters Node*

A *parameters node* consists of a list of *single-parameter nodes*. Each *single-parameter node* contains a *type* and an *identifier* – its name. *Parameters* and *single-parameter node* structures are presented in Listing 4.6.

```
class ParamsNode extends MainNode {
   private List<SingleParam> params;
}

class SingleParam extends ParamsNode {
   private String type;
   private String id;
}
```

Listing 4.6: Definition of a parameters node

4.2.6 *Pre and Post-condition Nodes*

Pre and *post-conditions* have similar structures, since their sole purpose is to express program specifications using Qbricks-Spec language. Thus, they contain a list of *String*, where each element corresponds to a logical expression. *Pre* and *post-condition node* structures are presented in Listing 4.7.

```
class PreNode extends MainNode {
   private List<String> conds;
}

class PosNode extends MainNode {
   private List<String> conds;
}
```

Listing 4.7: Definition of *Pre* and *post*-condition nodes

4.2.7 *Circuit Node*

A *Circuit Node* contains two *children-nodes*: one for the circuit's quantum **registers** and the other for the circuit's **body**. *Circuit node* structure is presented in Listing 4.8.

```
class CircNode extends MainNode {
   private CircIds ids;
   private BodyNode body;
}
```

Listing 4.8: Definition of a circuit node

4.2.8 *Circuit registers Node*

A *Circuit registers Node* contains a list of *Quantum register Nodes* that represent the quantum registers composing a circuit.

```
class CircIds extends CircNode {
   private List<QregNode> regs;
}
```

Listing 4.9: Definition of a circuit identifiers node

4.2.9 *Quantum register and Range Nodes*

Quantum register nodes are present in several different node structures:

- Circuit registers Node, previously introduced,
- *For-loop* Iteration Node, for the case of iteration through a register,
- Gate application Nodes (which includes both *single* and *multi-qubit* gates).

A quantum register has a *string* identifier "`id`", an optional *range node* and a Boolean for checking the existence of the latter.

```
class QregNode extends AST {
   public RangeNode range;
   public Boolean has_range;
   public String id;
}
```

Listing 4.10: Definition of a quantum register node

Range nodes, used in *quantum register* nodes, contain a *string* identifier "`iterator`" (inherited from the *quantum register node*), and a *term node* for both the *start* and *end* of the iteration range.

```
class RangeNode extends QregNode {
   public String iterator;
   public TermNode start;
   public TermNode end;
}
```

Listing 4.11: Definition of a range node

4.2.10 *Body and Assertion Nodes*

A *body node* contains a list of *instruction nodes* and an *assertion node*. *Body nodes* extend *circuit nodes* and are used by the *instruction nodes for*, *if*, *else* and *conjugated*.

```
class BodyNode extends CircNode {
   private List<InstrNode> bodyInstr;
   private AssertNode assertion;
}
```

Listing 4.12: Definition of a body node

An *assertion node* has a similar structure to *pre* and *post-condition* nodes in Subsection 4.2.6. It consists of a list of conditional expressions represented by *strings*. *Assertion nodes* are present in all *instruction nodes*, since all instructions can be followed by an assertion.

```
class AssertNode extends BodyNode {
   private List<String> assertions;
}
```

Listing 4.13: Definition of an assertion node

4.2.11 *Instruction Node*

An *instruction node* is described by **one** of the following:

- *For node*,
- *If node*,

- *Apply node*,
- *Control node*,
- *Conjugated node*,
- *Return node*.

Therefore, each one of these nodes **extends** an *instruction node*.

4.2.12 *For and Invariant Nodes*

A *for node* contains four different *nodes*:

1. *Invariant node* which represents the *loop-invariant* for the cycle;
2. *For-iteration node* which contains information about the iteration values for the loop;
3. *Body node* which corresponds to the loop's *body*;
4. *Assertion node* which corresponds to the loop's *post-condition*.

```
class ForNode extends InstrNode {
   private InvariantNode invariant;
   private ForIter iter;
   private BodyNode body;
   private AssertNode assertion;
}
```

Listing 4.14: Definition of a for node

An *invariant node* has a similar structure to an *assertion node*. It is defined by a list of conditional expressions, represented by *strings*.

```
class InvariantNode extends ForNode {
   private List<String> conds;
}
```

Listing 4.15: Definition of an invariant node

4.2.13 *For-iteration Node*

A *for-iteration node* contains the information regarding a *for-loop*'s iteration range. It consists of:

- a *string* identifier "`iterator`", representing the loop's iterator name;
- a *Boolean* "`iterQr`" which is true if a *quantum register node* is being iterated and false if instead an *term node* is being iterated;

- a *Boolean* "`range`" which is true if the *range operator* is being used for the iteration;
- a *term node* and a *quantum register node*: each *for-iteration node* will contain either a *term* or a *quantum register node*, which will represent the loop's iteration interval.

Method `setIterable` (Listing 4.16) checks whether the iterable object is an instance of the *Term node* class or the *Quantum register node* class. If it is a *Term Node*, it sets the `iterableExpr` field to the iterable object and sets the `iterQr` field to false. If it is a *Quantum register node*, it sets the `iterableQr` field to the iterable object and sets the `iterQr` field to true.

```
class ForIter extends ForNode {
   private String iterator;
   private Boolean iterQr;
   private Boolean range;
   private TermNode iterableExpr;
   private QregNode iterableQr;

   public void setIterable(AST iterable){
      if (iterable instanceof TermNode) {
         this.iterableExpr = (TermNode) iterable;
         iterQr = false;
      } else {
         this.iterableQr = (QregNode) iterable;
         iterQr = true;
      }
   }
}
```

Listing 4.16: Definition of a for-iteration node

4.2.14 *If and If-condition Nodes*

An *if node* has four fields: `withElse`, `cond`, `assertion`, and `ifBody`, as well as an optional field `elseBody`:

- The `withElse` field is a *Boolean* that indicates whether the *if* statement has an *else* clause;
- The `cond` field represents the condition of the *if* statement;
- The `assertion` represents the *if* statement's assertion;
- The `ifBody` field is an object of the *Body Node* class, which represents the body of the *if* statement;
- The `elseBody` field, if present, is also an object of the *Body Node* class and represents the body of the *else* clause.

The *If-condition node* has a single field: `cond`, which is an object of the *ExpressionNode* class and represents the condition of the *if* statement.

```
class IfNode extends InstrNode {
   private Boolean withElse;
   private IfCond cond;
   private AssertNode assertion;
   private BodyNode ifBody;
   private BodyNode elseBody;
}

class IfCond extends IfNode {
   private ExpressionNode cond;
}
```

Listing 4.17: Definition of If and If-condition nodes

4.2.15 *Apply node*

An *Apply node* corresponds to a (*non-control*) gate application. These gate applications are defined by four different types of *nodes*, depending on the number of the gate's parameters[1]:

1. Nodes with only **one parameter** – the *quantum register* – corresponding to gates: H, X, Y, Z, T, S (Listing 4.18);

```
class SingleApply extends ApplyNode {
   private QregNode qreg;
   private AssertNode assertion;
}
```

Listing 4.18: Definition of a Single argument gate application node

2. Nodes with **two parameters** – the *quantum register* and the rotation angle θ – corresponding to gates: $R_x(\theta), R_y(\theta), R_z(\theta), Ph(\theta)$ (Listing 4.19);

```
class DoubleApply extends ApplyNode {
   private QregNode qreg;
   private TermNode angle;
   private AssertNode assertion;
}
```

Listing 4.19: Definition of a two argument gate application node

3. *Swap gate* node, with two *quantum registers* (Listing 4.20);

```
class SwapApply extends ApplyNode {
   private QregNode qleft;
   private QregNode qright;
   private AssertNode assertion;
}
```

Listing 4.20: Definition of a Swap gate application node

1 Note that all gate applications are proceeded with an *assertion*, thus all *apply nodes* contain an *assertion node*.

4. *Function* and *reverse-function* application nodes, which can receive **multiple arguments**, and will therefore store these in a *list* of *Term nodes*(Listing 4.21);

```
class FunApply extends ApplyNode {
   private String funID;
   public List<TermNode> termArgs;
   private AssertNode assertion;
}
```

Listing 4.21: Definition of a function gate application node

4.2.16 *Control node*

A *control node* consists of one of four controlled gate applications[2]:

- *With-controlled* operator, which can receive multiple *control* registers as arguments, and either an *Apply node* or a *Control node* as the gate being used for the control. The two different cases will be distinguished by the Boolean `isMulti`, which will be true in the case where the gate is represented by a *Control node* (Listing 4.22). Note that in both cases, the *target* registers will be specified in the gate *node*'s structure;

```
class WithCtlNode extends CtlNode {
   private List<QregNode> ctlArgs;
   private ApplyNode ctlGate;
   private CtlNode ctlMulti;
   private Boolean isMulti;
   private AssertNode assertion;
}
```

Listing 4.22: Definition of a *with-controlled* gate application node

- *C-NOT* gate (Listing 4.23), which receives the *control* and *target quantum registers*;

```
class CnotNode extends CtlNode {
   private QregNode ctl;
   private QregNode target;
   private AssertNode assertion;
}
```

Listing 4.23: Definition of a *C-NOT* gate application node

- *Toffoli* gate (Listing 4.24) which receives the first and second *control* and *target quantum registers*;

```
class ToffNode extends CtlNode {
   private QregNode ctl1;
   private QregNode ctl2;
   private QregNode target;
   private AssertNode assertion;
```

2 Note that similarly to gate applications, all control instructions are proceeded with an *assertion*, thus all *control nodes* contain an *assertion node*.

```
}
```

Listing 4.24: Definition of a *Toffoli* gate application node

- *Fredkin* gate (Listing 4.25) which receives the first and second *control* and *target quantum registers*;

```
class FredNode extends CtlNode {
   private QregNode ctl1;
   private QregNode ctl2;
   private QregNode target;
   private AssertNode assertion;
}
```

Listing 4.25: Definition of a *Fredkin* gate application node

4.2.17 *Conjugated node*

A *conjugated node* contains three different nodes (Listing 4.26):

- An *apply node*, which represents the gate or circuit being used as the *conjugated-basis*;
- A *body node*, consisting in the conjugated block's body;
- An *assertion node*, corresponding to the *conjugated instruction*'s assertion.

```
class ConjNode extends InstrNode {
   private ApplyNode apply;
   private BodyNode body;
   private AssertNode assertion;
}
```

Listing 4.26: Definition of a *conjugate-based* gate application node

4.2.18 *Return node*

A *return node* consists of a Boolean which states if the *return* expression is *empty* and an *expression node*, which represents the expression being returned.

```
class RetNode extends InstrNode {
   private Boolean argBool;
   private ExpressionNode args;
}
```

Listing 4.27: Definition of a *return*

4.2.19 *Expression node*

An *expression node* represents any type of expression in this language. Thus, it is extended by several other nodes:

- `InfixExpressionNode` extends the `ExpressionNode` class, and represents an *infix expression* which is an expression that has a left and right operand and an operator between them. It has two fields: `Left`, that represents the left operand of the expression, and `Right`, that represents the right operand of the expression (Listing 4.28).

```
class InfixExpressionNode extends ExpressionNode {
   private ExpressionNode Left;
   private ExpressionNode Right;
}
```

Listing 4.28: Definition of an *infix* expression node

- `EqualNode`, `GTNode`, `LTNode`, `GEQNode`, `LEQNode`, and `NEqualNode` classes all extend the `InfixExpressionNode` class and represent specific infix expressions: `EqualNode` for equal to (==), `GTNode` for greater than (>), `LTNode` for less than (<), `GEQNode` for greater than or equal to (>=), `LEQNode` for less than or equal to (<=), and `NEqualNode` for not equal to (!=) (Listing 4.29).

```
class EqualNode extends InfixExpressionNode {}
class GTNode extends InfixExpressionNode {}
class LTNode extends InfixExpressionNode {}
class GEQNode extends InfixExpressionNode {}
class LEQNode extends InfixExpressionNode {}
class NEqualNode extends InfixExpressionNode {}
```

Listing 4.29: Definition of different expression operations nodes

4.2.20 *Term node*

A *Term Node* extends the *Expression Node* class and represents a simple term in an expression. A *term* expression can represent different operations: an arithmetic expression taking two *terms*; a *unary* or an *atom*.

A *Term node* is extended by the following nodes:

- `InfixTermNode` extends the `TermNode` class, and represents an *infix term* which is a term that has a left and right operand and an operator between them. It has two fields: `Left`, that represents the left operand of the expression, and `Right`, that represents the right operand of the expression (Listing 4.30).

```
class InfixTermNode extends TermNode {
   private TermNode Left;
```

```
   private TermNode Right;
}
```

Listing 4.30: Definition of an *infix Term* expression node

- `PowerNode`, `MulNode`, `DivNode`, `AddNode` and `SubNode` classes all extend the `InfixTermNode` class and represent specific infix terms: `PowerNode` for power (`^`), `MulNode` for multiplication (`*`), `DivNode` for division (`/`), `AddNode` for addition (`+`) and `SubNode` for subtraction (`-`) (Listing 4.31).

```
class PowerNode extends InfixTermNode {}
class MulNode extends InfixTermNode {}
class DivNode extends InfixTermNode {}
class AddNode extends InfixTermNode {}
class SubNode extends InfixTermNode {}
```

Listing 4.31: Definition of different *term* expression operations nodes

- `ParenNode` represents a term expression surrounded by parenthesis, thus its only parameter (`term`) will correspond to the respective *term* (Listing 4.32).

```
class ParenNode extends TermNode {
   private TermNode term;
}
```

Listing 4.32: Definition of a parenthesis *term* node

4.2.21 *Unary and Length nodes*

A `UnOpNode` extends the `TermNode` class, and represents a *unary operator term* which is a term that has (1) a *unary* operator (*length*, *square-root* or *negative sign*) and (2) an *inner term*. It has two fields: `op`, that represents the applied operator as a string, and `InnerTerm`, that represents the operation's *inner term* (Listing 4.33).

```
class UnOpNode extends TermNode {
   private String op;
   private TermNode InnerTerm;
}
```

Listing 4.33: Definition of an *unary* term node

`LenNode` represents a *unary term* where the *length* keyword is used as operator, and is applied to a *quantum-register node* (Listing 4.34).

```
class LenNode extends TermNode {
   private QregNode QrTerm;
}
```

Listing 4.34: Definition of a *length* term node

4.2.22 *Atom node*

An *Atom node* extends the `TermNode` class, and corresponds to an *atom*, which can be a **number** value, a **variable** name or the number **Pi**. This node's structure will therefore contain two fields: `Type` and `Value`, both represented by strings (Listing 4.35).

```
class AtomNode extends TermNode {
   private String Value;
   private String Type;
}
```

Listing 4.35: Definition of an *atom* term node

4.3 AST BUILDER

In this section the process of building the actual *AST* from the generated *Parse-tree* will be thoroughly explained.

ANTLR automatically generates a *base Visitor* class for traversing the *parse-tree* generated from a source file – in this case, a program written in IQBRICKS. The generated *Visitor* contains a *visit method* for every *rule* present in the language's **grammar**. This *base Visitor* class provides an empty implementation which can be extended to create a visitor which only needs to handle a subset of the available methods. The generated *Visitor*'s implementation is shown (partially) in Listing 4.36.

```
public class QbricksBaseVisitor<T> extends AbstractParseTreeVisitor<T> implements
    QbricksVisitor<T> {
   @Override
   public T visitProgram(QbricksParser.ProgramContext ctx) {
      return visitChildren(ctx);
   }

   @Override
   public T visitImports(QbricksParser.ImportsContext ctx) {
      return visitChildren(ctx);
   }

   @Override
   public T visitMainFun(QbricksParser.MainFunContext ctx) {
      return visitChildren(ctx);
   }

   @Override
   public T visitAuxFun(QbricksParser.AuxFunContext ctx) {
      return visitChildren(ctx);
   }
```

Listing 4.36: Generated *Base Visitor*

The purpose of the *Visitor* that extends the generated *Base Visitor* is to traverse through the generated *parse-tree*, building a *new* tree with the same structure of the *AST* described in

Section 4.2. Thus this *Visitor* class is named "*ASTBuilder*", and its definition and methods will be described in the following subsections.

4.3.1 *Creating the AST Builder*

In order to create the *AST Builder*, a new class is created that extends the automatically generated *Base Visitor* (Listing 4.36), as shown in Listing 4.37.

```
public class ASTBuilder extends QbricksBaseVisitor<AST> {
   // Visit methods for grammar rules
}
```

Listing 4.37: *AST Builder* definition

Visit methods generate *AST* nodes and assign values to their parameters with information from the generated *parse-tree*. Each *visit* method takes a single argument, "`ctx`", which is the root context of the parse tree for the respective *grammar rule*.

4.3.2 *Program Visit*

Revisiting the grammar definition for the `program` rule in Listing 4.38, it states that a program is composed of three parts: an optional "`imports`" section, a required "`main`" section, and zero or more "`aux`" sections.

```
program :
      (imports)? main (aux)*
      ;
```

Listing 4.38: Program rule definition

The *visit* method for this rule (Listing 4.39)

- creates three new *AST* nodes: *Program Node* "`node`", *Imports Node* "`imports`", and *Main Node* "`main`";
- assigns the result of *visiting* the "`main`" section from the parse-tree context to the "`main`" node
- creates a list of *Aux Node* objects "`auxList`";
- checks if the "`imports`" section from the parse-tree context is not null, if so assigns the the result of *visiting* this section to the "`imports`" node;
- loops through the "`aux`" sections in the parse-tree context and assigns the result of visiting each specific section to the "`aux`" variable. The "`aux`" variable is then added to the "`auxList`" list of nodes;

- sets the values of the "imports", "main", and "auxList" nodes on the "node" object and returns the "node" object.

```
@Override
   public AST visitProgram(QbricksParser.ProgramContext ctx) {
      ProgramNode node = new ProgramNode();
      ImportsNode imports = new ImportsNode();
      MainNode main = (MainNode) visit(ctx.main());
      List<AuxNode> auxList = new ArrayList<>();
      AuxNode aux;
      if (ctx.imports()!=null)
         imports = (ImportsNode) visit(ctx.imports());
      for (int c = 0; c < ctx.aux().size(); ++c){
         aux = (AuxNode) visit(ctx.aux(c));
         auxList.add(aux);
      }
      node.setImports(imports);
      node.setMain(main);
      node.setAuxList(auxList);
      return node;
   }
```

Listing 4.39: Program *visit method* definition

4.3.3 *Imports Visit*

The grammar definition for the imports rule in Listing 4.40 states that the "imports" section is defined by one or more occurrences of the "IMPORT" keyword ("import") followed by an identifier "file" which is matched to the token "ID" (a sequence of one or more letters, underscores or digits that starts with a letter).

```
imports:
      (IMPORT file=ID)+
      ;
```

Listing 4.40: Imports rule definition

The *visit* method for this rule (Listing 4.41)

- creates a new *Imports Node* "node" and a list of strings "files";
- checks if the imports context from the parse-tree is empty, if not it sets the "hasImports" flag to true on the "node" and then it loops through all the "ID" – which represent the files' names – found in the imports context and adds them to the "files" list;
- sets the values on the "files" list on the "node" and returns the "node" object.

```
@Override
   public AST visitImports(QbricksParser.ImportsContext ctx) {
      ImportsNode node = new ImportsNode();
      List<String> files = new ArrayList<>();
```

```
      if(!ctx.isEmpty()){
         node.setHasImports(true);
         for (int i=0; i<ctx.ID().size(); i+=2){
            files.add(ctx.ID(i).getText());
         }
      }
      node.setFiles(files);
      return node;
   }
```

