Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Static Analysis of Resources in QASM: Estimation of the Number of Qubits

Relatore:	**Presentata da:**
Chiar.mo Prof.	**Damiano Scevola**
Ugo Dal Lago	

Sessione I
Anno Accademico 2020/2021

A nonno Francesco

Contents

Chapter 1

Introduction

The idea of quantum computing was conceived in the 1970s, when researchers of the caliber of Feynman, Manin and Deustch started to theorize that the properties of quantum mechanics might prove useful for performing computations differently from the classical paradigm [**zuw**]. The main concept that lays at the core of this new model is the superposition of states, that is the simultaneous co-existence of multiple computational states at the same moment, with the possibility to perform the same operation to all of those states with a single computational step. Physical devices that are capable of such power are called "quantum computers", and they constitute a polynomial-time implementation of non-determinism, thus achieving an exponential-order level of parallelism. This fact invalidates the assumption that some problems (NP-complete ones, for instance) require too much time to be solved compared to human life duration, hence it opens the doors to a new variety of applications, and it also tears down the ones which rely on this assumption (like RSA cryptography). Literature usually refers to this scenario as "Quantum Supremacy".
Quantum computers operate through qubits, which are the quantum equivalent of bits, and can have not only 0s and 1s as values, but also combinations of them. Qubits are sequenced together to form quantum registers, which hold the actual computational power of quantum computers. For instance, a 3-qubit register can be in a superposition of the states $|000\rangle, |001\rangle, |010\rangle, \ldots, |111\rangle$, that are $2^3 = 8$ base states. Generalizing the previous result, we notice that an n-qubit register can represent up to 2^n states simultaneously, thus showing where the exponential speedup resides.
Some companies like Google and IBM have already built fully-functional quantum computers. However, all of them have a limited number of qubits due to the elevate cost of implementing them physically. Therefore, it is crucial that quantum programmers are thrifty with the usage of qubits while designing quantum circuits. The aim of this document is to show a simple tool which is able to analyze programs written in QASM language (Quantum Assembly) and check whether the number of qubits that are *effectively* used is bounded by a certain expression. We start by introducing some concepts

of quantum computing, then we describe the QASM language and we provide details about the features that are taken into account by the analyzer. Finally we describe the tool that performs the estimation of the number of qubits by presenting the modules APIs it is built upon one by one, and in the last chapter we discuss how the estimation of the number of qubits is actually performed, going through some examples.

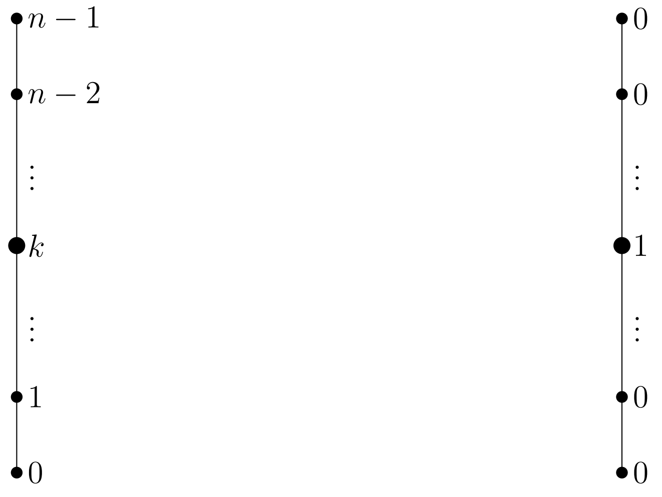
Chapter 2

Quantum Computing

2.1 Quantum states

2.1.1 Deterministic systems

In classical systems, when we need to encode the state of an object, we often use numbers. For example, consider a simple case where we have a skyscraper with n floors (counting the ground floor also), and we want to represent on which one an employee is located. Trivially, this can be done using an integer number k, and if we assume there are no underground floors, then we can state $k \in \{0, 1, \dots, n-1\}$.
Another option for representing such a state would be using a *state vector*, that is a vector $\mathbf{v} \in \{0, 1\}^n$ whose entries are all zeros except for the k-th, which is a 1.



Such an approach seems exaggerated for this simple case, but it is crucial to understand how quantum systems work. Let's take a look at another scenario.