Listing 4.41: Imports *visit method* definition

4.3.4 *Main and Auxiliary function Visit*

The grammar definition for the `main` and `aux` function rules in Listing 4.42 states that a *function* is defined by and *identifier* "`idFun`" surrounded by the characters "`||`" if it is the `main` function or by the character "`|`" if it is an `auxiliary` one, followed by an optional section of *parameters* "`params`" and the *function*'s *pre-condition* "`pre`", *circuit* definition "`circ`" and *post-condition* "`pos`" sections.

```
main :
      BB idFun BB (params)? pre circ pos #mainFun
      ;

aux :
      SB idFun SB (params)? pre circ pos #auxFun
      ;
```

Listing 4.42: Main and Aux rules definition

The *visit* methods for both *main* (Listing 4.43) and *aux* (Listing 4.44) rules are identical. In detail, they

- create a new *Main* or *Auxiliary node* "`node`";
- create three new *AST* nodes: *Circ node* "`circ`", *Pre-condition node* "`pre`" and *Post-condition node* "`pos`", and assign the result of *visiting* each respective section from the parse-tree context to its *node*;
- create a *Parameter node* "`pNode`" and check if the "`params`" section exists by *counting* the *number of children* in the root context. If it exists, the method assigns the result of *visiting* the "`params`" section from the parse-tree context to "`pNode`" and the "`hasParams`" flag is set to true on the "`node`"; if not, "`pNode`" is set to `null` and the "`hasParams`" flag is set to false;
- set the *identifier* value in the "`node`" to the "`idFun`" section's text;
- set the values of the "`circ`", "`pNode`", "`pre`", and "`pos`" nodes on the "`node`" object and return the "`node`" object.

```
@Override
   public AST visitMainFun(QbricksParser.MainFunContext ctx) {
      MainNode node = new MainNode();
      CircNode circ = (CircNode) visit(ctx.circ());
      PreNode pre = (PreNode) visit(ctx.pre());
      PosNode pos = (PosNode) visit(ctx.pos());
      ParamsNode pNode;
      if(ctx.getChildCount()>6) {
         pNode = (ParamsNode) visit(ctx.params());
         node.setHasParams(true);
      } else {
         pNode = null;
         node.setHasParams(false);
      }
      node.setID(ctx.idFun().getText());
      node.setCirc(circ);
      node.setParams(pNode);
      node.setPre(pre);
      node.setPos(pos);
      return node;
   }
```

Listing 4.43: Main function *visit method* definition

```
@Override
   public AST visitAuxFun(QbricksParser.MainFunContext ctx) {
      AuxNode node = new AuxNode();
      CircNode circ = (CircNode) visit(ctx.circ());
      PreNode pre = (PreNode) visit(ctx.pre());
      PosNode pos = (PosNode) visit(ctx.pos());
      ParamsNode pNode;
      if(ctx.getChildCount()>6) {
         pNode = (ParamsNode) visit(ctx.params());
         node.setHasParams(true);
      } else {
         pNode = null;
         node.setHasParams(false);
      }
      node.setID(ctx.idFun().getText());
      node.setCirc(circ);
      node.setParams(pNode);
      node.setPre(pre);
      node.setPos(pos);
      return node;
   }
```

Listing 4.44: Auxiliary function *visit method* definition

4.3.5 *Function parameters Visit*

The grammar definition for the *parameters* rule in Listing 4.45 states that the "`params`" section is defined by one or more occurrences of a *single parameter* "`param`" section, separated by a comma "`VG`" and surrounded by parenthesis tokens "`OP`" and "`CL`". A *single parameter* "`param`"

section is defined by a *type* "ptype" section followed by an identifier "id" which is matched to the token "ID".

```
params :
      OP param (VG param)* CL #funParams
      ;

param :
      ptype=type id=ID #singlePar
      ;

type :
      INT #intType
      | FLOAT #floatType
      | BOOL #boolType
      | CIRC #circType
      | QREG #qrType
      ;
```

Listing 4.45: Parameters rule definition

The *visit* method for the *parameters* rule (Listing 4.46)

- creates a new *Parameters Node* "node" and a list of *single parameter* nodes "params";
- loops through the list of parameters from the parse-tree root context and adds the result of *visiting* each "param" section to the "params" list;
- sets the values on the "params" list on the "node" and returns the "node" object.

```
@Override
   public AST visitFunParams(QbricksParser.FunParamsContext ctx) {
      ParamsNode node = new ParamsNode();
      List<SingleParam> params = new ArrayList<>();
      for (int c=0; c < ctx.param().size(); ++c){
         params.add((SingleParam) visit(ctx.param(c)));
      }
      node.setPs(params);
      return node;
   }
```

Listing 4.46: Parameters *visit method* definition

The *visit* method for the *single-parameter* rule (Listing 4.47)

- creates a new *Single-parameter Node* "node";
- sets the values of root context's "ptype" and "id" on the parameter node's *type* and *identifier*.

```
@Override
   public AST visitSinglePar(QbricksParser.SingleParContext ctx) {
      SingleParam node = new SingleParam();
      node.setParam(ctx.ptype.getText(),ctx.id.getText());
      return node;
```

```
    }
```

Listing 4.47: Single-parameter *visit method* definition

4.3.6 *Circuit and Quantum-register Visit*

The grammar definition for the *circ* rule in Listing 4.48 states that the "`circ`" section is defined by a token "`CIRC`", followed by a list of *quantum registers* "`id_list`" and a token "`ARROW`", proceeded by the circuit's *body* "`circbody`" section. Rule `id_list` (Listing 4.48) states that the "`id_list`" section is defined by a list of one or more occurrences of "`qReg`" sections, separated by a "`VG`" token (the *comma* character). Finally, rule `qReg` (Listing 4.48) states that the "`qReg`" section is defined by a *identifier* "`id`" which is matched to the token "`ID`", followed by an optional section, which consists in a *range* section "`size`" surrounded by brackets tokens "`ROP`" and "`RCL`".

```
circ :
      CIRC id_list ARROW circbody=body #regCirc
      ;

id_list :
      qReg (VG qReg)* #qrList
      ;

qReg :
      id=ID (ROP size=range RCL)? #qr
      ;
```

Listing 4.48: Circuit, quantum register and list of registers rules definition

The *visit* method for the *circ* rule (Listing 4.49)

- creates a new *Circuit Node* "`node`", a *Circuit registers Node* "`ids`" and a list of *Quantum register Nodes* "`circQrs`";
- creates a *Body Node* "`body`" and assigns the result of *visiting* the "`circbody`" section from the parse-tree context to the *node*
- creates a variable "`id_list`" which is then assigned the list of *quantum registers* "`id_list`" section from the parse-tree context;
- loops through the list of quantum registers from the "`id_list`" section and adds the result of *visiting* each "`qReg`" section to the "`circQrs`" list;
- sets the value of the *circuit registers* "`circQrs`" on the *Circuit registers Node* "`ids`";
- sets the values of the *Circuit registers Node* "`ids`" and *Body Node* "`body`" on the "`node`" and returns the "`node`" object.

```
@Override
   public AST visitRegCirc(QbricksParser.RegCircContext ctx) {
      CircNode node = new CircNode();
      CircIds ids = new CircIds();
      List<QregNode> circQrs = new ArrayList<>();
      BodyNode body = (BodyNode) visit(ctx.circbody);
      QregNode qr;
      ParseTree id_list = ctx.getChild(1);
      for (int c = 0; c < id_list.getChildCount(); c+=2){
         qr = (QregNode) visit(id_list.getChild(c));
         circQrs.add(qr);
      }
      ids.setRegs(circQrs);
      node.setIds(ids);
      node.setBody(body);
      return node;
   }
```

Listing 4.49: Circuit *visit method* definition

The *visit* method for the *qReg* rule (Listing 4.50)

- creates a new *Quantum register Node* "`qr`" and a *Range Node* "`range`";
- checks if the number of *children* on the root context is superior to one: if so, this means the parse-tree context contains a *range* section. In this case, the result of *visiting* the *range* section is assigned to the "`range`" *node*, and the Boolean "`hasRange`" is set to true; In case there **is not** a *range* section, the method sets the "`range`" *node* to `null` and the Boolean "`hasRange`" to false;
- sets the values of the *Range Node* "`range`" and the identifier "`id`" from the parse-tree context on the "`qr`" *node* and returns the "`qr`" object.

```
@Override
   public AST visitQr(QbricksParser.QrContext ctx) {
      QregNode qr = new QregNode();
      RangeNode range;
      if (ctx.getChildCount()>1) {
         range = (RangeNode) visit(ctx.size);
         qr.setHasRange(true);
      }
      else {
         range = null;
         qr.setHasRange(false);
      }
      qr.setRange(range);
      qr.setId(ctx.id.getText());
      return qr;
   }
```

Listing 4.50: Quantum register *visit method* definition

4.3.7 *Range Visit*

The grammar definition for the *range* rule in Listing 4.45 states that the "`range`" section is defined in four different ways, covered in Subsection 3.3.4:

1. as a *term* expression;
2. iterating **up to** a specific index by placing a "`TP`" (`TP=":"`) *token* before a *term*;
3. iterating **from** a specific index (up to the last one) by placing a "`TP`" *token* after a *term*;
4. iterating **over an interval** between two specific indexes by placing two "`PT`" (`PT="."`) *tokens* between the "`start`" and "`end`" *terms*.

```
range :
      term #termRange
      | TP term #uptoRange
      | term TP #fromRange
      | start=term PT PT end=term #intervalRange
      ;
```

Listing 4.51: Range rule definition

There are four *visit* methods for the *range* rule (Listing 4.46):

1. *Visit* method for a *term* range *node* (Listing 4.52) creates a *Range node* "`node`", a *Term node* "`term`" and assigns the result of *visiting* the "`term`" section from the parse-tree context to the "`term`" variable; it then creates a string "`id`", which is assigned the value of the *parent*'s (*quantum-register*) identifier and set as the *iterator* parameter on the "`node`"; *Term nodes* which represent the *start* and *end* parameters in the *Range node* are both set as "`term`" and the "`node`" is returned;

```
@Override
public AST visitTermRange(QbricksParser.TermRangeContext ctx) {
   RangeNode node = new RangeNode();
   TermNode term = (TermNode) visit(ctx.term());
   String id = ctx.getParent().getChild(0).getText();
   node.setIterator(id);
   node.set(term,term);
   return node;
}
```

Listing 4.52: *Term* range *visit method* definition

2. *Visit* method for an *up-to* range *node* (Listing 4.53) is similar to the *visit* method in Listing 4.52, but the *Term node* which represents the *start* parameter is set as `null` and *end* parameter is set as the "`end`" variable, after which the "`node`" is returned;

```
@Override
public AST visitUptoRange(QbricksParser.UptoRangeContext ctx) {
   RangeNode node = new RangeNode();
```

```
   TermNode end = (TermNode) visit(ctx.term());
   String id = ctx.getParent().getChild(0).getText();
   node.setIterator(id);
   node.set(null,end);
   return node;
}
```

Listing 4.53: *Up to* range *visit method* definition

3. *Visit* method for a *from* range *node* (Listing 4.54) is similar to the *visit* methods in Listings 4.52 and 4.53, but the *Term node* which represents the *start* parameter is set as the "`start`" variable and *end* parameter is set as `null`, after which the "`node`" is returned;

```
@Override
public AST visitFromRange(QbricksParser.FromRangeContext ctx) {
   RangeNode node = new RangeNode();
   TermNode start = (TermNode) visit(ctx.term());
   String id = ctx.getParent().getChild(0).getText();
   node.setIterator(id);
   node.set(start,null);
   return node;
}
```

Listing 4.54: *From* range *visit method* definition

4. *Visit* method for an *interval* range *node* (Listing 4.55) is similar to the *visit* methods in Listings 4.52, 4.53 and 4.54, but the *Term node* which represents the *start* parameter is set as the "`start`" variable and *end* parameter is set as the "`end`" variable, after which the "`node`" is returned;

```
@Override
public AST visitIntervalRange(QbricksParser.IntervalRangeContext ctx) {
   RangeNode node = new RangeNode();
   TermNode start = (TermNode) visit(ctx.getChild(0));
   TermNode end = (TermNode) visit(ctx.getChild(3));
   String id = ctx.getParent().getChild(0).getText();
   node.setIterator(id);
   node.set(start,end);
   return node;
}
```

Listing 4.55: *Interval* range *visit method* definition

4.3.8 *Assertion, Pre and Post-condition and Invariant Visit*

The grammar definition for the `assertion`, `pre-condition`, `post-condition` and `invariant` rules in Listing 4.42 state that all these *specifications* consist in an identifier – respective to each of the rules – followed by a list of (one or more) "`FORMULA`" tokens, surrounded by curly brace tokens "`COP`" and "`CCL`". One particular aspect about the `assertion` rule is that it can

be *empty*, for the cases where the user does not intend to add specification after a certain *instruction*.

```
assert_ : #emptyAssert
      | ASSERT COP? FORMULA (VG FORMULA)* CCL? #assertSpec
      ;

pre :
      PRE COP? FORMULA (VG FORMULA)* CCL? #preSpec
      ;

pos :
      POS COP? FORMULA (VG FORMULA)* CCL? #posSpec
      ;

invariant :
      INVARIANT COP? FORMULA (VG FORMULA)* CCL? #invSpec
      ;
```

Listing 4.56: Specification rules definition

The *visit* methods for the specification rules in Listing 4.56 are identical (Listing 4.57), and

- create a new *node* "`node`";
- create a list of strings for storing "`FORMULA`" tokens from the parse-tree context;
- loop through the list of formulas from the parse-tree root context and add the value of each "`FORMULA`" token to the list of strings;
- set the values of the list of formulas on the "`node`" object and return the "`node`" object.

```
@Override
   public AST visitAssertSpec(QbricksParser.AssertSpecContext ctx) {
      AssertNode node = new AssertNode();
      List<String> asserts = new ArrayList<>();
      for(int c = 0; c < ctx.FORMULA().size(); ++c){
         asserts.add(ctx.FORMULA(c).getText());
      }
      node.setAssertions(asserts);
      return node;
   }

@Override public AST visitPreSpec(QbricksParser.PreSpecContext ctx) {
      PreNode node = new PreNode();
      List<String> preconds = new ArrayList<>();
      for (int c = 0; c < ctx.FORMULA().size(); c++){
         preconds.add(ctx.FORMULA(c).getText());
      }
      node.set(preconds);
      return node;
   }

@Override
   public AST visitPosSpec(QbricksParser.PosSpecContext ctx) {
      PosNode node = new PosNode();
```

```
      List<String> posconds = new ArrayList<>();
      for (int c = 0; c < ctx.FORMULA().size(); c++){
         posconds.add(ctx.FORMULA(c).getText());
      }
      node.set(posconds);
      return node;
   }

@Override
   public AST visitInvSpec(QbricksParser.InvSpecContext ctx) {
      InvariantNode node = new InvariantNode();
      List<String> invariants = new ArrayList<>();
      for (int c = 0; c < ctx.FORMULA().size(); c++){
         invariants.add(ctx.FORMULA(c).getText());
      }
      node.set(invariants);
      return node;
   }
```

Listing 4.57: Specification *visit methods* definition

4.3.9 *Body Visit*

The grammar definition for the *body* rule (Listing 4.58) states that a "`body`" section consists in an *assertion* section followed by a list of **zero or more** *instruction* sections.

```
body :
      assert_ (instr)* #regBody
      ;
```

Listing 4.58: Body rule definition

The *visit* method for the *body* rule (Listing 4.59)

- creates a *Body Node* "`node`", a list of *Instruction Nodes* "`nodeList`" and an *Instruction Node* "`instr`";
- loops through the list of instructions from the *body*'s "`instr`" section and adds the result of *visiting* each "`instr`" section to the "`nodeList`" list;
- sets the value of the "`nodeList`" list and the result of *visiting* the *assertion* section on the "`node`" and returns the "`node`" object.

```
@Override
   public AST visitRegBody(QbricksParser.RegBodyContext ctx) {
      BodyNode node = new BodyNode();
      List<InstrNode> nodeList = new ArrayList<>();
      InstrNode instr;
      for (int c = 0; c < ctx.instr().size(); ++c){
         instr = (InstrNode) visit(ctx.instr(c));
         nodeList.add(instr);
      }
```

```
      node.setBodyInstr(nodeList);
      node.setAssertion((AssertNode) visit(ctx.assert_()));
      return node;
   }
```

Listing 4.59: Body *visit method* definition

4.3.10 *Instruction Visit*

The grammar definition for the *instruction* rule (Listing 4.60) states that an "`instr`" section is defined by any *regular* instruction: *for, if, apply, control* or *conjugated* followed by an *assertion*, or else it is defined by a *return* instruction, which has an optional *expression* section.

```
instr :
      (forinst=for_ | ifinst=if_
      | applyinst=apply | ctlinst=control
      | conjinst=conjugated) assert_ #regInst
      | RET (expr)? #retInst
      ;
```

Listing 4.60: Instruction rules definition

The *visit* method for the *regular instruction* rule (Listing 4.61) returns the result of *visiting* the *child* with *index* zero, corresponding to a *for, if, apply, control* or *conjugated* section.

```
@Override
   public AST visitRegInst(QbricksParser.RegInstContext ctx) {
      return visit(ctx.getChild(0));
   }
```

Listing 4.61: Regular Instruction *visit method* definition

4.3.11 *Return Visit*

The *visit* method for the *return instruction* rule (Listing 4.62)

- creates a *Return Node* "`ret`" and an *Expression Node* "`expr`";
- checks if the parse-tree context contains **two** *children*, in which case the result of *visiting* the "`expr`" section is set on the "`args`" parameter of the "`ret`" *node* and the Boolean "`argBool`" is set to true. In case the parse-tree context does not contain **two** *children*, the "`args`" parameter of the "`ret`" *node* is set to `null` and the Boolean "`argBool`" is set to false;
- returns the "`ret`" *node*.

```
@Override
   public AST visitRetInst(QbricksParser.RetInstContext ctx) {
```

```
      RetNode ret = new RetNode();
      ExpressionNode expr;
      if (ctx.getChildCount()==2) {
         expr = (ExpressionNode) visit(ctx.expr());
         ret.setArgs(expr);
         ret.setArgBool(true);
      } else {
         ret.setArgs(null);
         ret.setArgBool(false);
      }
      return ret;
   }
```

Listing 4.62: Return Instruction *visit method* definition

4.3.12 *For-loop Visit*

The grammar definition for the *for-loop* rule (Listing 4.63) states that a "`for_`" section is defined by a "`FOR`" token followed by an *identifier* "`var`" (the iterator variable), an "`IN`" token, a *for-iteration* section "`iteration`", the *for-loop*'s *invariant* "`inv`" and *body* "`forbody`" sections, both surrounded by *curly-brace* tokens "`COP`" and "`CCL`". *For-iteration* rule (Listing 4.63) states that an "`iter`" section is defined by a "`RANGE`" token followed by either a *term* or a *quantum-register* section surrounded by parenthesis tokens; an "`iter`" section can also be defined by a single *quantum-register* section.

```
for_ :
      FOR var=ID IN iteration=iter
      COP inv=invariant forbody=body CCL #forLoop
      ;

iter :
      RANGE OP (expvalue=term | qrvalue=qReg) CL #rangeIter
      | qReg #qrIter
      ;
```

Listing 4.63: For-loop and iteration rules definition

The *visit* method for the *for-loop* rule (Listing 4.64)

- creates two new *AST* nodes: *For-loop node* "`node`" and *For-iteration node* "`iter`";
- sets the *For-loop node*'s assertion by *visiting* the *parent instruction* section;
- creates an *Invariant node* "`invariant`" and the result of *visiting* the "`inv`" section is assigned to it;
- creates a *Body node* "`body`" and the result of *visiting* the "`forbody`" section is assigned to it;
- creates a variable "`iterable`" and the the "`iteration`" section is assigned to it;

- sets the *For-iteration node*'s *iterator* as the "`var`" *identifier* section;
- checks whether the "`iterable`" variable corresponds to a *range* or a *quantum-register* section; if it is a *range* section, the result of *visiting* the *term* or *quantum-register* section is set as the *iterable* value and the *range* Boolean is set to true on the *for-iteration node*; in case it is a *quantum-register* section, the result of visiting this section is set as the *iterable* value and the *range* Boolean is set to false on the *for-iteration node*;
- sets the values of the *Invariant Node* "`invariant`", the *Body Node* "`body`" and the *For-iteration Node* "`iter`" on the *For-loop node*. Finally, it returns the "*node*".

```
@Override
   public AST visitForLoop(QbricksParser.ForLoopContext ctx) {
      ForNode node = new ForNode();
      ForIter iter = new ForIter();
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
      InvariantNode invariant = (InvariantNode) visit(ctx.inv);
      BodyNode body = (BodyNode) visit(ctx.forbody);
      ParseTree iterable = ctx.iteration;
      iter.setIterator(ctx.var.getText());
      if(iterable.getChildCount() > 1) {
         iter.setIterable(visit(iterable.getChild(2)));
         iter.setRange(Boolean.TRUE);
      } else {
         iter.setIterable(visit(iterable.getChild(0)));
         iter.setRange(Boolean.FALSE);
      }
      node.setInvariant(invariant);
      node.setBody(body);
      node.setIter(iter);
      return node;
   }
```

Listing 4.64: *For-loop visit method* definition

4.3.13 *If-statement Visit*

The grammar definition for the *if-statement* rule (Listing 4.65) states that an "`if_`" section is defined by an "`IF`" token followed by a *conditional expression* "`cond`", a *body* "`ifbody`" section surrounded by *curly-brace* tokens "`COP`" and "`CCL`" and an optional *else* section, defined by an "`ELSE`" token followed by the *else's body* "`elsebody`" surrounded by *curly-brace* tokens.

```
if_ :
      IF cond=expr
      COP ifbody=body CCL
      (ELSE COP elsebody=else_ CCL)? #condIf
      ;

else_:
      body #elsebody
      ;
```

Listing 4.65: If and else rules definition

The *visit* method for the *if-statement* rule (Listing 4.66)

- creates two new *AST* nodes: *If Node* "`node`" and *If-condition Node* "`cond`";
- sets the *If node*'s assertion by *visiting* the *parent instruction* section;
- creates a *Body Node* "`ifBody`" and the result of *visiting* the "`ifbody`" section is assigned to it;
- sets the *if-condition node*'s *expression* as the result of *visiting* the "`cond`" section;
- sets the values of the *If-condition Node* "`cond`" and the *Body Node* "`ifbody`" on the *If node*;
- checks if the "`else`" section exists, in which case the result of *visiting* the "`elsebody`" section is set as the *else's body* value and the *withElse* Boolean is set to true on the *If Node*; in case it does not exist, "`elsebody`" section is set to null and the *withElse* Boolean is set to false on the *If Node*;
- returns the *If Node*.

```
@Override
   public AST visitCondIf(QbricksParser.CondIfContext ctx) {
      IfNode node = new IfNode();
      IfCond cond = new IfCond();
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
      BodyNode ifBody = (BodyNode) visit(ctx.ifbody);
      cond.setExpr((ExpressionNode) visit(ctx.cond));
      node.setCond(cond);
      node.setIfBody(ifBody);
      if(ctx.getChildCount()>5) {
         node.setElseBody((BodyNode) visit(ctx.elsebody));
         node.setWithElse(true);
      } else {
         node.setElseBody(null);
         node.setWithElse(false);
      }
      return node;
   }
```

Listing 4.66: *If-statement visit method* definition

4.3.14 *Apply Visit*

The grammar definition for the *gate application* rule (Listing 4.67) states that an "`apply`" section is defined by a *gate identifier*, followed by the gate's arguments. If the number of

arguments exceeds one they will be surrounded by parenthesis tokens "OP" and "CL" and separated by commas "VG".

```
apply :
      fun=idFun OP (fargs=args)? CL #funApply
      | REVERSE OP fun=idFun OP (fargs=args)? CL CL #revApply
      | HAD OP qr=qReg CL #hadApply
      | XGATE OP qr=qReg CL #xApply
      | YGATE OP qr=qReg CL #yApply
      | ZGATE OP qr=qReg CL #zApply
      | TGATE OP qr=qReg CL #tApply
      | SGATE OP qr=qReg CL #sApply
      | SWAP OP qrL=qReg VG qrR=qReg CL #swapApply
      | RX OP angle=ang VG qr=qReg CL #rxApply
      | RY OP angle=ang VG qr=qReg CL #ryApply
      | RZ OP angle=ang VG qr=qReg CL #rzApply
      | PHASE OP angle=ang VG qr=qReg CL #phApply
      ;

args :
      term (VG term)* #funArgs
      ;

ang :
      term #angTerm
      ;
```

Listing 4.67: Gate application definition

There are *four* different types of *Apply nodes*, depending on the number of the gate's parameters. Each type of *Apply node* corresponds to a *visit* method.

1. *Visit* method for nodes with only **one parameter** (Listing 4.68) creates a *SingleApply node* "node", checks if the gate is being used as a *controlled operation* in order to correctly set the associated *assertion* on the "node" object, creates a *Quantum register node* "qr" and the result of *visiting* the "qr" section from the parse-tree context is assigned to it. Then "qr" is set as the *quantum register* on the "node", after which "node" is returned;

```
@Override
public AST visitSingleApply(QbricksParser.SingleApplyContext ctx) {
   SingleApply node = new SingleApply();
   if(!ctx.getParent().getClass().getSimpleName().equals("ApplyControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   QregNode qr = (QregNode) visit(ctx.qr);
   node.setQreg(qr);
   return node;
}
```

Listing 4.68: Single argument gate application *visit method* definition

2. *Visit* method for nodes with **two parameters** (Listing 4.69) creates a *DoubleApply node* "node", checks if the gate is being used as a *controlled operation* in order to correctly set the associated *assertion* on the "node" object, creates a *Quantum register node* "qr" and the result of *visiting* the "qr" section from the parse-tree context is assigned to it. Creates a *Term node* "angle" and the result of *visiting* the "angle" section from the parse-tree context is assigned to it. Then "qr" is set as the *quantum register* and "angle" is set as the *angle* on the "node", after which the "node" is returned;

```
@Override
public AST visitDoubleApply(QbricksParser.DoubleApplyContext ctx) {
   DoubleApply node = new DoubleApply();
   if(!ctx.getParent().getClass().getSimpleName().equals("ApplyControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   QregNode qr = (QregNode) visit(ctx.qr);
   TermNode angle = (TermNode) visit(ctx.angle);
   node.setQreg(qr);
   node.setAngle(angle);
   return node;
}
```

Listing 4.69: Two argument gate application *visit method* definition

3. *Visit* method for the *Swap gate* node (Listing 4.70) creates a *SwapApply node* "node", checks if the gate is being used as a *controlled operation* in order to correctly set the associated *assertion* on the "node" object, creates two *Quantum register nodes*: "left" and "right" and the result of *visiting* the "qrL" and "qrR" sections from the parse-tree context are respectively assigned to it. Then "left" and "right" are set as the *quantum registers* on the "node", after which the "node" is returned;

```
@Override
public AST visitSwapApply(QbricksParser.SwapApplyContext ctx) {
   SwapApply node = new SwapApply();
   if(!ctx.getParent().getClass().getSimpleName().equals("ApplyControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   QregNode left = (QregNode) visit(ctx.qrL);
   QregNode right = (QregNode) visit(ctx.qrR);
   node.setQregs(left,right);
   return node;
}
```

Listing 4.70: Swap gate application *visit method* definition

4. *Visit* method for *Function* and *reverse-function* application nodes (Listing 4.71) creates a *FunApply node* "`node`", checks if the gate is being used as a *controlled operation* in order to correctly set the associated *assertion* on the "`node`" object, creates a list of *Term nodes*: "`termArgs`" and the result of *visiting* the "`term`" sections from the parse-tree context is added to it. Then the *function*'s identifier "`fun`" and arguments "`termArgs`" are set on the "`node`", after which the "`node`" is returned;

```
@Override
public AST visitFunApply(QbricksParser.FunApplyContext ctx) {
   FunApply node = new FunApply();
   List<TermNode> termArgs = new ArrayList<>();
   ParseTree args = ctx.fargs;
   if(!ctx.getParent().getClass().getSimpleName().equals("ApplyControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   if (ctx.getChildCount()>3) {
      TermNode arg = (TermNode) visit(args.getChild(0));
      termArgs.add(arg);
      if (args.getChildCount() > 1) {
         for (int c = 2; c < args.getChildCount(); c += 2) {
            arg = (TermNode) visit(args.getChild(c));
            termArgs.add(arg);
         }
      }
   }

   node.setFunID(ctx.fun.getText());
   node.setTermArgs(termArgs);

   return node;
}
```

Listing 4.71: Function gate application *visit method* definition

4.3.15 *Controlled gate Visit*

The grammar definition for the *controlled gate application* rule (Listing 4.72) states that a "`control`" section is defined by a *gate identifier – with-controlled* operator, *C-NOT*, *Toffoli* or *Fredkin* – followed by a list of *quantum registers* "`ctlqrs`", separated by commas, and a *control gate* "`ctlgate`", surrounded by parenthesis, in the *with-controlled* case; for the *C-NOT*, *Toffoli* or *Fredkin* gates, the *gate identifier* is followed by the gate's arguments, surrounded by parenthesis and separated by commas.

```
control :
      WITHCTL ctlqrs=id_list OP ctlgate=apply CL #applyControl
      | WITHCTL ctlqrs=id_list OP ctlgate=control CL #multiControl
      | CNOT OP ctlqr=qReg VG tqr=qReg CL #cnotControl
```

```
      | TOFF OP ctl1=qReg VG ctl2=qReg VG tg=qReg CL #toffControl
      | FRED OP ctl1=qReg VG ctl2=qReg VG tg=qReg CL #fredControl
      ;

id_list :
      qReg (VG qReg)* #qrList
      ;
```

Listing 4.72: Controlled gate application definition

There are *five* different *visit* methods for the *controlled gate application* rule:

1. *Visit* methods for *Control nodes* using a *with-controlled* operator (Listing 4.73) create a *with-controlled node* with an associated assertion, and a "`gate`" node consisting of a *gate application node* or a *controlled gate application node*. A list of quantum-register nodes is created by visiting the "`ctlqrs`" section from the parse-tree context, added to the "`ctlNodes`" list, and set as *control registers*, with the "`gate`" node as the control gate. The resulting node is then returned.

```
@Override
public AST visitApplyControl(QbricksParser.ApplyControlContext ctx) {
   WithCtlNode node = new WithCtlNode();
   node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   ApplyNode gate = (ApplyNode) visit(ctx.ctlgate);
   List<QregNode> ctlNodes = new ArrayList<>(); //list of control qubits
   ParseTree ctls = ctx.ctlqrs;
   QregNode qr = (QregNode) visit(ctls.getChild(0));
   ctlNodes.add(qr);
   if (ctls.getChildCount() > 1) {
      for (int c = 2; c < ctls.getChildCount(); c += 2) {
         qr = (QregNode) visit(ctls.getChild(c));
         ctlNodes.add(qr);
      }
   }
   node.setCtlArgs(ctlNodes);
   node.setCtlGate(gate);
   return node;
}

@Override
public AST visitMultiControl(QbricksParser.MultiControlContext ctx) {
   WithCtlNode node = new WithCtlNode();
   node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   CtlNode gate = (CtlNode) visit(ctx.ctlgate);
   List<QregNode> ctlNodes = new ArrayList<>(); //list of control qubits
   ParseTree ctls = ctx.ctlqrs;
   QregNode qr = (QregNode) visit(ctls.getChild(0));
   ctlNodes.add(qr);
   if (ctls.getChildCount() > 1) {
      for (int c = 2; c < ctls.getChildCount(); c += 2) {
         qr = (QregNode) visit(ctls.getChild(c));
         ctlNodes.add(qr);
      }
   }
   node.setCtlArgs(ctlNodes);
   node.setCtlMulti(gate);
```

```
   return node;
}
```

Listing 4.73: With-control *visit method* definition

2. *Visit* methods for *C-NOT* (Listing 4.74), *Toffoli* and *Fredkin nodes* (Listing 4.75) create a "node" respective to the applied *control* operation, checks if the gate is being used as a *controlled operation* in order to correctly set the associated *assertion* on the "node" object; then, *quantum-register nodes* are created and assigned the result of *visiting* both the *control* and *target* sections from the parse-tree context, which are then set on the "node", the "node" is then returned;

```
@Override
public AST visitCnotControl(QbricksParser.CnotControlContext ctx) {
   CnotNode node = new CnotNode();
   if(!ctx.getParent().getClass().getSimpleName().equals("MultiControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   QregNode ctl = (QregNode) visit(ctx.ctlqr);
   QregNode target = (QregNode) visit(ctx.tqr);
   cnot.setQregs(ctl,target);
   return node;
}
```

Listing 4.74: *C-NOT* gate application *visit method* definition

```
@Override
public AST visitFredControl(QbricksParser.FredControlContext ctx) {
   FredNode node = new FredNode();
   if(!ctx.getParent().getClass().getSimpleName().equals("MultiControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
      node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
          (1)));
   QregNode ctl1 = (QregNode) visit(ctx.ctl1);
   QregNode ctl2 = (QregNode) visit(ctx.ctl2);
   QregNode target = (QregNode) visit(ctx.tg);
   node.setQregs(ctl1,ctl2,target);
   return node;
}

@Override
public AST visitToffControl(QbricksParser.ToffControlContext ctx) {
   ToffNode node = new ToffNode();
   if(!ctx.getParent().getClass().getSimpleName().equals("MultiControlContext"))
       {
      node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
   }
   else
```

```
        node.setAssertion((AssertNode) visit(ctx.getParent().getParent().getChild
            (1)));
    QregNode ctl1 = (QregNode) visit(ctx.ctl1);
    QregNode ctl2 = (QregNode) visit(ctx.ctl2);
    QregNode target = (QregNode) visit(ctx.tg);
    node.setQregs(ctl1,ctl2,target);
    return node;
}
```

Listing 4.75: *Fredkin* and *Toffoli* gate application *visit method* definition

4.3.16 *Conjugated-basis Visit*

The grammar definition for the *conjugated-basis* rule (Listing 4.76) states that a "`conjugated`" section is defined by a "`WITHCJG`" token followed by an *gate application instruction* "`applyinst`" section and a *body* "`conjbody`" section surrounded by *curly-brace* tokens "`COP`" and "`CCL`".

```
conjugated :
      WITHCJG OP? applyinst=apply CL? COP conjbody=body CCL
      ;
```

Listing 4.76: *Conjugated-basis* rule definition

The *visit* method for the *conjugated-basis* rule (Listing 4.77)

- creates a *Conjugated node* "`node`";
- creates a *Body Node* "`body`" and the result of *visiting* the "`conjbody`" section is assigned to it;
- creates an *Apply Node* "`apply`" and the result of *visiting* the "`applyinst`" section is assigned to it;
- sets the associated *assertion* on the "`node`" object;
- sets the values of the "`body`" and "`apply`" *nodes* on the "`node`";
- returns the "`node`".

```
@Override
public AST visitConjugated(QbricksParser.ConjugatedContext ctx) {
    ConjNode node = new ConjNode();
    BodyNode body = (BodyNode) visit(ctx.conjbody);
    ApplyNode apply = (ApplyNode) visit(ctx.applyinst);
    node.setAssertion((AssertNode) visit(ctx.getParent().getChild(1)));
    node.setBody(body);
    node.setApply(apply);
    return node;
}
```

Listing 4.77: *Conjugated-basis visit method* definition

4.3.17 *Expression and Term Visit*

The grammar definition for the *expression* and *term* rules (Listing 4.78) states that an "`expr`" section is defined by a *term* – which can represent an arithmetic expression, a *unary* term or an *atom* – or by a logical expression. Both *expressions* and *terms* can be surrounded by parenthesis.

```
expr :
      term #termExpr
      | left=expr op=EQ right=expr #eqExpr
      | left=expr op=GT right=expr #gtExpr
      | left=expr op=LT right=expr #ltExpr
      | left=expr op=GEQ right=expr #geqExpr
      | left=expr op=LEQ right=expr #leqExpr
      | left=expr op=NEQ right=expr #neqExpr
      | OP expr CL #parenExpr
      ;
term :
      atom #atomTerm
      | left=term op=POW right=term #powTerm
      | left=term op=MUL right=term #mulTerm
      | left=term op=DIV right=term #divTerm
      | left=term op=PLUS right=term #addTerm
      | left=term op=MINUS right=term #subTerm
      | OP term CL #parenTerm
      | unOp #unaryTerm
      ;

atom :
      value = NUM #numAtom
      | pi = PI #piAtom
      | var = ID #varAtom
      ;

unOp :
      MINUS term #negUnary
      | LEN OP qReg CL #lenUnary
      | SQRT OP value=term CL #sqrtUnary
      ;
```

Listing 4.78: Expressions definition

- *Visit* method for a *term expression* (Listing 4.79) returns the result of *visiting* the "`term`" section from the parse-tree context;

```
@Override
public AST visitTermExpr(QbricksParser.TermExprContext ctx) {
   return visit(ctx.term());
}
```

Listing 4.79: *Term* expression *visit method* definition

- *Visit* methods for *logical expressions* (Listing 4.80) are similar: the respective *node* is created, *left* and *right* expressions are set on the *node* by *visiting* "`left`" and "`right`" sections from the parse-tree context, and the *node* is returned;

```
@Override
public AST visitEqExpr(QbricksParser.EqExprContext ctx) {
   InfixExpressionNode node = new EqualNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}

@Override
public AST visitNeqExpr(QbricksParser.NeqExprContext ctx) {
   InfixExpressionNode node = new NEqualNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}

@Override
public AST visitGeqExpr(QbricksParser.GeqExprContext ctx) {
   InfixExpressionNode node = new GEQNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}

@Override
public AST visitLeqExpr(QbricksParser.LeqExprContext ctx) {
   InfixExpressionNode node = new LEQNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}

@Override
public AST visitGtExpr(QbricksParser.GtExprContext ctx) {
   InfixExpressionNode node = new GTNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}

@Override
public AST visitLtExpr(QbricksParser.LtExprContext ctx) {
   InfixExpressionNode node = new LTNode();
   node.setLeft((ExpressionNode) visit(ctx.left));
   node.setRight((ExpressionNode) visit(ctx.right));
   return node;
}
```

Listing 4.80: Logical expression *visit methods* definition

- *Visit* method for a *parenthesis expression* (Listing 4.81) returns the result of *visiting* the "`expr`" section from the parse-tree context;

```
@Override
public AST visitParenExpr(QbricksParser.ParenExprContext ctx) {
   return visit(ctx.expr());
}
```

Listing 4.81: *Parenthesis* expression *visit method* definition

- *Visit* method for an *atom term* (Listing 4.82) returns the result of *visiting* the "`atom`" section from the parse-tree context;

```
@Override
public AST visitAtomTerm(QbricksParser.AtomTermContext ctx) {
   return visit(ctx.atom());
}
```