2.1.2 Probabilistic systems

Imagine now that the floor the employee is on is not known for sure, but we are only given probabilities of her being on each floor. The state vector now looks something like:

$$\mathbf{v} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} \in [0,1]^n$$

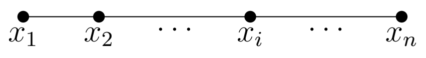
We now have a generalized version of our previously defined state vector, because each entry p_i is a real number between 0 and 1. Since we are talking about probabilities, we set the constraint that all the entries of $\mathbf{v}$ must sum up to 1.

$$\sum_{i=0}^{n-1} p_i = 1$$

It should be noticed that there is no way to represent probabilistic states with a single number like we do for deterministic ones. It is clear, however, that in case we know the exact location of the employee, the vector would collapse to the previous case. In fact, if we are certain that the employee is on the k-th floor, then $p_k = 1$, and $\forall i \in \{0, 1, \dots, n-1\} \setminus \{k\},\ p_i = 0$.
Now it is time to define state vectors for quantum systems.

2.1.3 Quantum systems

Consider a quantum object (e.g. an electron or a photon), and assume its position can be measured in the domain $\{x_1, x_2, \dots, x_n\}$.



To each position x_i, it corresponds a basic vector having as entries all zeros and a 1 as the i-th entry.

$$|x_1\rangle = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},\ |x_2\rangle = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix},\ \dots,\ |x_n\rangle = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Definition 2.1.1 (Quantum State)**.** A quantum state $|\psi\rangle$ is a linear combination of the basic vectors $|x_1\rangle, |x_2\rangle, \dots, |x_n\rangle$ having complex weights $c_1, c_2, \dots, c_n \in \mathbb{C}$.

$$|\psi\rangle = c_1 |x_1\rangle + c_2 |x_2\rangle + \cdots + c_n |x_n\rangle$$

Therefore, $|\psi\rangle$ can be represented as

$$|\psi\rangle = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} \in \mathbb{C}^n$$

where

$$\sum_{i=0}^{n-1} |c_i|^2 = 1$$

Since it is a weighted sum of basic states, we say that $|\psi\rangle$ is a *superposition* of the basic states. The square norm of c_i tells us the probability of observing the particle in the position x_i, and since the particle is indeed in one of those positions, the sum of square norms must add up to 1.
The reason why we need complex numbers to represent quantum states is that quantum objects (like photons or electrons) behave both like particles and waves, so they can interfere either constructively or disruptively. By using complex numbers, we can sum waves and capture the fact that they could interfere disruptively, since the square norm of the sum of two complex number is not necessarily greater than both of the square norms of the addends.

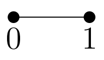
2.2 Qubits and quantum registers

2.2.1 Bits and qubits

In order to understand what qubits are, we should remark the meaning of classical bits and then make a generalization.

Definition 2.2.1 (Bit)**.** A bit is an elementary unit of information describing the state of the simplest classical system.

As we know, a bit can only assume values in the domain $\{0, 1\}$.



By following the approach of the previous section, we can also describe bits using a two-dimensional state vector, where the first element stands for 0, and the last for 1:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

If we take $|0\rangle$ and $|1\rangle$ as basic states, we can define a quantum system by considering all normalized linear combinations of those states. Such a quantum system is called *qubit.*

Definition 2.2.2 (Qubit)**.** A qubit is a quantum system having $|0\rangle$ and $|1\rangle$ as basic states, and therefore it can be represented as an element of $\mathbb{C}^2$.

$$|q\rangle = c_0 |0\rangle + c_1 |1\rangle = c_0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \in \mathbb{C}^2$$

Keep in mind that the sum of the square norms must be equal to 1. In other words, the state must be *normalized.*

$$|c_0|^2 + |c_1|^2 = 1$$

Example 2.2.1. Here is an example of a qubit:

$$|q\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

In fact, if we check the sum of the square norms of the coefficients, we see that they add up to 1.

$$\left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} + \frac{1}{2} = 1$$

Of course, the following are also qubits:

$$|q_0\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \; |q_1\rangle = \frac{|1\rangle - |0\rangle}{\sqrt{2}} = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

We shall notice that $|q\rangle$, $|q_0\rangle$ and $|q_1\rangle$ are all superpositions of the basic states $|0\rangle$ and $|1\rangle$.

2.2.2 Measurement

When we measure a qubit, the possible outcomes are only 0 and 1, and the probability of observing the one or the other is dictated by the square norms of the coefficients c_0 and c_1. In addition to this, the measurement affects the internal state of the qubit permanently, making it collapse to the value that has just been measured.