Listing 4.82: *Atom* term *visit method* definition

- *Visit* methods for *arithmetic expressions* (Listing 4.83) are similar: the respective *node* is created, *left* and *right* terms are set on the *node* by *visiting* "`left`" and "`right`" sections from the parse-tree context. The *node* is then returned;

```
@Override
public AST visitPowTerm(QbricksParser.PowTermContext ctx) {
   InfixTermNode node = new PowerNode();
   node.setLeft((TermNode) visit(ctx.left));
   node.setRight((TermNode) visit(ctx.right));
   return node;
}

@Override
public AST visitMulTerm(QbricksParser.MulTermContext ctx) {
   InfixTermNode node = new MulNode();
   node.setLeft((TermNode) visit(ctx.left));
   node.setRight((TermNode) visit(ctx.right));
   return node;
}

@Override
public AST visitDivTerm(QbricksParser.DivTermContext ctx) {
   InfixTermNode node = new DivNode();
   node.setLeft((TermNode) visit(ctx.left));
   node.setRight((TermNode) visit(ctx.right));
   return node;
}

@Override
public AST visitAddTerm(QbricksParser.AddTermContext ctx) {
   InfixTermNode node = new AddNode();
   node.setLeft((TermNode) visit(ctx.left));
   node.setRight((TermNode) visit(ctx.right));
   return node;
}

@Override
public AST visitSubTerm(QbricksParser.SubTermContext ctx) {
```

```
   InfixTermNode node = new SubNode();
   node.setLeft((TermNode) visit(ctx.left));
   node.setRight((TermNode) visit(ctx.right));
   return node;
}
```

Listing 4.83: Arithmetic expression *visit methods* definition

- *Visit* method for a *parenthesis term* (Listing 4.84) creates a *Parenthesis node*, sets the result of *visiting* the "`term`" section from the parse-tree context on the *node* and returns the *node*;

```
@Override
public AST visitParenTerm(QbricksParser.ParenTermContext ctx) {
   ParenNode node = new ParenNode();
   node.setTerm((TermNode) visit(ctx.term()));
   return node;
}
```

Listing 4.84: *Parenthesis* term *visit method* definition

- *Visit* method for a *unary term* (Listing 4.85) returns the result of *visiting* the "`unOp`" section from the parse-tree context;

```
@Override
public AST visitUnaryTerm(QbricksParser.UnaryTermContext ctx) {
   return visit(ctx.unOp());
}
```

Listing 4.85: *Parenthesis* expression *visit method* definition

- *Visit* methods for *negative* and *square-root unary terms* (Listing 4.86) create a *unary node* "`node`", set the respective *operator* and *inner term* on the *node*, and the *node* is returned;

```
@Override
public AST visitNegUnary(QbricksParser.NegUnaryContext ctx) {
   UnOpNode node = new UnOpNode();
   node.setOp("Minus ");
   node.setInnerTerm((TermNode) visit(ctx.term()));
   return node;
}

@Override
public AST visitSqrtUnary(QbricksParser.SqrtUnaryContext ctx) {
   UnOpNode node = new UnOpNode();
   node.setOp(ctx.SQRT().getText());
   node.setInnerTerm((TermNode) visit(ctx.term()));
   return node;
}
```

Listing 4.86: *Negative* and *square-root unary* terms *visit method* definition

- *Visit* method for a *length unary term* (Listing 4.87) creates a *length node* "`node`", sets the respective *quantum register* on the *node* by *visiting* the "`qReg`" section from the parse-tree context, and the *node* is returned;

```
@Override
public AST visitLenUnary(QbricksParser.LenUnaryContext ctx) {
   LenNode node = new LenNode();
   node.setQrTerm((QregNode) visit(ctx.qReg()));
   return node;
}
```

Listing 4.87: *Length unary* term *visit method* definition

- *Visit* method for an *atom term* (Listing 4.88) returns the result of *visiting* the "`atom`" section from the parse-tree context;

```
@Override
public AST visitAtomTerm(QbricksParser.AtomTermContext ctx) {
   return visit(ctx.atom());
}
```

Listing 4.88: *Atom* term *visit method* definition

- *Visit* methods for *number*, *pi* and *variable atoms* (Listing 4.89) create an *atom node* "`node`", set the respective *value* and *type* on the *node*, and the *node* is returned;

```
@Override
public AST visitNumAtom(QbricksParser.NumAtomContext ctx) {
   AtomNode node = new AtomNode();
   node.setValue("Num "+ctx.value.getText());
   node.setType("Num");
   return node;
}

@Override
public AST visitPiAtom(QbricksParser.PiAtomContext ctx) {
   AtomNode node = new AtomNode();
   node.setValue("Num pi");
   node.setType("Num");
   return node;
}

@Override
public AST visitVarAtom(QbricksParser.VarAtomContext ctx) {
   AtomNode node = new AtomNode();
   node.setValue("Var \""+ctx.var.getText()+"\"");
   node.setType("Var");
   return node;
}
```

Listing 4.89: *Number*, *pi* and *variable* atom terms *visit method* definition

4.4 CHANGING THE PARADIGM

Once we extract the relevant information from the *concrete-syntax tree* and build the *Java-AST*, we have a nicely organized data-structure that represents a program in iQbricks. By traversing the *Java-AST*, information stored in *nodes* can be easily extracted in order to either

1. translate it directly to a *target language* or
2. generate a different *abstract structure*, which can enable a more *fine-grained* representation of a program for later translating into the *target language*.

The implemented solution for this project follows the second approach. Since the *target language* (Qbricks) is a *functional* one, directly translating the *Java-AST* to a *functional paradigm* would result in a much more complicated and possibly unfeasible translation process, which could limit iQbricks' potential as a *quantum programming language*. A *functional* representation of a program's structure simplifies the translation process and provides a more **solid** and **encapsulated** implementation for iQbricks framework. Taking this into account, *Java-AST Evaluator* generates an *AST* in Ocaml language from the previously built *Java-AST*. The implementation for the *ML-AST* structure and *Java-AST Evaluator* will be covered in Sections 4.5 and 4.6, respectively.

4.5 Ocaml AST structure

Following the same process used to define the *Java-AST*, a structure that represents a program's information, *ML-AST*, was defined. The new *AST* was defined using *Ocaml*'s *records* feature, which is a concise and expressive system for declaring new data types. A *record* represents a collection of values stored together as one, where each component is identified by a different field name. The basic syntax for a record type declaration is shown in Listing 4.90.

```
type <record-name> =
   {
      <field> : <type>;
      <field> : <type>;
      ...
   }
```

Listing 4.90: Basic *record* syntax

4.5.1 *Program type*

A *program type* consists in:

- a string identifier "`id`",
- a list of strings "`imports`",
- a "`main`" function, defined by a *function type*,
- a list of *auxiliary* functions, defined by *function types*.

A *program type* has the structure expressed in Listing 4.91.

```
type program =
   {
       id: string;
       imports: string list;
       main: fun_;
       aux: fun_ list;
   }
```

Listing 4.91: *Program type* definition

4.5.2 *Function type*

A *function type* consists in:

- a string identifier "`id`",
- a *circuit type* "`circ`",
- a list of *parameter types* "`params`",
- a list of *pre and post-condition types* "`pre`" and "`pos`", respectively.

A *function type* has the structure expressed in Listing 4.92.

```
type fun_ =
   {
      id: string;
      circ: circ;
      params: param list;
      pre: string list;
      pos: string list
   }
```

Listing 4.92: *Function type* definition

4.5.3 *Parameter type*

A *parameter type* consists in:

- a string identifier "`id`",
- a *type* "`type_`" [3].

Parameter and "*type*" *type* structures are expressed in Listing 4.93.

3 The "`type_`" type is a *variant* type, which means it can take on one of several different forms, represented by the `"|"` symbol. In this case, the "`type_`" type can be one of four possible values: "`Qreg`", "`Circ`", "`Int`" or "`Bool`". These values represent the different types in IQBRICKS.

```
type type_ = | Qreg | Circ | Int | Bool ;;

type param =
   {
      id: string;
      type_: type_
   }
```

Listing 4.93: *Parameter* and "*type*" *type* definition

4.5.4 *Circuit type*

A *circuit type* consists in:

- a list of *quantum register types* "`qregs`", which have two fields: a string identifier "`qrid`" and an *expression type* "`size`",
- a list of *instruction types* "`body`".

Circuit and *quantum register types* structures are expressed in Listing 4.94.

```
type circ =
   {
      qregs: qreg list;
      body: instruction list
   }

type qreg =
   {
      qrid: string;
      size: expr;
   }
```

Listing 4.94: *Circuit* and *quantum register types* definition

4.5.5 *Instruction type*

An *instruction type* is a variant type and can take different forms. Each variant has a record type associated with it, which contains different fields:

- "`Conjugated`" variant has three fields: a *unitary type* "`gate`", a list of *instruction types* "`body`" and a list of strings "`assertion`",
- "`For`" variant has five fields: an *iteration type* "`iter`", a string identifier "`qr`" (if a *quantum register* is being iterated it consists in its identifier) a list of strings "`inv`" (the loop's invariant), a list of *instruction types* "`body`" and a list of strings "`assertion`",

- "`If`" variant has three fields: a *condition type* "`cond`", a list of *instruction types* "`body`" and a list of strings "`assertion`",
- "`IfElse`" variant has the same fields as the "`If`" variant, with the addition of a list of *instruction types* "`elsebody`" for the *else*'s body,
- "`Unitary`" variant has a single field: a *unitary type*,
- "`Return`" variant has a single field, defined by a string.

Instruction type's structure is expressed in Listing 4.95.

```
type instruction =
   | Conjugated of {gate:unitary; body:instruction list; assertion: string list}
   | For of {iter:iter; qr:string; inv: string list; body:instruction list; assertion:
       string list}
   | If of {cond: cond; body:instruction list ; assertion: string list}
   | IfElse of {cond:cond; ifbody:instruction list; elsebody:instruction list;
      assertion:string list}
   | Unitary of unitary
   | Return of string ;;
```

Listing 4.95: *Instruction type* definition

4.5.6 *Iteration type*

An *iteration type* consists in:

- a string identifier "`iterator`",
- *expression types* "`starts`" and "`ends`".

Iteration type's structure is expressed in Listing 4.96.

```
type iter =
   {
      iterator: string;
      starts: expr;
      ends: expr;
   }
```

Listing 4.96: *Iteration type* definition

4.5.7 *Unitary type*

A *unitary type* is a variant type and each different variant form has a record type associated with it, which contains different fields:

- "`Sequence`" variant has two *unitary type* fields,

- "`Apply`" variant has four fields: a *gate type* "`gate`", a string identifier "`qreg`", a *range type* "`range`" and a list of strings "`assertion`",
- "`MultiApply`" variant has three fields: a *multigate type* "`gate`", a list of *iter types* "`regs`", and a list of strings "`assertion`",
- "`WithControl`" variant has four fields: a *unitary type* "`ctlgate`", a list of *iter types* "`ctls`", an *iter type* "`tg`" and a list of strings "`assertion`",
- "`FUN`" and "`REV`" variants have two fields: a string identifier "`id`" and a list of *expression types* "`args`".

Unitary type's structure is expressed in Listing 4.97.

```
type unitary =
   | Sequence of unitary * unitary
   | Apply of {gate:gate; qreg:string; range:range; assertion:string list}
   | MultiApply of {gate:multigate; regs:iter list; assertion:string list}
   | WithControl of {ctlgate:unitary; ctls:iter list; tg:iter; assertion:string list}
   | FUN of {id:string; args: expr list}
   | REV of {id:string; args: expr list} ;;
```

Listing 4.97: *Unitary type* definition

4.5.8 *Range type*

A *range type* consists in two *expression types*: "`starts`" and "`ends`". Its structure is expressed in Listing 4.98.

```
type range =
   {
      starts: expr;
      ends: expr;
   }
```

Listing 4.98: *Range type* definition

4.5.9 *Gate and Multi-gate types*

A *gate type* is a variant type that represents the different single-qubit gates in iQbricks. The possible variants are: $H, X, Y, Z, T, S, R_x(\theta), R_y(\theta), R_z(\theta)$ and $Ph(\theta)$. In rotation gates $R_x(\theta), R_y(\theta), R_z(\theta)$ and $Ph(\theta)$, the angle θ is specified by an *expression type*. A *multi-gate type* is another variant type that represents multi-qubit gates in iQbricks. The possible variants are: *C-NOT*, *Toffoli*, *Fredkin* and *Swap*.

Gate and Multi-gate type structures are expressed in Listing 4.99.

```
type gate =
   | H | X | Y | Z | T | S
   | Rx of expr
   | Ry of expr
   | Rz of expr
   | Ph of expr ;;

type multigate =
   | Cnot | Toff | Fred | SWAP ;;
```

Listing 4.99: *Gate and Multi-gate types* definition

4.5.10 *Condition and Expression types*

A *condition type* defines the different comparison operations between two expressions: *equal to*, *not equal to*, *greater than*, *less than*, *greater than* or *equal to*, and *less than* or *equal to*. An *expression type* defines the different mathematical expressions: *addition*, *subtraction*, *multiplication*, *division* and *power*. It also includes unary operators *minus*, *length* and *square-root* and atoms for *numbers* and *variables*.

Condition and Expression type structures are expressed in Listing 4.100.

```
type expr =
   | Plus of expr * expr
   | Subtract of expr * expr
   | Times of expr * expr
   | Divide of expr * expr
   | Power of expr * expr
   | Minus of expr
   | Len of string
   | Sqrt of expr
   | Num of int
   | Var of string ;;

type cond =
   | Eq of expr * expr
   | NEq of expr * expr
   | Gt of expr * expr
   | Lt of expr * expr
   | GEq of expr * expr
   | LEq of expr * expr ;;
```

Listing 4.100: *Condition* and *expression types* definition

4.6 EVALUATING THE JAVA-AST

Once the *Java-AST* is defined, it is necessary to build an *AST Visitor* class to traverse it. This class contains the *Visit methods* that are implemented in the *Java-AST Evaluator*. For *nodes* that are extended by other *nodes* – *Instruction*, *Control*, *Apply*, *Term* and *Expression* – it is necessary

to check which instance of the *node* is being *visited*. This is done by using a sequence of *if-statements* that verify all possible instances of the *node*. Listing 4.101 is an abridged version of the implemented *AST Visitor*, the full code is detailed in Carneiro (2023).

```
public abstract class MyASTVisitor <T> {
   public abstract T Visit(ProgramNode node);
   public abstract T Visit(ImportsNode node);
   public abstract T Visit(MainNode node);
   public abstract T Visit(AuxNode node);
   // Visit methods for remaining rules
   public T Visit(InstrNode node) {
      if (node instanceof ForNode)
         return Visit((ForNode) node);
      else if (node instanceof IfNode)
         return Visit((IfNode) node);
      ...
   public T Visit(CtlNode node) {
      if (node instanceof WithCtlNode)
         return Visit((WithCtlNode) node);
      else if (node instanceof CnotNode)
         return Visit((CnotNode) node);
      ...
   public T Visit(ApplyNode node) {
      if (node instanceof FunApply)
         return Visit((FunApply) node);
      else if (node instanceof RevApply)
         return Visit((RevApply) node);
      else if (node instanceof HadApply)
         return Visit((HadApply) node);
      else if (node instanceof RxApply)
      ...
   public T Visit(TermNode node) {
     if (node instanceof PowerNode)
        return Visit((PowerNode) node);
     else if (node instanceof MulNode)
        return Visit((MulNode) node);
     else if (node instanceof DivNode)
     ...
   public T Visit(ExpressionNode node) {
      if (node instanceof EqualNode)
         return Visit((EqualNode) node);
      else if (node instanceof GTNode)
         return Visit((GTNode) node);
      else if (node instanceof LTNode)
      ...
```

Listing 4.101: *AST Visitor* definition

4.6.1 *Java-AST Evaluator*

The *Java-AST Evaluator* is responsible for processing the *Java-AST* tree structure by implementing the necessary *visit methods*. The purpose of this *Evaluator* is to convert the *Java-AST* into an *ML-AST* representation. The *Evaluator* does this by walking through the nodes in the *Java-AST* and generating a text file that represents the *ML-AST* using the defined data types.

An abridged example of a program written in *Ocaml* which represents an *ML-AST* structure is shown in Listing 4.102, where a program with *id* "`grover_cz`", an auxiliary function "`oracle`", *import files* and the *main* function are expressed. The program begins by *opening* "`ML_AST`" and "`ML_eval`" modules, which correspond to the *ML-AST* structure definition and the *ML-AST Evaluator*, respectively. Then a *program type* structure is defined.

```
open ML_AST
open ML_eval

let p = {
   id = "grover_cz";
   aux = [
      {
      id = "oracle";
      circ = {
         qregs= [{qrid="qr"; size=Num 0}];
         body = [
         ... ]
      }
      }
   ];
   imports = ["grover"; ];
   main = {
      ...
   }
};;
```

Listing 4.102: *ML-AST* structure example

4.6.2 *Program Node Evaluation*

The *Visit method* for the *Program node* (Listing 4.103):

- starts by declaring a *StringBuilder* variable "`t`", which contains the generated code that is later written to a file;
- declares lists "`auxs`", "`undeclared_funs`" "`declared_funs`" and "`index`";
- creates a *string* "`main`" and the result of *visiting* the *Main node* is assigned to it;
- appends the initial portion of the *Ocaml* program: the file imports and declaration of the *program* "`p`" and its *identifier*;
- handles the case of *auxiliary functions* in the *program*: if there are *auxiliary functions* in the *program*, the results of visiting the *node*'s auxiliary list are stored in the list "`auxs`" and the *identifier* of each *auxiliary function* is added to the list "`declared_funs`". If the number of *auxiliary functions* in the *node*'s auxiliary list is smaller than the

number of auxiliary functions in the globally declared list "auxIds" [4] it iterates over "auxIds" to find any auxiliary functions that have not been declared. These undeclared functions are added to the "undeclared_funs" list. Finally, these undeclared functions are removed from "auxIds" which stores the names of all declared functions in a *program*, in their order of appearance.

- appends *auxiliary functions* to the variable "t" in their correct order [5];
- in case *functions* are received as arguments by either the *main* or *auxiliary* functions, removes them from the "undeclared_funs" list;
- appends files which need to be imported – either *imports* declared at the beginning of a program or functions in the "undeclared_funs" list – to "t";
- appends the *string* "main" to "t";
- finally, appends a function – which applies the *ML-eval* "run_program" function to the generated program "p" and outputs the result to a file – to "t", after which the *string* value of "t" is returned.

```
public List<String> auxIds = new ArrayList<>();

public String Visit(ProgramNode node) {
   StringBuilder t = new StringBuilder();
   List<String> auxs = new ArrayList<>();
   List<String> undeclared_funs = new ArrayList<>();
   List<String> declared_funs = new ArrayList<>();
   List<Integer> index = new ArrayList<>();
   String main = Visit(node.getMain());
   t.append("""
         open ML_AST
         open ML_eval

         let p = {
            id = \""").append(node.getMain().getID()).append("\";\n");
   if (!node.getAuxList().isEmpty()) {
      for (AuxNode c : node.getAuxList()) {
         auxs.add(Visit(c));
         declared_funs.add(c.getID());
      }
      if (node.getAuxList().size() < auxIds.size()){
         for (String s : auxIds) {
            if (!declared_funs.contains(s))
               undeclared_funs.add(s);
         }
         for (String s: undeclared_funs) {
            auxIds.remove(s);
```

4 The globally declared list "auxIds" stores the names of auxiliary functions that are either *declared* or *called* throughout a *program* in their order of appearance.

5 *Ocaml* programs require functions to be declared *before* they are *called*, hence the need to input them in the correct order.

```
            }
        }
        t.append("aux = [");

        for (AuxNode c : node.getAuxList())
            index.add(auxIds.indexOf(c.getID()));
        for (int n, i = 0; i < index.size(); ++i) {
            n = index.indexOf(i);
            t.append("\n").append(auxs.get(n));
        }
        t.append("\n];\n");
    } else {
        undeclared_funs = auxIds;
        t.append("aux = [];\n");
    }
    if (node.getMain().getHasParams()) {
        for (SingleParam p : node.getMain().getParams().getPs()) {
            undeclared_funs.remove(p.getId());
        }
    }
    for (AuxNode n: node.getAuxList()){
        if (n.getHasParams()) {
            for (SingleParam p : n.getParams().getPs()) {
                undeclared_funs.remove(p.getId());
            }
        }
    }
    t.append("imports = [");
    if(node.getImports().hasImports)
        t.append(Visit(node.getImports()));
    for (String s: undeclared_funs) {
        t.append("\"").append(s).append("\"; ");
    }
    t.append("];\n");
    t.append(main);
    t.append("""
        let () =
            let run = run_program p in
            let oc = open_out (p.id ^ ".mlw") in
              Printf.fprintf oc "%s" run;
              close_out oc;
              print_endline "translation generated successfully";""");
    return t.toString();
}
```

Listing 4.103: *Program node* Visit method

4.6.3 *Imports Node Evaluation*

The *Visit method* for the *Imports node* (Listing 4.104):

- starts by declaring a *StringBuilder* variable "`s`";
- loops through the *node*'s *import files*, appending them to "`s`";

- returns the *string* value of "s".

```
public String Visit(ImportsNode node) {
   StringBuilder s = new StringBuilder();
   for (String i: node.getFiles()) {
      s.append("\"").append(i).append("\"; ");
   }
   return s.toString();
}
```

Listing 4.104: *Imports node* Visit method

4.6.4 *Main Node Evaluation*

The *Visit method* for the *Main node* (Listing 4.105) first extracts information from the *Main node* and creates a string representation of it in the *ML-AST* format. Specifically, it extracts the "ID" of the *Main node*, the result of *visiting* its *Circuit node* object, and parameters associated with the *Main node*. It then concatenates this information into string variable "r", along with the results of *visiting* the *pre* and *post-condition nodes* of the *Main node*. The method then returns the string "r".

```
public String Visit(MainNode node) {
      String r = "main = {\n id = \"" + node.getID() + "\";\n" +
            "circ = {\n" + Visit(node.getCirc()) + "}; \n"
            + "params = [";
      if (node.hasParams)
         r += Visit(node.getParams());
      r += "]; \n" + Visit(node.getPre()) + Visit(node.getPos()) +
            "}};;\n";
      return r;
   }
```

Listing 4.105: *Main node* Visit method

4.6.5 *Auxiliary Node Evaluation*

The *Visit method* for the *Auxiliary node* (Listing 4.106) has a similar structure to the *visit method* for the *Main node*. It first adds the auxiliary function's "ID" to the global variable "auxIds" list, if it has not been added yet. It then constructs and returns the string representation of the *Auxiliary node*: the "ID" of the auxiliary function, the result of *visiting* its *Circuit node* and any parameters associated to the *node*, as well as the results of *visiting* the *pre* and *post-condition nodes*.

```
public String Visit(AuxNode node) {
   if (!auxIds.contains(node.getID())) auxIds.add(node.getID());
   String r = "{\nid = \"" + node.getID() + "\";\n" +
         "circ = {\n" + Visit(node.getCirc()) + "};\n"
```

```
        + "params = [";
   if (node.hasParams)
      r += Visit(node.getParams());
   r += "]; \n" + Visit(node.getPre()) + Visit(node.getPos()) +
        "}; \n";
   return r;
}
```

Listing 4.106: *Auxiliary node* Visit method

4.6.6 *Parameters Node Evaluation*

The *Visit method* for the *Parameters node* (Listing 4.107) iterates through the list of *parameters* (in case they exist) and for each one it adds their "`id`" and "`type`" to a *string* "`r`", after which "`r`" is returned.

```
public String Visit(ParamsNode node) {
   StringBuilder r = new StringBuilder();
   if (node.getPs()!=null) {
      for (SingleParam c : node.getPs()) {
         r.append("{id=\"").append(c.getId()).append("\"; ");
         r.append(" type_=").append(c.getType()).append("}; ");
      }
   }
   return r.toString();
}
```

Listing 4.107: *Parameters node* Visit method

4.6.7 *Pre, Post-condition and Assertion Nodes Evaluation*

The *Visit methods* for *Pre, Post-condition, Assertion and Invariant nodes* (Listing 4.108) have similar structures: after declaring a *StringBuilder* variable "`r`", the *node*'s list of *conditions* is iterated and each one of its elements is added to "`r`", after which "`r`" is returned. *Visit methods* for *Assertion* and *Invariant nodes* also check if the *node* is not null, since it is not mandatory for it to be specified after an instruction.

```
public String Visit(PreNode node) {
   StringBuilder r = new StringBuilder();
   r.append("pre = [");
   for (String c: node.get()){
      r.append("\"").append(c).append("\"; ");
   }
   return r+"];\n";
}

public String Visit(PosNode node) {
   StringBuilder r = new StringBuilder();
   r.append("pos = [");
   for (String c: node.get()){
```

```
      r.append("\"").append(c).append("\"; ");
   }
   return r+"];\n";
}

public String Visit(AssertNode node) {
   StringBuilder r = new StringBuilder();
   r.append("assertion=[");
   if (node!=null) {
      for (String c : node.getAssertions())
         r.append("\"").append(c).append("\"; ");
   }
   return r+"]\n";
}

public String Visit(InvariantNode node) {
   StringBuilder r = new StringBuilder("inv = [");
   if (node!=null) {
      for (String c : node.get())
         r.append("\"").append(c).append("\"; ");
   }
   return r+"];\n";
}
```

Listing 4.108: *Pre* and *Post-condition, Assertion and Invariant nodes* Visit methods

4.6.8 *Circuit Node Evaluation*

The *Visit method* for the *Circuit node* (Listing 4.109) starts by declaring the circuit's *quantum registers* and *visiting* the *circuit registers node* "`Ids`". Then the result of *visiting* the *body node* is added to the *string* "`r`", which is returned.

```
public String Visit(CircNode node) {
   String r;
   r = "qregs= ["+Visit(node.getIds())+"];\n"+
      Visit(node.getBody());
   return r;
}
```

Listing 4.109: *Circuit node* Visit method

4.6.9 *Circuit registers Node Evaluation*

The *Visit method* for the *Circuit registers node* (Listing 4.110):

- starts by creating a new *StringBuilder* "`qregs`" which will be used to build the *string* representation of the *circuit registers*;
- retrieves the first *quantum register node* from the input node and extracts its "`ID`". If this *quantum register* has a *range* specified (which refers to the *register*'s size), the size

is retrieved by *visiting* the *range node*. Otherwise, the size is assumed to be 0; The *register*'s identifier and size are then added to "`qregs`";

- clears the "`global_qrs`" list, which is used to keep track of all the *quantum registers* in the circuit, and adds the first *quantum register*'s "`ID`" to it;
- if there are more than one *quantum register nodes* in the input *node*'s list, the method loops over the rest of the *registers*. For each *quantum register node*, the identifier and size (if applicable) are extracted in the same way as before and added to "`qregs`". The identifier is also added to the "`global_qrs`" list;
- returns the *string* value of the variable "`qregs`";

```
public String Visit(CircIds node) {
   StringBuilder qregs = new StringBuilder();
   String size, qr = node.getRegs().get(0).getId();
   if (node.getRegs().get(0).hasRange())
      size = Visit(node.getRegs().get(0).getRange());
   else size = "Num 0";
   qregs.append("{qrid=\"").append(qr).append("\"; ");
   qregs.append("size=").append(size).append("}");
   global_qrs.clear();
   global_qrs.add(qr);
   if (node.getRegs().size()>1) {
      for (QregNode c : node.getRegs().subList(1, node.getRegs().size())) {
         qr = c.getId();
         qregs.append("; {qrid=\"").append(qr).append("\"; ");
         if (c.hasRange())
            size = Visit(c.getRange());
         else size = "Num 0";
         qregs.append("size=").append(size).append("}");
         global_qrs.add(qr);
      }
   }
   return qregs.toString();
}
```

Listing 4.110: *Circuit registers node* Visit method

4.6.10 *Quantum register and Range Nodes Evaluation*

The *Visit method* for the *quantum register node* returns either the *register*'s "`ID`" if the range *Boolean* is false, or the result of *visiting* the *range node* if the range *Boolean* is true.

```
public String Visit(QregNode node) {
   if (!node.hasRange())
      return node.getId();
   else return Visit(node.getRange());
}
```

Listing 4.111: *Quantum register node* Visit method

The *Visit method* for the *range node* (Listing 4.112) handles the different representations for *range nodes* (as covered in Subsection 4.3.7):

- it starts by extracting the input *node*'s identifier, as well as the "`start`" and "`end`" *term nodes*;
- if the *start term* is *null*, *i.e.* not specified (*'up-to' node*), *string* variable "`s`" is set to "`Num 0`" (the number *zero*). The result of *visiting* the "`end`" *term node* is assigned to *string* variable "`e`". If "`e`" is equal to "`Minus (Num 1)`", this refers to the case where we wish to iterate *up-to* the *last index* (not-inclusive) of the *register*, so "`e`" is set to the *subtraction* of the *register*'s length by the number 2 [6]. If not, "`e`" is set to the *subtraction* of the value of "`e`" by the number 1;
- if the *end term* is *null* (*'from' node*), this refers to the case where we wish to iterate from the *start term up-to* the *last index* (inclusive) of the *register*. Thus the result of *visiting* the "`start`" *term node* is assigned to "`s`" and "`e`" "`e`" is set to the *subtraction* of the the *register*'s length by the number 1;
- if the *start* and *end terms* are the same (*term range node*), the result of *visiting* the "`start`" *term node* is assigned to "`s`", and if "`s`" is equal to "`Minus (Num 1)`", this refers to the case where we wish to refer to the *last index* of the *register*, thus "`s`" is set to the *subtraction* of the the *register*'s length by the number 1;
- if instead the *range node* is an *interval node*, the result of *visiting* the "`start`" and "`end`" *term nodes* is assigned to "`s`" and "`e`", respectively. If "`e`" is equal to "`Minus (Num 1)`", this refers to the case where we wish to iterate *up-to* the *last index* (not-inclusive) of the *register*, so "`e`" is set to the *subtraction* of the *register*'s length by the number 2. If not, "`e`" is set to the *subtraction* of the value of "`e`" by the number 1;

```
public String Visit(RangeNode node) {
   String r, s, e, id=node.getIterator();
   TermNode start = node.getStart();
   TermNode end = node.getEnd();
   if (start==null){
      s = "Num 0";
      e = Visit(end);
      if (e.equals("Minus (Num 1)")) {
         e = "Subtract (Len \""+id+"\", Num 2)";
      }
      else e = "Subtract ("+e+", Num 1)";
      r = s + ";" + e;
   } else if (end==null) {
      s = Visit(start);
      e = "Subtract (Len \""+id+"\", Num 1)";
      r = s + ";" + e;
```

6 The size of a register is given by its length subtracted by 1, since the first index is considered to be 0.

```
   } else if (start.equals(end)){
      s = Visit(start);
      if (s.equals("Minus (Num 1)")) {
         s = "Subtract (Len \""+id+"\", Num 1)";
      }
      r = s;
   } else {
      s = Visit(start);
      e = Visit(end);
      if (e.equals("Minus (Num 1)")) {
         e = "Subtract (Len \""+id+"\", Num 2)";
      }
      else e = "Subtract ("+e+", Num 1)";
      r = s + ";" + e;
   }
   return r;
}
```

Listing 4.112: *Range node* Visit method

4.6.11 *Body Node Evaluation*

The *Visit method* for the *body node* (Listing 4.113):

- starts by declaring a *StringBuilder* "r" which will contain the *string* representation of the *body*;
- clears the global list "unitaries" [7];
- adds the result of *visiting* each *instruction node* in the *input node* to "r";
- in case the *body* contains only a sequence of *gate application* instructions, the function "printUnitaries" (Listing 4.113) is used to add the sequential composition of these gates in the *ML-AST* format to "r";
- returns the *string* value of "r".

```
public String Visit(BodyNode node) {
   StringBuilder r = new StringBuilder("body = [\n");
   unitaries.clear();
   for(InstrNode c:node.getBodyInstr()){
      r.append(Visit(c));
   }
   if (r.toString().equals("body = [\n"))
      r.append(printUnitaries(unitaries)).append("];\n");
   else r.append("];\n");
   return r.toString();
}
```

7 This list is used by *visit methods* for *instruction nodes*, in order to correctly extract the circuit's gate compositions and express them using the *ML-AST* format

```
public String printUnitaries (List<String> l) {
   String old, s="";
   if (!l.isEmpty()) {
      if (l.size() < 2)
         s = "Unitary ("+l.get(0)+");\n";
      else {
         s = "Sequence ("+l.get(1)+","+l.get(0)+")";
         for (int i = 2; l.size()>i; i++){
            old = s;
            s = "Sequence (" + l.get(i) + "," + old + ")";
         } s = "Unitary("+s+");\n";
      }
   }
   return s;
}
```

Listing 4.113: *Body node* Visit method and "`printUnitaries`" function

4.6.12 *Return Node Evaluation*

The *Visit method* for the *return node* (Listing 4.114):

- prints the "`unitaries`" list, since a *return instruction* signals the end of the function;
- if the *input node* contains arguments, returns the result of *visiting* the respective *expression nodes* as a *string* (surrounded by parenthesis);
- if the *input node* does not contain arguments, returns an *empty string* instead;

```
public String Visit(RetNode node) {
   String s = printUnitaries(unitaries);
   if(node.getArgBool())
      return s+"Return \""+ Visit(node.getArgs())+"\";\n";
   else return s+"Return \"\";\n";
}
```

Listing 4.114: *Return node* Visit method

4.6.13 *For-loop Node Evaluation*

The *Visit method* for the *for-loop node* (Listing 4.115):

- prints the "`unitaries`" list, since the list is deleted upon *visiting* the *input node*'s *body*;
- adds the result of *visiting* the *for-iteration*, *invariant*, *body* and *assertion nodes* to the *string* "`s`";
- clears the "`unitaries`" list and returns the *string* "`s`".

```
public String Visit(ForNode node) {
   String s, body;
   s = printUnitaries(unitaries);
   s += "For {\n" + Visit(node.getIter())
      + Visit(node.getInvariant());
   body = Visit(node.getBody());
   s += body + Visit(node.getAssertion()) + "};\n";
   unitaries.clear();
   return s;
}
```

Listing 4.115: *For-loop node* Visit method

4.6.14 *For-iteration Node Evaluation*

The *Visit method* for the *for-iteration node* (Listing 4.116) handles the different iteration cases (covered in Subsection 3.3.6):

- starts by extracting the *iterator* identifier;
- if both "`range`" and "`iterQr`" *Boolean* fields are true in the *input node*, the iteration *limits* – defined by *string* variables "`s`" (start) and "`e`" (end) – are extracted from the result of *visiting* the *quantum register node*: the *string* **before** the semicolon character "`;`" represents the *start* and the *string* **after** represents the *end*;
- if the "`range`" *Boolean* is true and "`iterQr`" is false, variable "`s`" is set to "`Num 0`" (the number 0) and the result of *visiting* the *expression node* is assigned to "`e`";
- if instead only the "`iterQr`" *Boolean* is true, there are two cases: (1) in case the limits are specified in the *quantum register*, *i.e.* "`qr[a..b]`", the *string* which represents the *start* is assigned to "`s`" and the *string* which represents the *end* is assigned to "`e`" and (2) if the entire register is being iterated, *i.e.* "`qr`", "`s`" is set to "`Num 0`" (the number 0) and the last index of the register (given by the *subtraction* of the *register*'s length by the number 1) is assigned to "`e`";
- finally, if the "`iterQr`" *Boolean* is true, the *iteration*'s "`qr`" is set as the *quantum register*'s identifier, otherwise "`qr`" is set as " " (the space character).

```
public String Visit(ForIter node) {
   String s,t,e,iterator=node.getIterator();
   String[] limits;
   t = "iter = {\n";
   if(node.getRange() && node.getIterQr()) { // for i in range(qr)
      limits = Visit(node.getIterableQr()).split(";");
      s = limits[0];
      e = limits[1];
```

```
      t += "iterator= \"" + iterator + "\";\n"
         + "starts = " + s + ";\n"
         + "ends = " + e + "\n";
   }
   else if (node.getRange() && !node.getIterQr()) { // for i in range(expr)
      s = "Num 0";
      e = Visit(node.getIterableExpr());
      t += "iterator= \"" + iterator + "\";\n"
           + "starts = " + s + ";\n"
           + "ends = " + e + "\n";
   }
   else if(node.getIterQr()){ // for i in qr
      limits = Visit(node.getIterableQr()).split(";");
      if (limits.length>1) {
         s = limits[0];
         e = limits[1];
      }
      else {
         s = "Num 0";
         e = "Subtract(Var \"" + limits[0] + "\", Num 1)";
      }
      t += "iterator= \"" + iterator + "\";\n"
           + "starts = " + s + ";\n"
           + "ends = " + e + "\n";
   }
   else return "Can only iterate qreg or range operator\n";
   if (node.getIterQr())
      t += "};\nqr = \"" + node.getIterableQr().getId() + "\";\n";
   else t += "};\nqr = \" \";\n";
   return t;
}
```

Listing 4.116: *For-iteration node* Visit method

4.6.15 *If-statement and If-condition Nodes Evaluation*

The *Visit method* for the *if-statement* and *if-condition nodes* (Listing 4.117):

- prints the "`unitaries`" list, since the list is deleted upon *visiting* the *input node*'s *body*;
- if the *if-statement* does not contain an "*else*" section, the method declares an "`If`"*-type instruction*, and adds the result of *visiting* the *if-condition* [8], *body* and *assertion nodes* to the *string* "`s`";
- if the *if-statement* contains an "*else*" section the method declares an "`IfElse`"*-type instruction* and adds the result of *visiting* the *if-condition*, *if* and *else*'s *body* and *assertion nodes* to the *string* "`s`";
- clears the "`unitaries`" list and returns the *string* "`s`".

8 The *visit method* for the *if-condition node* returns the result of *visiting* the *term node* which expresses the *if*'s condition

```
public String Visit(IfNode node) {
   String s, body;
   s = printUnitaries(unitaries);
   if(!node.getWithElse()) {
      s += "If {\ncond= " + Visit(node.getCond()) + ";\n";
      body = Visit(node.getIfBody());
      s += body + Visit(node.getAssertion()) + "};\n";
   }
   else {
      s += "IfElse {\ncond= " + Visit(node.getCond()) + ";\nif"
            + Visit(node.getIfBody())
            + "else" + Visit(node.getElseBody())
            + Visit(node.getAssertion())
            + "};\n";
   }
   unitaries.clear();
   return s;
}

public String Visit(IfCond node) {
   return Visit(node.getExpr());
}
```

Listing 4.117: *If-statement* and *If-condition nodes* Visit method

4.6.16 *SingleApply node Visit*

The *Visit method* for *SingleApply nodes* – corresponding to gates H, X, Y, Z, T and S – update the global list "`unitaries`" with the respective *unitary type* gate-application. A *Visit method* is defined for each *SingleApply* gate. Code in Listing 4.118 defines the *Visit method* for the *Hadamard* gate. *Visit methods* for the remaining *SingleApply* gates have a similar definition, and differ solely in the *gate type* identifier, *i.e.* "`H`" for the *Hadamard* gate.
The method starts by extracting the *input node*'s identifier, after which the result of *visiting* the *quantum-register node* is assigned to a *string* variable "`qr`". Then there are three possibilities regarding "`qr`":

1. If the "`global_qrs`" list contains "`qr`", the gate is being **globally** applied to the register, *i.e.* "`H(qr)`". Thus the *start* and *end* values of the *range* parameter are given by "`Num 0`" and the *subtraction* of the *register*'s length by the number 1, respectively;

2. If the "`qr`" *string* contains a semi-colon character "`;`", the gate is being applied to a specific *interval* of the register, *i.e.* "`H(qr[a..b])`". The *sub-strings* **before** and **after** the semi-colon character represent the *start* and *end* values of the *range* parameter;

3. Otherwise, the gate is being applied to a specific *index* of the register, *i.e.* "`H(qr[a])`". In this case, both *start* and *end* values of the *range* parameter are set as "`qr`".