Example 2.2.2. Consider a qubit whose state is

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

If we measure it, we have $\left(\frac{\sqrt{3}}{2}\right)^2 = \frac{3}{4} = 75\%$ of probability to observe the outcome being 0, and $\left(\frac{1}{2}\right)^2 = \frac{1}{4} = 25\%$ of probability of it being 1.
Let's say we obtained 0 from the measurement. The state of the qubit is not $|\psi\rangle$ anymore, but it has become $|\psi'\rangle = |0\rangle$ because the measurement made it collapse as a side effect, so if we measure it again we have 100% of chances to observe the value 0 again.
This fact is at the core of quantum mechanics, and it just tells us that in order to observe something we must inevitably interact with it and alter its internal state.

2.2.3 Quantum registers

Quantum computers with a single qubit are not very interesting, so we need a way to combine qubits and form registers. In order to combine quantum systems, we need to use the tensor product. For example, let's say we wanted to represent the two-qubits state $|01\rangle$. We would then need to compute the tensor product $|0\rangle \otimes |1\rangle$ as follows:

$$|0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\begin{bmatrix}0\\1\end{bmatrix}\\0\begin{bmatrix}0\\1\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}c_{00}\\c_{01}\\c_{10}\\c_{11}\end{bmatrix}$$

If we observe the result carefully, we can notice that the normalization constraint still holds, and the only 1 that appears in the final result corresponds to c_{01}, which is right the coefficient labelled with our state $|01\rangle$. If we try to do the same thing with the state $|101\rangle$, we will obtain:

$$|101\rangle = |1\rangle \otimes |0\rangle \otimes |1\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}c_{000}\\c_{001}\\c_{010}\\c_{011}\\c_{100}\\c_{101}\\c_{110}\\c_{111}\end{bmatrix}$$

Since the result of the tensor product enumerates all possible states with n qubits, the dimension of the output is as large as 2^n.
Of course, considering the fact that quantum computers allow superpositions of many states, we shall clarify that the previous examples have only taken into account basic states. Hence we can also have a 3-qubit register in the state

$$|\psi\rangle = \frac{|000\rangle + |010\rangle - |011\rangle - |100\rangle + |110\rangle}{\sqrt{5}}$$

2.2.4 Entanglement

When multiple qubits are involved to form a quantum register, it can sometimes have the property that if we measure one of the qubits, then we automatically know the value of the others. We say, then, that those qubits are *entangled.*

Example 2.2.3. Consider a quantum register with 2 qubits and assume its state is

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Then, if we measure the first qubit and we observe the value 1, we automatically know that the other qubit also has value 1 by just noticing that the only basic state that appears in $|\psi\rangle$ and has a $|1\rangle$ at the first position is $|11\rangle$, so we know for sure that the other one is $|1\rangle$ too.
In this case, the measurement not only affects the state of the measured qubit making it collapse to $|1\rangle$, but it also collapses the state of the entangled one.

2.3 Quantum Gates

So far we have discussed how to represent qubits and quantum registers and what their properties are, but in order to perform a computation we need a way to manipulate them according to an algorithm and exploit their computational power.
Just like classical computers manipulate bits using logic gates, quantum computers use quantum gates to achieve their objective.
The physical implementation of quantum gates is as complicated as that of qubits, but we can represent what these gates do by using mathematical objects that operate on state vectors and keep the output normalized so that the sum of the square norms of the coefficients of the output vector is always 1. These objects are unitary matrices.
There are infinite possible quantum gates. Here we describe the set of universal ones $\{H, CNOT, R(\theta)\}$, which includes three gates that can be combined together to simulate an arbitrary gate.

2.3.1 Hadamard gate

The Hadamard gate is encoded by the following matrix:

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

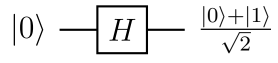
The Hadamard matrix is the transition matrix from the canonical basis $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$ to the Hadamard basis $\left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \right\}$, which is used to express arbitrary states in terms of superpositions of basic states, and this allows calculations to be performed on a superposition, hence on all of the basic states it is composed of with a single computational step.
Since H is unitary, by applying it twice we go back to the canonical basis, so after calculations are finished, Hadamard gate can be applied again to translate the results back into the canonical basis, ready to be measured by the observer.