The *input node*'s *assertion* is extracted by *visiting* the respective *assertion node*.

After adding these parameters together in a *string* "s", the defined "*Apply*" *unitary* is added to the global "unitaries" list.

```
public String Visit(HadApply node) {
   String s, start, end, id, qr;
   String[] limits;
   id = node.getQreg().getId();
   qr = Visit(node.getQreg());
   if (global_qrs.contains(qr)) {
      s = "Apply {gate=H; qreg=\"" + id +
         "\"; range={starts=Num 0; ends=Subtract(Len \""+id+"\", Num 1)}; " +
         Visit(node.getAssertion())+"}";
   }
   else if (qr.contains(";")) {
      limits = qr.split(";");
      start = limits[0];
      end = limits[1];
      s = "Apply {gate=H; qreg=\"" + id +
            "\"; range={starts="+start+
            "; ends="+end+"}; "+Visit(node.getAssertion())+"}";
   }
   else {
      s = "Apply {gate=H; qreg=\"" + id +
            "\"; range={starts="+qr+
            "; ends="+qr+"}; "+Visit(node.getAssertion())+"}";
   }
   unitaries.add(s);
   return "";
}
```

Listing 4.118: *SingleApply* Visit method

4.6.17 *DoubleApply node Visit*

The *Visit method* for *DoubleApply nodes* – corresponding to gates $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$ and $Ph(\theta)$ – update the global list "unitaries" with the respective *unitary type* gate-application. A *Visit method* is defined for each *DoubleApply* gate. Code in Listing 4.119 defines the *Visit method* for the $R_x(\theta)$ gate. *Visit methods* for the remaining *DoubleApply* gates have a similar definition, and differ solely in the *gate type* identifier, *i.e.* "RX" for the $R_x(\theta)$ gate.
This method has the same structure as the *SingleApply* method (Listing 4.118), with the addition of the *angle expression* parameter, which is extracted by *visiting* the *input node*'s angle.

```
public String Visit(RxApply node) {
   String s, start, end;
   String id = node.getQreg().getId();
   String qr = Visit(node.getQreg());
   String[] limits;
   if (global_qrs.contains(qr)) {
      s = "Apply {gate=Rx ("+Visit(node.getAngle())
            +"); qreg=\"" + id
            + "\"; range={starts=Num 0; ends=Subtract(Len \""+id+"\", Num 1)}; "
```

```
            +Visit(node.getAssertion())+"}";
   }
   else if (qr.contains(";")) {
      limits = qr.split(";");
      start = limits[0];
      end = limits[1];
      s = "Apply {gate=Rx ("+Visit(node.getAngle())
            +"); qreg=\"" + id + "\"; range={starts="+
            start+"; ends="+end+"}; "+Visit(node.getAssertion())+"}";
   }
   else {
      s = "Apply {gate=Rx ("+Visit(node.getAngle())
            +"); qreg=\"" + id + "\"; range={starts="+
            qr+"; ends="+qr+"}; "+Visit(node.getAssertion())+"}";
   }
   unitaries.add(s);
   return "";
}
```

Listing 4.119: *DoubleApply* Visit method

4.6.18 *SwapApply node Visit*

The *Visit method* for the *SwapApply node* (Listing 4.120) – corresponding to the *Swap* gate – updates the global list "`unitaries`" with the respective *unitary type* gate-application. This method extracts both *input registers*' identifiers "`id1`" and "`id2`" and assigns the result of *visiting* the respective *quantum-register nodes* to "`lqr`" and "`rqr`". A *Swap* gate can only be applied to *specific indexes* of a register, thus an *if-statement* verifies this property by checking if neither "`lqr`" or "`rqr`" **contain** the semi-colon character, or **are contained** in the "`global_qrs`" list. The *input node*'s *assertion* is extracted by *visiting* the respective *assertion node*. After adding these parameters together in a *string* "`s`", the defined "*MultiApply*" (*Swap*) *unitary* is added to the global "`unitaries`" list.

```
public String Visit(SwapApply node) {
   String r, s, id1, id2, lqr, rqr;
   id1 = node.getLQreg().getId();
   id2 = node.getRQreg().getId();
   lqr = Visit(node.getLQreg());
   rqr = Visit(node.getRQreg());
   if (!(lqr.contains(";") || rqr.contains(";") || global_qrs.contains(lqr) ||
       global_qrs.contains(rqr))) {
      s = "MultiApply {gate=SWAP; " +
            "regs=[{iterator=\"" + id1 +
            "\"; starts=" + lqr +
            "; ends=" + lqr
            + "}; " + "{iterator=\"" + id2 +
            "\"; starts=" + rqr +
            "; ends=" + rqr
            + "}]; " + Visit(node.getAssertion())+"}";
   }
   else return "Can only apply SWAP gate to two specific indexes\n";
   unitaries.add(s);
```

```
   return "";
}
```

Listing 4.120: *SwapApply* Visit method

4.6.19 *Function and reverse-function application nodes Visit*

The *Visit method* for *function application node* (Listing 4.121) has a similar definition to the *reverse-function application node*'s *Visit method*. This method updates global lists "`auxIds`" and "`unitaries`" with the respective *function* identifier and *unitary type* gate-application, respectively. The method extracts the function's identifier and its *string* value it is added as the list's first element. If the identifier is present in "`auxIds`" list, the previous value is deleted. Then, the result of *visiting* each one of the function's *argument nodes* is added to the "`args`" variable. The function's identifier and argument parameters are placed together in a *string* "`s`", which defines a "*FUN*" (or "*REV*") *unitary*. Finally, "`s`" is added to the global "`unitaries`" list.

```
public String Visit(FunApply node) {
   StringBuilder args = new StringBuilder();
   String s;
   if (!auxIds.contains(node.getFunID()))
      auxIds.add(0,node.getFunID());
   else {
      auxIds.remove(node.getFunID());
      auxIds.add(0, node.getFunID());
   }
   for(TermNode arg:node.getTermArgs()){
      args.append(Visit(arg)).append("; ");
   }
   s = "FUN {id=\"" + node.getFunID() + "\"; args=["
      + args + "]}";
   unitaries.add(s);
   return "";
}
```

Listing 4.121: *Function* application Visit method

4.6.20 *With-controlled gate application Visit*

The *Visit method* for the *with-controlled node* (Listing 4.122) updates the global list "`unitaries`" with the respective *unitary type controlled-gate* application. The method starts by checking if the *gate* being applied as the *control* is also a *controlled gate* (*C-NOT*, *Toffoli* or *Fredkin*), in which case:

- the *controlled-gate node* is *visited*, which adds an element to the "`unitaries`" list;

- the added *unitary* is extracted, added to *string* "`s`" and removed from the "`unitaries`" list;
- variable "`id`" (the target *register*'s identifier) is set as "`tg`" and variable "`target`" (the target *register*'s range) is set as "`Num 0`" [9]

If the *gate* being applied as the *control* is a *non-controlled gate*:

- the *gate node* is *visited*, which adds an element to the "`unitaries`" list;
- the added *unitary* is extracted, added to *string* "`s`" and removed from the "`unitaries`" list;
- the method then checks which *gate* instance is being used and sets variables "`id`" (the target *register*'s identifier) and "`target`" (the target *register*'s range) accordingly;
- the list of *control registers* is iterated and for each *register*, its **identifier** and **range** is extracted and added to *string* "`s`";
- the target *register*'s **identifier** and **range** are extracted and added to *string* "`s`"
- the *input node*'s *assertion* is extracted by *visiting* the respective *assertion node* and added to *string* "`s`";
- *string* "`s`", which contains the defined "*WithControl*" *unitary* is added to the global "`unitaries`" list;

```
public String Visit(WithCtlNode node) {
   String[] limits;
   String angle,id=null,r,start,end,range,target,target1,target2;
   StringBuilder s = new StringBuilder("WithControl{ctlgate=");
   if (node.getIsMulti()){
      Visit(node.getCtlMulti());
      s.append(unitaries.get(unitaries.size() - 1)); // get last unitary in list
      unitaries.remove(unitaries.size() - 1);
      id = "tg";
      target = "Num 0";
   }
   else {
      ApplyNode gate = node.getCtlGate();
      Visit(gate);
      s.append(unitaries.get(unitaries.size() - 1)); // get last unitary in list
      unitaries.remove(unitaries.size() - 1); // delete it since its not needed
          anymore
      if (gate instanceof FunApply) {
         target = "Var \"null\"";
         id = "Var \"null\"";
      } else if (gate instanceof HadApply) {
         target = Visit(((HadApply) gate).getQreg());
```

9 Note that for the *controlled gates* these parameters become irrelevant when the final translation occurs.

```
            id = ((HadApply) gate).getQreg().getId();
         } else if (gate instanceof RxApply) {
            target = Visit(((RxApply) gate).getQreg());
            id = ((RxApply) gate).getQreg().getId();
         ...
      s.append("; ctls=[");
      for (QregNode n:node.getCtlArgs()){
         s.append("{iterator=\"").append(n.getId()).append("\"; ");
         range = Visit(n);
         if (range.contains(";")) {
            limits = range.split(";");
            start = limits[0]; end = limits[1];
         }
         else {
            start = range;
            end = range;
         }
         s.append("starts=").append(start);
         s.append("; ends=").append(end).append("}; ");
      }
      s.append("]; tg={iterator=\"").append(id).append("\"; ");
      if (global_qrs.contains(target)) {
         target1 = "Num 0";
         target2 = "Subtract(Len \""+id+"\", Num 1)";
      }
      else if (target.contains(";")) {
         limits = target.split(";");
         target1 = limits[0];
         target2 = limits[1];
      }
      else {
         target1 = target;
         target2 = target;
      }
      s.append("starts=").append(target1).append("; ends=").append(target2).append("}; ")
          ;
      s.append("starts=").append(target).append("; ends=").append(target).append("}; ");
      s.append(Visit(node.getAssertion())).append("}\n");
      unitaries.add(s.toString());
      return "";
   }
```

Listing 4.122: *With-controlled* gate application Visit method

4.6.21 *C-NOT, Toffoli and Fredkin nodes Visit*

The *Visit methods* for *C-NOT, Toffoli and Fredkin nodes* (Listing 4.123) have similar definitions, and differ in the *gate type* identifier, *i.e.* "`Cnot`" for the *C-NOT* gate, and number of *control registers* (**one** for the *C-NOT* gate and **two** for the *Toffoli and Fredkin* gates). This method extracts the *control* and *target registers*' identifiers "`ctl`" and "`tg`" and assigns the result of *visiting* the respective *quantum-register nodes* to "`ctlqr`" and "`tgqr`". Any of the mentioned gates can only be applied to *specific indexes* of a register, thus an *if-statement* verifies this

property by checking if neither "ctlqr" or "tgqr" **contain** the semi-colon character, or **are contained** in the "global_qrs" list. The *input node*'s *assertion* is extracted by *visiting* the respective *assertion node*. After adding these parameters together in a *string* "s", the defined "*MultiApply*" (*C-NOT*) *unitary* is added to the global "unitaries" list.

```
public String Visit(CnotNode node) {
   String s,ctl=node.getCtl().getId(),tg=node.getTarget().getId();
   String ctlqr = Visit(node.getCtl()), tgqr = Visit(node.getTarget());
   if (!(ctlqr.contains(";") || tgqr.contains(";") || global_qrs.contains(ctlqr) ||
       global_qrs.contains(tgqr) )) {
      s = "MultiApply {gate=Cnot; regs=[{iterator=\""
            + ctl + "\"; starts="+ ctlqr + "; ends=" + ctlqr
            +"}; {iterator=\""+
            tg + "\"; starts=" + tgqr + "; ends=" + tgqr
            +"}]; " + Visit(node.getAssertion())+"}";
   }
   else return "Can only apply CNOT gate to two specific indexes\n";
   unitaries.add(s);
   return "";
}
```

Listing 4.123: *C-NOT* gate application Visit method

4.6.22 *Conjugated-basis Node Evaluation*

The *Visit method* for the *conjugated node* (Listing 4.124):

- prints the "unitaries" list, since the list is deleted upon *visiting* the *input node*'s *body*;
- the *gate* being applied as the *conjugated operator* is *visited*, which adds an element to the "unitaries" list;
- the added *unitary* is extracted, added to *string* "s" and removed from the "unitaries" list;
- adds the result of *visiting* the *body* and *assertion nodes* to the *string* "s";
- clears the "unitaries" list and returns the *string* "s".

```
public String Visit(ConjNode node) {
   String body, s = printUnitaries(unitaries);
   s += "Conjugated {gate=";
   Visit(node.getApply());
   s += unitaries.get(unitaries.size()-1);
   unitaries.remove(unitaries.size()-1);
   s += ";\n ";
   body = Visit(node.getBody());
   s += body+ Visit(node.getAssertion()) + "};\n";
   unitaries.clear();
   return s;
}
```

Listing 4.124: *Conjugated node* Visit method

4.6.23 *Expression and Term nodes Visit*

Visit methods for *expression* and *term nodes* which consist in operations with **two** terms (Listing 4.125) return the respective *condition* or *expression type* with the *left* and *right* terms, as covered in Subsection 4.5.10, *i.e.* "`Eq(a,b)`" for an "`a==b`" expression or "`Plus(a,b)`" for an "`a+b`" expression.

```
public String Visit(EqualNode node) {
   return "Eq ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(GTNode node) {
   return "Gt ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(LTNode node) {
   return "Lt ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(GEQNode node) {
   return "GEq ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(LEQNode node) {
   return "LEq ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(NEqualNode node) {
   return "NEq ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(PowerNode node) {
   return "Power ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(MulNode node) {
   return "Times ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(DivNode node) {
   return "Divide ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(AddNode node) {
   return "Plus ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
public String Visit(SubNode node) {
   return "Subtract ("+Visit(node.getLeft()) + "," + Visit(node.getRight())+")";
}
```

Listing 4.125: *Two-term expressions and terms* Visit methods

The *Visit* method for the *unary node* (Listing 4.126) checks which operator is being used (either *square-root* or *negative sign*) and returns the respective *expression type*.

```
public String Visit(UnOpNode node) {
   String r;
   if (node.getOp().equals("sqrt"))
      r = "Sqrt " + "("+Visit(node.getInnerTerm())+")";
   else r = node.getOp() + "(" + Visit(node.getInnerTerm())+")";
   return r;
}
```

Listing 4.126: *Unary node* Visit method

The *Visit* method for the *length node* (Listing 4.127) returns the value associated to a *register*'s size: the *subtraction* of the *register*'s length by the number 1.

```
public String Visit(LenNode node) {
   return "Subtract (Len \""+Visit(node.getQrTerm())+"\", Num 1)";
}
```

Listing 4.127: *Length node* Visit method

The *Visit* method for the *parenthesis node* (Listing 4.128) returns the result of visiting the *inner term node*.

```
public String Visit(ParenNode node) {
   return Visit(node.getTerm());
}
```

Listing 4.128: *Parenthesis node* Visit method

The *Visit* method for the *Atom node* (Listing 4.129) returns the *input node*'s value.

```
public String Visit(AtomNode node) {
   return node.getValue();
}
```

Listing 4.129: *Atom node* Visit method

4.7 GENERATING A PROGRAM IN QBRICKS

In order to translate from a program written in iQbricks to an AST, a *Main* class must be defined. The purpose of this main class is to provide a convenient entry point for running the language's compiler and to ensure that the various components of the compiler are properly integrated and executed in the correct order to produce a valid output.

This *Main* class (Listing 4.130) is responsible for orchestrating the different components of the iQbricks compiler. It reads in the input file (*i.e.* "`qft`") containing the source code, passes it through the *lexer* and *parser* to produce a *Concrete-Syntax Tree*, which is then visited by the "`ASTBuilder`" to generate a *Java-AST*. The *Java-AST* is then *visited* by the *Java-AST Evaluator* to generate the corresponding *ML-AST* code. Finally, the generated *ML-AST* code is written to a file named "`ocaml_ast.ml`".

```
public class Main {
   public static void main(String[] args)
   {
      try {
         QbricksLexer lexer = new QbricksLexer(CharStreams.fromFileName("qft.txt"));
         CommonTokenStream stream = new CommonTokenStream(lexer);
         QbricksParser parser = new QbricksParser(stream);
         ParseTree tree = parser.program();
         ASTBuilder builder = new ASTBuilder();
         AST ast = builder.visit(tree);
         String val = new EvaluateVisitor().Visit((ProgramNode) ast);
         PrintWriter writer = new PrintWriter("ocaml_ast.ml");
```

```
            writer.write(val);
            writer.close();
            System.out.println("Translation successful");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 4.130: *Java* Main Class

4.7.1 *ML-AST Evaluator*

The implemented *ML-AST Evaluator* is a collection of functions that operate on the generated *ML-AST* types and produce code in the Qbricks language. Each data-type in the *ML-AST* has a corresponding function defined in the *Evaluator*, which takes the input data-type, processes its contents, and returns the resulting code in Qbricks language. Essentially, the *Evaluator* provides a mapping from the *ML-AST* to the Qbricks language by defining a translation process for each *ML-AST* data-type.

In the context of programming in *Ocaml*, a program's designation begins with the definition of the *module*, which in this particular implementation corresponds to the identifier of the program's *main function*. Subsequently, *modules* that are to be imported into the program are included. Following this, the program's *auxiliary* and *main* functions are written.

Functions in iQbricks represent quantum circuits, thus each function in the translated program returns a '*circuit*' object (defined in Qbricks' library). Program instructions consist of control-flow operations and quantum gate applications. Instructions are therefore defined as a series of '*circuit*' objects, which can be sequenced together in order to compose a more complex circuit.

4.7.2 *Illustrating Example*

Function "`f`" defined in Listing 4.131 represents a program in iQbricks which receives a *quantum-register* "`qr`" as a parameter and applies the *X* and *Hadamard* gates to its first index. The program's *pre* and *post-conditions* state that the *size* of "`qr`" is bigger than zero and the *width* of the resulting circuit is equal to the *size* of "`qr`", respectively.

```
|| f || (qreg qr)
pre {qr > 0}
circ qr ->
      X(qr[0])
      H(qr[0])
pos {width result = qr}
```

Listing 4.131: Function "`f`" represented using iQbricks

The translated program (Listing 4.132) begins by defining a new *module* named "F" (the *main* function's identifier). The module then includes a set of predefined *imports*, which are specified using the use keyword "use" followed by the name of the library to be imported. These imports provide access to the functions and data structures defined in QBRICKS libraries, which are necessary to write the program.

```
module F

use export binary.Bit_vector
use wired_circuits.Circuit_c
(...)
use unit_circle.Angle
```

Listing 4.132: Function "f" after translation to QBRICKS (1)

After adding the predefined and manually added imports to the module, the *main* function is included (Listing 4.133). This function:

- is named "f";
- has two integer parameters – "qr" and "n", which represent the **size** of the register "qr" and the **width** of the circuit, respectively;
- returns a "circuit" object

Since "qr" is the circuit's only register, a *pre-condition* is automatically generated which states that "qr" and "n" have the same value. Besides this *pre-condition* and the one manually introduced in the original program (*size* of "qr" is bigger than zero), another is generated which states that "n" is bigger than zero [10].

```
let f (qr n:int) : circuit
requires{0 < n}
requires{qr = n}
requires{qr > 0}
```

Listing 4.133: Function "f" after translation to QBRICKS (2)

Following the *pre-conditions*, the function's body is included (Listing 4.134), in which:

- a reference integer "qr_index" with a value of zero is declared, which is used to correctly place gates on register's indexes;
- a reference circuit "c0" with a value of "m_skip n" [11] is declared;
- the circuit resulting from applying an X gate to the index zero of register "qr" is added in sequence to the previous value of the "c0" reference using the QBRICKS sequence operator "--";

10 This condition is universal for circuits in QBRICKS, thus it is automatically generated for all functions
11 "m_skip N" is a QBRICKS function which creates an empty circuit with size "N"

- the same sequencing method is used for applying an *Hadamard* gate, which also updates the previous value of the "`c0`" reference;
- reference "`c0`", which represents the resulting circuit, is returned, after which the program's *post-condition* is included and the program ends.

```
let f (qr n:int) : circuit
requires{0 < n}
requires{qr = n}
requires{qr > 0}
=
begin
let qr_index = ref (0: int) in
   let c0 = ref (m_skip n) in
      c0 := !c0 -- (place xx (!qr_index + 0) n);
      c0 := !c0 -- (place_hadamard (!qr_index + 0) n);
   return !c0;
ensures{width result = qr}
end
```

Listing 4.134: Function "`f`" after translation to Qbricks (3)

After the translated program is generated, it can then be tested against several *SMT-Solvers* using *Why3* (Filliâtre and Paskevich, 2013). *Why3* is a specification environment that offers a *ML-like* language for both programming and writing specifications. From a specified program, *Why3* generates a set of *proof obligations* that need to be satisfied to certify the program. These proofs can be manipulated through a dedicated graphical interface, and *Why3* also provides an interactive proof assistant. To prove a theorem, one can call a set of automatic *SMT solvers* (*CVC, Z3, Alt-ergo,* etc.) accessible from the interface. It is also possible to call lemmas to provide proof steps to the solvers, as well as to directly simplify proof objectives through different term transformation commands.

5

TESTING

The present chapter presents the implementation and translated programs of two quantum algorithms: *Grover* and *Quantum Fourier Transform* (QFT), along with the obtained results for each implementation.

5.1 INTRODUCTION

Test files are crucial in the development of IQBRICKS language because they enable the verification that the language implementation behaves as expected. Testing helps to identify and fix errors, and ensures that new features or changes do not introduce regressions into existing functionality. By re-running the tests after each change, it can be ensured that any modifications to the implementation have not broken existing functionality or introduced new bugs. This helps to maintain the overall stability and quality of the language implementation, and builds confidence in its correctness and reliability.

5.2 GROVER'S ALGORITHM

Grover's algorithm (Grover, 1996) demonstrates the speed advantage a quantum computer has over a classical computer in searching unstructured databases. This algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms (Nielsen and Chuang, 2010). Its engine is a technique called *amplitude amplification*.

Suppose we wish to search through a search space of N elements. Instead of searching its elements directly, we will focus on the *index* of those elements, which is in the interval $[0, N-1]$. For convenience, let us assume that $N = 2^n$, so the index can be stored in n bits, and that the problem has exactly M solutions, with $1 \leq M \leq N$. Now, let us consider the existence of a function f, which takes an input integer x in the range $[0, N-1]$ and returns

values $f(x) = 1$ if x is a solution to the search problem, and $f(x) = 0$ if it is not. We can now define a quantum *oracle* which has the ability to recognize solutions to the search problem. This recognition is signalled by making use of an *oracle qubit*, which is a unitary operator denoted O:

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle \tag{30}$$

where $|x\rangle$ is the index register, $\oplus$ denotes addition modulo 2, and the *oracle qubit* $|q\rangle$ is a single qubit which is flipped if $f(x) = 1$, and is unchanged otherwise. We can now define the oracle O to act on any of the standard basis states $|x\rangle$ by $|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle$. We then say that the oracle *marks* the solutions to the search problem, by shifting the phase of the solution. More details are available in Nielsen and Chuang (2010).

5.2.1 *The procedure*

The algorithm begins with the equal superposition state,

$$H |0\rangle^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |\psi\rangle \ . \tag{31}$$

The search algorithm then consists of a repeated application of a quantum subroutine, known as the *Grover iterator*, which will be denoted G (Figure 18). The number of necessary applications of G for an N item search problem with M solutions is given approximately by $\sqrt{N/M}$.

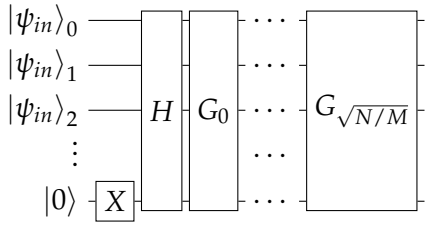


Figure 18: Circuit respective to *Grover*'s algorithm

The procedure for the *Grover iterator* (Figure 19) can be divided into four steps:

1. Apply the oracle O;
2. Apply the Hadamard transform $H^{\otimes n}$;
3. Perform the conditional phase shift[1]

$$|\psi\rangle \rightarrow -(-1)^{\delta_{x0}} |\psi\rangle$$

1 every computational basis state except $|0\rangle$ receives a phase shift of -1

4. Apply the Hadamard transform $H^{\otimes n}$;

Steps 2, 3, and 4 are commonly known as the *Grover diffusion* operator.

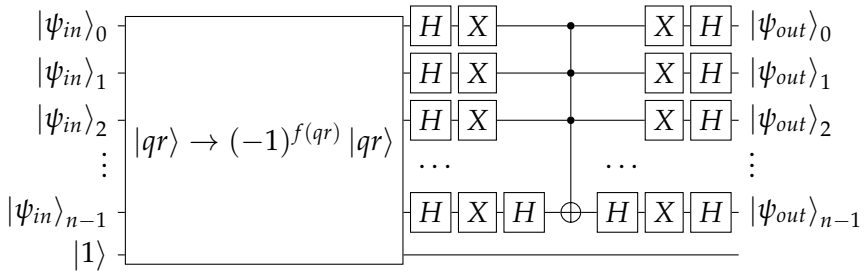


Figure 19: Circuit respective to *Grover*'s iteration subroutine

5.2.2 *Program in* iQbricks

Grover's algorithm implementation using iQbricks consists of a *main* function "`grover`" and three *auxiliary* functions "`init`", "`grover_iter`" and "`diffusor`". This implementation of the algorithm **does not** include its specification.

Main function "`grover`" (Listing 5.1) receives four parameters:

- the *oracle* **circuit** "`oracle`";
- input **registers** "`qr`" and "`aux`";
- the number of **iterations** "`iters`".

The circuit for the *main* function:

- starts by declaring "`qr`" and "`aux`" as the circuit's registers;
- applies "`init`" function to "`qr`" and "`aux`" registers;
- applies "`grover_iter`" function using the input parameter "`oracle`" and registers "`qr`" and "`aux`";
- returns the generated circuit.

```
|| grover || (circ oracle, qreg qr, qreg aux, int iters)
pre {true}
   circ qr, aux ->
      init(qr,aux)
      for i in range(iters) {
         invariant{true}
         grover_iter(oracle,qr,aux)
      }
      return
pos {true}
```

Listing 5.1: Implementation in iQbricks of a fragment of Grover's algorithm

Auxiliary function "`init`" (Listing 5.2) corresponds to *Grover*'s initialization routine, which creates the initial superposition state and inverts the state of input auxiliary register from $|0\rangle$ to $|1\rangle$. This function receives two arguments: registers "`qr`" and "`aux`". Its circuit:

- declares "`qr`" and "`aux`" as the circuit's registers;
- applies *Hadamard* gates globally to "`qr`" register;
- applies an X gate to "`aux`" register;
- applies an *Hadamard* gate to "`aux`" register;
- returns the generated circuit.

```
| init | (qreg qr, qreg aux)
pre {true}
   circ qr, aux ->
      H(qr)
      X(aux)
      H(aux)
      return
pos {true}
```

Listing 5.2: Implementation in iQbricks of a fragment of Grover's algorithm (`init`)

Auxiliary function "`grover_iter`" (Listing 5.3) corresponds to *Grover iterator* subroutine, which applies the *oracle* and *diffusion* operators. This function receives three arguments: the circuit *oracle*, registers "`qr`" and "`aux`". Its circuit:

- declares "`qr`" and "`aux`" as the circuit's registers;
- applies the *oracle* circuit globally [2];
- applies "`diffusor`" function to "`qr`" and "`aux`" registers;
- returns the generated circuit.

2 oracle circuits correspond to functions which don't contain parameters. When received as input by a function, such functions are always applied to all of the function's circuit registers

```
| grover_iter | (circ oracle, qreg qr, qreg aux)
pre {true}
   circ qr, aux ->
      oracle()
      diffusor(qr,aux)
      return
pos {true}
```

Listing 5.3: Implementation in IQBRICKS of a fragment of Grover's algorithm (`grover_iter`)

Auxiliary function "`diffusor`" (Listing 5.4) corresponds to the application of the *Grover diffusion* operator. This function receives two arguments: registers "`qr`" and "`aux`". Its circuit:

- declares "`qr`" and "`aux`" as the circuit's registers;
- applies *Hadamard* and X gates as *conjugate-based* operators to the "`qr`" register [3]. This means that these gates are applied globally to "`qr`" at both the beginning and end of the circuit;
- applies a *controlled-Z* gate using the indexes of "`qr`" **up-to** the last one as *controls* and the last index of "`qr`" as the *target* [4];
- returns the generated circuit.

```
| diffusor | (qreg qr, qreg aux)
pre {true}
   circ qr, aux ->
      with conjugated (H(qr)) {
         with conjugated (X(qr)) {
            with control qr[:-1] (Z(qr[-1]))
         }
      }
      return
pos {true}
```

Listing 5.4: Implementation in IQBRICKS of a fragment of Grover's algorithm (`diffusor`)

In order to run the *Grover* implementation, another program, which creates an *oracle* and calls the "`grover`" function, was developed. This program contains a *main* function "`grover_run`", which:

- receives the number of iterations "`iters`" and input registers "`qr`" and "`aux`" as arguments;
- calls the "`grover`" *main* function using "`oracle_toff`" function, "`qr`" and "`aux`" registers and the number of iterations "`iters`" as the function arguments.

3 note that $H^\dagger = H$ and $X^\dagger = X$

4 note that applying a Z gate is the same as applying the sequence of an *Hadamard*, followed by an X and another *Hadamard* gate

Auxiliary function "`oracle_toff`" applies a *Toffoli* gate using the first and second registers as controls and the third register as the target. This oracle was chosen arbitrarily, thus it does not necessarily correspond to a specific problem. This program's implementation is expressed in Listing 5.5.

```
|| grover_run || (qreg qr, qreg aux, int iters)
pre {true}
   circ qr, aux ->
      grover(oracle_toff, qr, aux, iters)
pos {true}

| oracle_toff |
pre {true}
   circ qr ->
      toff(qr[0],qr[1],qr[2])
pos {true}
```

Listing 5.5: Implementation in iQbricks of a fragment of Grover's algorithm (`grover_run` and `oracle_toff`)

5.2.3 *Translation generated*

After translation of the iQbricks implementation of *Grover*'s algorithm, the generated Qbricks program consists of a *module* named "`Grover`" which includes the *main* and three *auxiliary* functions.

The *main* "`grover`" function (Listing 5.6):

- receives a *circuit* type variable "`oracle`" and *integer* types "`qr`", "`aux`", "`iters`" and "`n`" as arguments;
- generates *pre-conditions* "`0<n`" and "`qr+aux=n`";
- declares the starting indexes for both "`qr`" and "`aux`" registers;
- creates empty circuits "`c0`" and "`c1`";
- iterates from 0 up until the value of "`iters`", updating the "`c1`" circuit with the "`grover_iter`" function application at each iteration, using "`oracle`", "`qr`", "`aux`" and "`n`" as arguments. Then, "`c1`" is sequenced to "`c0`", after which an empty circuit is again assigned to "`c1`". Two *loop-invariants* are automatically generated which state that the *width* of both "`c0`" and "`c1`" circuits is equal to "`n`";
- applies the "`init`" function (using "`qr`", "`aux`" and "`n`" as arguments) at the **beginning** of the circuit, updating "`c0`";
- returns the "`c0`" circuit.

```
let grover(oracle : int -> circuit) (qr aux iters n:int) : circuit
requires{0<n}
requires{qr+aux=n}
requires{true}
=
begin
let qr_index = ref (0: int) in
let aux_index = ref (0+qr: int) in
let c0 = ref (m_skip n) in
let c1 = ref (m_skip n) in
for i = 0 to (iters) do
   invariant{width !c0=n}
   invariant{width !c1=n}
   invariant{true}
   c1 := !c1 -- (grover_iter oracle qr aux n);
   c0 := !c0 -- !c1;
   c1 := m_skip n;
   done;
c0 := (init qr aux n) -- !c0;
return !c0;
ensures{true}
end
```

Listing 5.6: Grover *main* function translation

Auxiliary "`init`" function (Listing 5.7):

- receives *integer* types "`qr`", "`aux`" and "`n`" as arguments;
- generates *pre-conditions* "`0<n`" and "`qr+aux=n`";
- declares the starting indexes for both "`qr`" and "`aux`" registers;
- creates empty circuit "`c0`";
- creates empty circuit "`circ_aux`", which is used to perform the global application of the *Hadamard* gate on the "`qr`" register. This is done using a *for-loop* which iterates from the first index up to the last index of "`qr`" and applies an *Hadamard* gate at each iteration. After the *loop*, "`circ_aux`" is sequenced to "`c0`";
- follows the same procedure done with the *Hadamard* gates, this time applying X and *Hadamard* gates to "`aux`". As done before, after each *loop*, "`circ_aux`" circuits are sequenced to "`c0`";
- returns the "`c0`" circuit.

```
let init (qr aux n:int) : circuit
requires{0<n}
requires{qr+aux=n}
requires{true}
=
begin
let qr_index = ref (0: int) in
let aux_index = ref (0+qr: int) in
let c0 = ref (m_skip n) in
let circ_aux = ref (m_skip n) in
for i= (!qr_index + 0) to (!qr_index + qr - 1) do
   invariant{width !c0=n}
   circ_aux := !circ_aux -- (place_hadamard i n);
   done;
c0 := !c0 -- !circ_aux;
let circ_aux = ref (m_skip n) in
for i= (!aux_index + 0) to (!aux_index + aux - 1) do
   invariant{width !c0=n}
   circ_aux := !circ_aux -- (place xx i n);
   done;
c0 := !c0 -- !circ_aux;
assert{true};
let circ_aux = ref (m_skip n) in
for i= (!aux_index + 0) to (!aux_index + aux - 1) do
   invariant{width !c0=n}
   circ_aux := !circ_aux -- (place_hadamard i n);
   done;
c0 := !c0 -- !circ_aux;
assert{true};
return !c0;
ensures{true}
end
```

Listing 5.7: Grover "`init`" function translation

Auxiliary "`grover_iter`" function (Listing 5.8):

- receives *integer* types "`qr`", "`aux`" and "`n`" as arguments;
- generates *pre-conditions* "`0<n`" and "`qr+aux=n`";
- declares the starting indexes for both "`qr`" and "`aux`" registers;
- creates empty circuit "`c0`";
- applies the "`oracle`" function (using "`n`" as argument), updating "`c0`";
- applies the "`diffusor`" function (using "`qr`", "`aux`" and "`n`" as arguments), updating "`c0`";
- returns the "`c0`" circuit.

```
let grover_iter(oracle : int -> circuit) (qr aux n:int) : circuit
requires{0<n}
requires{qr+aux=n}
requires{true}
=
begin
let qr_index = ref (0: int) in
let aux_index = ref (0+qr: int) in
let c0 = ref (m_skip n) in
c0 := !c0 -- (oracle n);
c0 := !c0 -- (diffusor qr aux n);
return !c0;
ensures{true}
end
```

Listing 5.8: Grover "iter" function translation

Auxiliary "diffusor" function (Listing 5.9):

- receives *integer* types "qr", "aux" and "n" as arguments;
- generates *pre-conditions* "0<n" and "qr+aux=n";
- declares the starting indexes for both "qr" and "aux" registers;
- creates empty circuits "c0", "c1" and "c2";
- creates an empty circuit "circ_aux", which is used to apply the *multi-controlled Z* gate, using a *for-loop* which iterates from the first index up to the next-to-last index of "qr" and applies the *controlled* gate at each iteration; "circ_aux" is then sequenced with the empty "c2" circuit;
- creates another empty circuit "circ_aux", which is used to perform the *conjugate-based* operation on the "qr" register using the *X* gate, again using the *for-loop* which applies an *X* gate at each iteration; "circ_aux" and its *reverse* are then sequenced before and after the "c2" circuit, respectively, after which this sequence is attached to the empty "c1" circuit;
- applies the same process used for the *conjugate-based* operation with the *X* gate, now using the *Hadamard* gate and this time updating "c0" circuit;
- returns the "c0" circuit.

```
let diffusor (qr aux n:int) : circuit
requires{0<n}
requires{qr+aux=n}
requires{true}
=
begin
let qr_index = ref (0: int) in
```

```
let aux_index = ref (0+qr: int) in
let c0 = ref (m_skip n) in
let c1 = ref (m_skip n) in
let c2 = ref (m_skip n) in
let circ_aux = ref (m_skip n) in
for ctl= (!qr_index + 0) to (!qr_index + qr - 2) do
   invariant{width !c2=n}
   circ_aux := !circ_aux -- (cont zz ctl (!qr_index + qr - 1) n);
   done;
c2 := !c2 -- !circ_aux;
let circ_aux = ref (m_skip n) in
for i= (!qr_index + 0) to (!qr_index + qr - 1) do
   invariant{width !c1=n}
   circ_aux := !circ_aux -- (place xx i n);
   done;
c1 := !c1 -- !circ_aux -- !c2 -- reverse (!circ_aux);
let circ_aux = ref (m_skip n) in
for i= (!qr_index + 0) to (!qr_index + qr - 1) do
   invariant{width !c0=n}
   circ_aux := !circ_aux -- (place_hadamard i n);
   done;
c0 := !c0 -- !circ_aux -- !c1 -- reverse (!circ_aux);
return !c0;
ensures{true}
end
```

Listing 5.9: Grover "diffusor" function translation

The translation for the "grover_run" *main* and "oracle_toff" functions is expressed in Listing 5.10.

Regarding the "oracle_toff" function, it receives a single parameter "n" (the circuit's size) and sequences a *Toffoli* gate using the first and second registers as controls and the third register as the target to an empty circuit "c0", which is then returned.

Main "grover_run" function sequences the resulting circuit from the application of the "grover" function using "oracle_toff", "qr", "aux", "iters" and "n" as function arguments to an empty circuit "c0", which is then returned.