Example 2.3.1. By applying Hadamard gate to the qubits $|0\rangle$ and $|1\rangle$ we get:

$$H|0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle$$

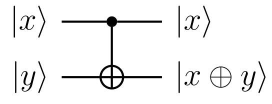
$$H|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

It is easy to verify that applying Hadamard on $|+\rangle$ and $|-\rangle$ we go back respectively to $|0\rangle$ and $|1\rangle$.
We will represent the Hadamard gate as a box labelled with 'H'



2.3.2 Controlled NOT gate

While Hadamard gate affects a single qubit, the controlled NOT gate (CNOT or CX for short) needs two of them. One is the control qubit, and the other is the target.



Here $\oplus$ denotes the binary exclusive OR operation. That is, if $|x\rangle = |0\rangle$ then the bottom output will be left unaltered, otherwise if $|x\rangle = |1\rangle$, then the bottom qubit is flipped. The matrix that corresponds to the CNOT gate is:

$$^{C}X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2.3.3 Phase shift gate

Phase shift gate is a parametric gate that affects a single qubit by rotating it in the Bloch sphere, adjusting its phase. We will not go over the details of Bloch sphere representation, but we just give the matrix which encodes the phase shift here.

$$R(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\theta} \end{bmatrix}$$

Now that we have discussed quantum gates, we can move to designing quantum circuits (or algorithms).

2.4 Quantum Circuits

Quantum circuits are the equivalent of algorithms for classical computing. By combining quantum gates, we can build a circuit that performs a certain computation and exploits the power of superpositions to drastically reduce the time of execution compared to a classical program.

Definition 2.4.1 (Quantum circuit)**.** A quantum circuit is a computational routine consisting of coherent quantum operations on quantum data, such as qubits, and concurrent real-time classical computation [**qsk**]. It is an ordered sequence of quantum gates, measurements and resets, all of which may be conditioned on and use data from the real-time classical computation.

In order to understand how quantum circuits work, we will discuss the Deutsch algorithm for balanced and constant function classification.

2.4.1 Deutsch algorithm

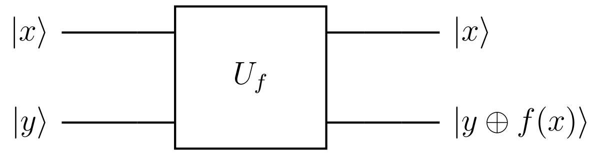
Before describing the Deutsch algorithm, let's define what balanced and constant functions are.

Definition 2.4.2 (Balanced and constant functions)**.** A function $f : \{0,1\} \longrightarrow \{0,1\}$ is *balanced* if $f(0) \neq f(1)$, otherwise it is *constant*.

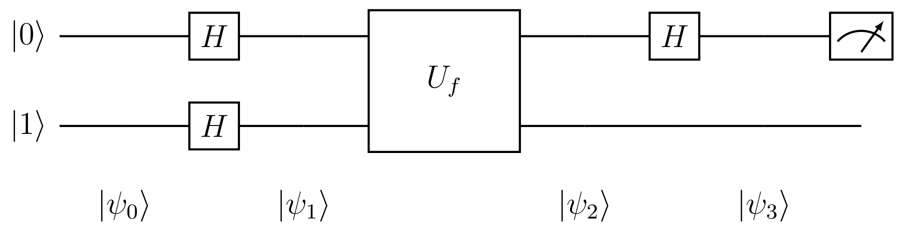
The Deutch algorithm solves the following problem:

Problem 2.4.1 (Deutsch-Jozsa)**.** Suppose we have a function $f : \{0,1\} \longrightarrow \{0,1\}$ which we can evaluate, but we do not know its exact definition. Determine whether f is balanced or constant.

If we wanted to use the classical approach, we would need to evaluate f twice and compare the results. With the Deutsch algorithm, we will see that only one evaluation is sufficient.
Suppose we have a unitary matrix U_f that encodes f:



Let's now build a circuit that uses U_f and solves our problem:



In terms of matrices, the circuit corresponds to

$$(H \otimes I)U_f(H \otimes H)|0,1\rangle$$

The algorithm starts with the state

$$|\psi_0\rangle = |0,1\rangle$$

Then it proceeds by applying Hadamard to both qubits, putting them in a superposition:

$$|\psi_1\rangle = |+,-\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] = \frac{|0,0\rangle - |0,1\rangle + |1,0\rangle - |1,1\rangle}{2}$$

Now let's pause for a moment and ponder about what happens if we apply U_f to a generic state $|x,-\rangle$.

$$U_f|x,-\rangle = U_f(|x\rangle\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right]) = U_f\left[\frac{|x,0\rangle - |x,1\rangle}{\sqrt{2}}\right] = |x\rangle\left[\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}}\right]$$