```
let oracle_toff (n:int) : circuit
requires{0<n}
requires{true}
=
begin
let qr_index = ref (0: int) in
let c0 = ref (m_skip n) in
c0 := !c0 -- (toffoli (!qr_index + 0) (!qr_index + 1) (!qr_index + 2) n);
return !c0;
ensures{true}
end

let grover_run (qr aux iters n:int) : circuit
requires{0<n}
requires{qr+aux=n}
requires{true}
=
```

```
begin
let qr_index = ref (0: int) in
let aux_index = ref (0+qr: int) in
let c0 = ref (m_skip n) in
c0 := !c0 -- (grover oracle_toff qr aux iters n);
return !c0;
ensures{true}
end
```

Listing 5.10: "grover_run" and "oracle_toff" functions translation

5.2.4 *Results*

From the translation of iQbricks *Grover´s* algorithm implementation, two Qbricks programs are generated: "grover" (Listings 5.6, 5.7, 5.8 and 5.9) and "grover_run" (Listing 5.10). These programs were tested by making use of Qbricks' *transpilation* tool, which generates an *OpenQASM* (Cross et al., 2017) circuit from a Qbricks program. The transpilation tool allowed not only to verify the syntactic correctness of the generated Qbricks programs, but also to visually verify the circuit by generating the respective *OpenQASM* circuit.

The generated *OpenQASM* circuit for the "grover_run" Qbricks program with parameters: qr=2, aux=1, iters=2 and n=3 is shown in Figure 20.

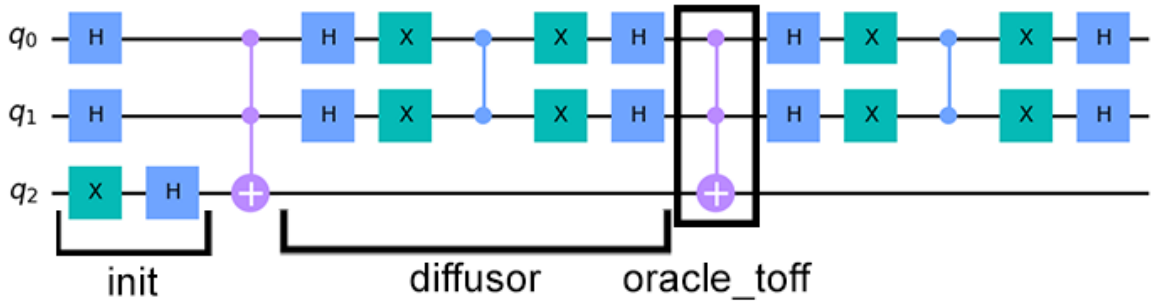


Figure 20: Generated *OpenQASM* grover_run circuit

We can thus conclude that the translation of Qbricks *Grover´s* algorithm implementation is successful and the generated quantum circuit matches the intended design.

5.3 QFT ALGORITHM

The *Quantum Fourier Transform* (*QFT*) algorithm, illustrated in Subsection 3.1.3, for any $n > 0$, transforms a superposition state $|x\rangle_n = \sum_{k=0}^{2^n-1} x_k |k\rangle_n$ in the state:

$$QFT(|x\rangle_n) = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} \sum_{j=0}^{2^n-1} x_j \, e^{\frac{2\pi i \cdot jk}{2^n}} |k\rangle_n . \tag{32}$$

The circuit for *QFT* is drawn in Figure 21, where R_k is a rotation gate defined by the matrix:

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix} \tag{33}$$

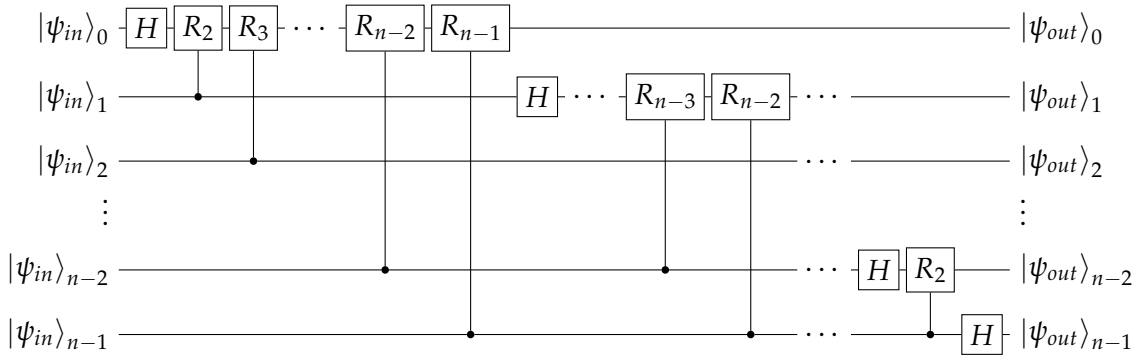


Figure 21: *QFT* circuit implementation

5.3.1 *Program in* iQbricks

The implementation for the *QFT* algorithm in iQbricks (Listing 5.11) consists in a *main* function "`qft`" and contains several specifications (both *assertions* and *invariants*) that define the expected behavior of the circuit.

This function receives a single register "`qr`" as an argument, and makes use of a *nested-for* structure in order to construct the *QFT* circuit. The function loops over the register "`qr`", applying a *Hadamard* gate to each qubit in turn. It then applies a series of *controlled Z-rotation* ($R_z(\theta)$) gates to the remaining qubits in the register, using the current qubit as the control and the angle $\theta = \frac{2\pi i}{2^{i-q+1}}$, where q is the *target* qubit and $i+1$ the current control qubit. The generated *QFT* circuit is then returned.

Regarding the program's specification, *invariants* and *assertions* are used to specify properties of the circuit at specific points during its execution. In this case, there are two *invariants* that occur within the *main* for-loop. The first *invariant* specifies the number of created *superpositions* ("`range result = q`", so it corresponds to the number of applied *Hadamard* gates) as well as the *basis* and *angular indexing* of the circuit before the *Hadamard* gate is applied. The second *invariant* specifies the number of created *superpositions* (in this case, "`range result = 0`", since there are no *Hadamard* gates inside the loop) and the *basis* and *angular indexing* of the circuit after the *controlled Z-rotation* gates are applied.

The *assertions* are used to check that the *invariants* hold after each step of the circuit. If an assertion fails, it means that the circuit has not performed as expected, and the program will not pass the proof obligation generated by the *Why3* tool.

The *post-condition* specifies that the output circuit must have the same *width* as the input ("`width result = n`"), have n created *superpositions* from the successive application of *Hadamard* gates ("`range result = n`") and follow the correct *basis* and *angular indexing*.

```
|| qft || (qreg qr)
pre {true}
      circ qr ->
      for q in range(len(qr)) {
         invariant {
            {range circ = q},
            {forall x y i. 0<= i < n -> basis_ket circ x y i = if 0<= i < q then y i
                else x i},
            {forall x y. ang_ind circ x y = (ind_isum(fun k -> (ind_isum (fun l -> x l
                 * y k * power 2 (n-l - 1+k)) k n))0 q) /./ n}
         }
         H(qr[q])
         assert {
            {width circ = n},
            {forall x y: int->int. forall i:int. 0<=i<n ->basis_ket circ x y i = if i
                = q then y 0 else x i},
            {forall x y. ang_ind circ x y = (ind_isum (fun l -> x l * y 0 * power 2 (n-
                l - 1+ q)) q n) /./ n}
         }
         for i in range(qr[q+1:]) {
            invariant {
               {range circ = 0},
               {forall x y i. 0<= i < n -> basis_ket circ x y i = x i},
               {forall x y. ang_ind circ x y = (ind_isum (fun l -> x l * x q * power 2
                    (n- l -1+ q)) (q+1) i) /./n}
            }
            with control qr[i+1] (RZ(i-q, qr[q]))
         }
      }
      return
pos {
   {width result = n},
   {range result = n},
   {forall x y i. 0<= i < n -> basis_ket result x y i = y i},
   {forall x y. ang_ind result x y = (ind_isum(fun k -> (ind_isum (fun l -> x l * y k
       * power 2 (n-l - 1+k)) k n))0 n) /./ n}
}
```

Listing 5.11: QFT IQBRICKS implementation

5.3.2 *Translation generated*

From the translation of iQbricks *QFT* implementation, a Qbricks program is generated which consists in a *module* named "`Qft`" which contains the *main* "`qft`" function (Listing 5.12).

This function:

- receives *integer* types "`qr`" and "`n`" as arguments;
- generates *pre-conditions* "`0<n`" and "`qr=n`";
- declares the starting index for the "`qr`" register;
- creates empty circuits "`c0`" and "`c1`";
- iterates through the "`qr`" register, generating a *variant* "`qr-1-q`" and *invariants*: "`0 <= q <= qr - 1`", "`width !c0=n`" and "`width !c1=n`". Specification added in the iQbricks program is then included;
- creates an empty circuit "`c2`" for the *inner-loop* and iterates through the remaining indexes of the register. *Variant* "`(!qr_index + qr - 1) - i`" and *invariants*: "`(!qr_index + q + 1) <= i <= (!qr_index + qr - 1)`" and "`width !c2=n`" are generated, after which the specification added in the iQbricks program is included;
- sequences the *controlled Z-rotation* gate application (defined as "`crz`" in Qbricks) to circuit "`!c2`";
- after the *inner-loop*, "`!c2`" (which now contains the sequence of the *controlled Z-rotation* gates) is updated by placing an *Hadamard* gate at the beginning of this circuit;
- following the *Hadamard* gate assertions, "`!c1`" is sequenced with the updated "`!c2`" circuit and "`!c0`" is sequenced with the updated "`!c1`" circuit, after which "`!c1`" and "`!c2`" are cleared for the following *loop* iterations;
- after the *outer-loop*, "`!c0`" is returned and the program's *post-conditions* are included, after which the program ends.

```
let qft (qr n:int) : circuit
requires{0<n}
requires{qr=n}
requires{true}
=
begin
let qr_index = ref (0: int) in
let c0 = ref (m_skip n) in
let c1 = ref (m_skip n) in
for q = 0 to (qr - 1) do
```

```
   variant{qr - 1 - q}
   invariant{0 <= q <= qr - 1}
   invariant{width !c0=n}
   invariant{width !c1=n}
   invariant{range !c1 = q}
   invariant{forall x y i. 0<= i < n -> basis_ket !c1 x y i = if 0<= i < q then y i
       else x i}
   invariant{forall x y. ang_ind !c1 x y = (ind_isum(fun k -> (ind_isum (fun l -> x l
       * y k * power 2 (n-l - 1+k)) k n))0 q) /./ n}
   let c2 = ref (m_skip n) in
   for i = (!qr_index + q + 1) to ((!qr_index + qr - 1)) do
      variant{(!qr_index + qr - 1) - i}
      invariant{(!qr_index + q + 1) <= i <= (!qr_index + qr - 1)}
      invariant{width !c2=n}
      invariant{range !c2 = 0}
      invariant{forall x y i. 0<= i < n -> basis_ket !c2 x y i = x i}
      invariant{forall x y. ang_ind !c2 x y = (ind_isum (fun l -> x l * x q * power 2
          (n- l -1+ q)) (q+1) i) /./n}
      c2 := !c2 -- (crz ((!qr_index + i + 1)) ((!qr_index + q)) (i - q) n);
   done;
   c2 := (place_hadamard (!qr_index + q) n) -- !c2;
   assert{width !c2 = n};
   assert{forall x y: int->int. forall i:int. 0<=i<n ->basis_ket !c2 x y i = if i = q
       then y 0 else x i};
   assert{forall x y. ang_ind !c2 x y = (ind_isum (fun l -> x l * y 0 * power 2 (n-l -
        1+ q)) q n) /./ n};
   c1 := !c1 -- !c2;
   c2 := m_skip n;
   c0 := !c0 -- !c1;
   c1 := m_skip n;
done;
return !c0;
ensures{width result = n}
ensures{range result = n}
ensures{forall x y i. 0<= i < n -> basis_ket result x y i = y i}
ensures{forall x y. ang_ind result x y = (ind_isum(fun k -> (ind_isum (fun l -> x l *
    y k * power 2 (n-l - 1+k)) k n))0 n) /./ n}
end
```

Listing 5.12: QFT translation

5.3.3 *Results*

The generated QBRICKS program was tested against several *SMT solvers* using *Why3*. From the specified *QFT* program, *Why3* generates a set of *proof obligations* which, when satisfied, provide a *formally verified* quantum program implementing the *QFT* circuit.

Figure 22 illustrates the generated *proof-obligations* for the *QFT* program. The green "check-mark" on the left side of each *proof-obligation* signals that a proof was obtained.

Given that all the *proof-obligations* have been satisfied, the implementation of the *QFT* algorithm is considered to be formally verified. This verification provides strong evidence that the program behaves correctly according to its intended specification.

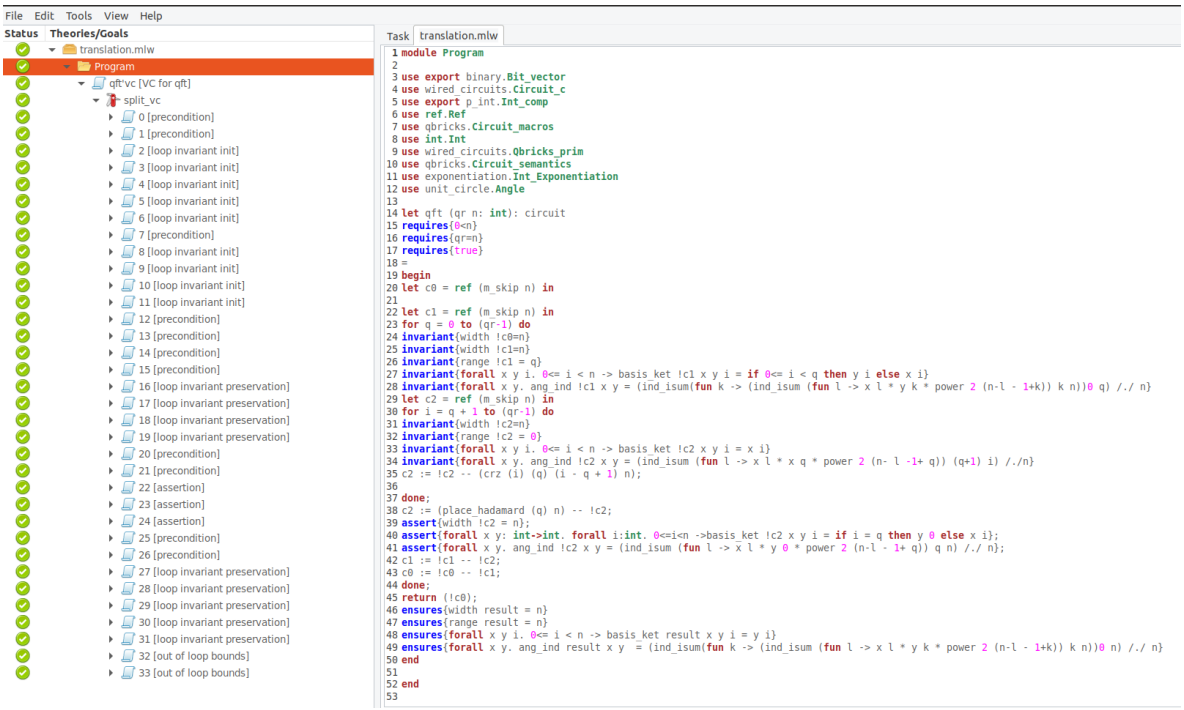


Figure 22: *QFT* program with generated *proof-obligations*

As done with the *Grover* circuit implementation, an *OpenQASM* circuit for the "`qft`" Qbricks program with parameters: `qr=4` and `n=4` was generated, and is shown in Figure 23.

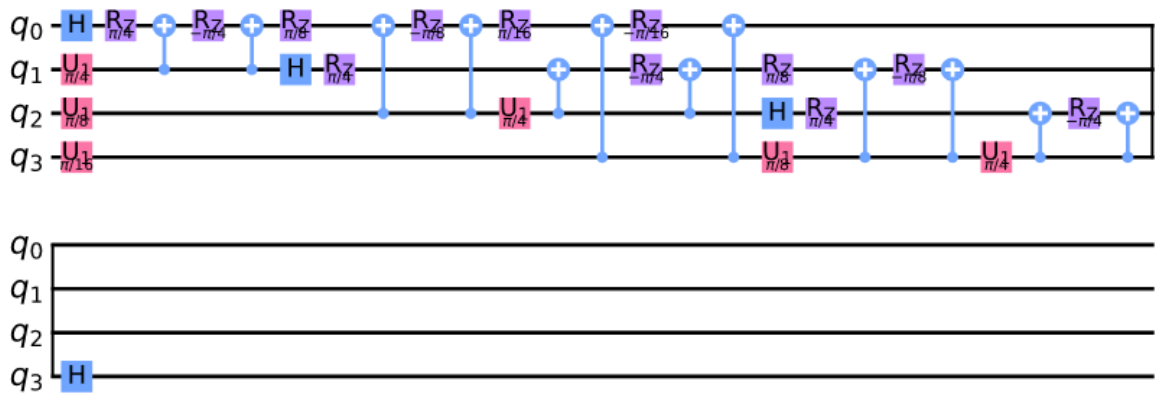


Figure 23: Generated *OpenQASM* `qft` circuit

We can thus conclude that the translation of Qbricks *QFT* implementation is successful, the generated quantum circuit matches the intended design and the included specification allows the program to be formally verified.

6

CONCLUSION

In conclusion, this project has made significant contributions to the field of quantum computing. By designing a new high-level programming language for quantum circuits, iQbricks, that translates to Qbricks, this project has made quantum programs more accessible and user-friendly. The introduction of an AST for quantum programs not only roots the connection between iQbricks and Qbricks, but also enables the connection of the circuit object language with a specification language. Finally, the developed framework can be easily expanded to other quantum programming languages, making this project a valuable contribution to the growing field of quantum computing.

6.1 PROSPECTS FOR FUTURE WORK

This work has opened up exciting possibilities in the field of quantum computing. Specifically the following are several potential areas for further exploration:

1. Development of a **source specification language**: There are two main tasks involved in this development: (1) designing the language and (2) parsing it to WhyML. The goal of this development is to provide users with a more user-friendly way of writing specifications and to automate some of the more repetitive proofs through static analysis and proof tactics.

2. Bridging the gap between quantum programs and other widely used languages: Using the *ML-AST* as an abstract structure for quantum programs, it may be possible to develop **bridges** that allow for **translations** back and forth between other commonly used programming **languages**. This would allow for the integration of verification frameworks with communities such as Qiskit and others. Additionally, the translation of iQbricks programs to Qiskit enables the use of existing simulators or even real quantum computers.

3. Extension of the object/specification language to **hybrid programming**: in particular incorporation of quantum measurement and classical control, this extension is crucial

for the integration of quantum computing into hybrid environments, such as those involving error correction, unitary decomposition, HPC integration, and pre-post processing.

These future prospects will be pursued in a PhD project, beginning in the near future.

BIBLIOGRAPHY

Matthew Amy. Towards large-scale functional verification of universal quantum circuits. *Electronic Proceedings in Theoretical Computer Science*, 287:1–21, jan 2019. doi: 10.4204/eptcs.287.1. URL https://doi.org/10.4204%2Feptcs.287.1.

Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

J. S. Bell. On the einstein podolsky rosen paradox. *Physics Physique Fizika*, 1:195–200, Nov 1964. doi: 10.1103/PhysicsPhysiqueFizika.1.195. URL https://link.aps.org/doi/10.1103/PhysicsPhysiqueFizika.1.195.

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386007. URL https://doi.org/10.1145/3385412.3386007.

Tomás Carneiro. iqbricks. https://github.com/tbc23/iQbricks, 2023.

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. A deductive verification framework for circuit-building quantum programs. *arXiv preprint arXiv:2003.05841*, 2020.

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 148–177, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72019-3.

Andrew Cross. The ibm q experience and qiskit open-source quantum computing software. In *APS March Meeting Abstracts*, volume 2018, pages L58–003, 2018.

Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. URL https://doi.org/10.48550/arXiv.1707.03429.

Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6/7):467–488, 1982.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.

Doug Finke. Quantum computing's time is coming. 2022. URL https://quantumcomputingreport.com/quantum-computings-time-is-coming/.

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.

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving quantum programs correct. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 21:1–21:19, 2021a. doi: 10.4230/LIPIcs.ITP.2021.21.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021b. doi: 10.1145/3434318.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, and Jérôme Vouillon. The ocaml system release 4.14 documentation and user's manual, 2022. URL https://v2.ocaml.org/releases/4.14/manual/index.html.

Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi: 10.1017/CBO9780511976667.

Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172. URL http://dx.doi.org/10.1137/S0097539795293172.

Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020.