Which simplifies to

$$U_f|x,-\rangle = |x\rangle\left[\frac{|f(x)\rangle - \left|\overline{f(x)}\right\rangle}{\sqrt{2}}\right]$$

Where we denoted the opposite of $f(x)$ with $\overline{f(x)}$. So we have

$$U_f|x,-\rangle = \begin{cases} |x\rangle\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & if\ f(x) = 0 \\ |x\rangle\left[\frac{|1\rangle - |0\rangle}{\sqrt{2}}\right] & if\ f(x) = 1 \end{cases}$$

We can write the last expression as

$$U_f|x,-\rangle = (-1)^{f(x)}|x\rangle\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] = (-1)^{f(x)}|x\rangle|-\rangle$$

Going back to our circuit, this means that, after applying U_f to $|\psi_1\rangle$, we get

$$\begin{aligned}|\psi_2\rangle = U_f|+,-\rangle &= \frac{U_f|0,-\rangle + U_f|1,-\rangle}{\sqrt{2}} \\ &= \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}}|-\rangle\end{aligned}$$

Now let's see what happens in the two cases where f is balanced or it is constant. If f is balanced $f(0) \neq f(1)$, then the top qubit becomes either $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ or $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$ depending on which way it is balanced, but in either case we can write

$$|\psi_2\rangle = (\pm 1)\frac{|0\rangle - |1\rangle}{\sqrt{2}}|-\rangle = (\pm 1)|-,-\rangle$$

Since the coefficient (± 1) is global, it can be ignored due to the normalization constraint, so we can say $|\psi_2\rangle = |-,-\rangle$ if f is balanced.
On the other hand, if f is constant $f(0) = f(1)$, then the top qubit becomes either $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ or $\frac{-|0\rangle-|1\rangle}{\sqrt{2}}$, but in either case we can write

$$|\psi_2\rangle = (\pm 1)\frac{|0\rangle + |1\rangle}{\sqrt{2}}|-\rangle = (\pm 1)|+,-\rangle$$

Here the argument for (± 1) still holds, so we can say $|\psi_2\rangle = |+,-\rangle$ if f is constant.
Summing up, we have

$$|\psi_2\rangle = \begin{cases}|+,-\rangle & if\ f\ is\ constant \\ |-,-\rangle & if\ f\ is\ balanced\end{cases}$$

Finally, the last Hadamard gate brings our state back to the canonical basis, and so

$$|\psi_3\rangle = \begin{cases}|0,-\rangle & if\ f\ is\ constant \\ |1,-\rangle & if\ f\ is\ balanced\end{cases}$$

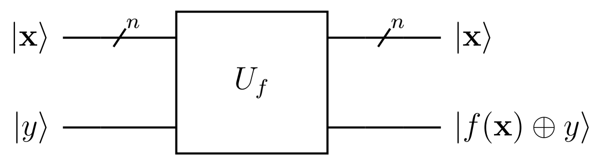
Now we can measure the top qubit, and we can be sure about the function being balanced or constant by just looking at the result.
Notice that the quantum algorithm only evaluated the function once, namely on the superposition $|+,-\rangle$. The generalized version of this algorithm, the Deutsch-Jozsa algorithm, shows the speedup in the computation even more.

2.4.2 Deutsch-Jozsa algorithm

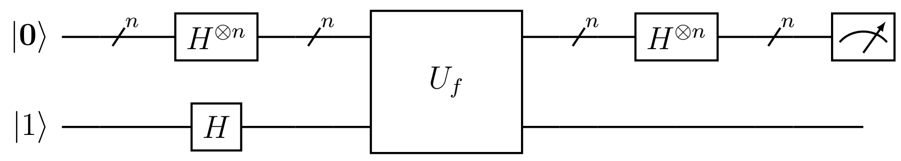
Although we are not going to illustrate all the analysis of the Deutsch-Jozsa algorithm, we will still define the problem, show the circuit that solves it and state some facts about its computational complexity compared to the classical case. Let's generalize the definition of balanced and constant functions.

Definition 2.4.3 (Balanced and constant functions)**.** A function $f : \{0,1\}^n \longrightarrow \{0,1\}$ is *balanced* if exactly half of the inputs go to 0 (and the other half go to 1), while it is *constant* if all of the inputs go to the same output (either 0 or 1).

The Deutsch-Jozsa problem is the same as the Deutsch one, with the assumptions that $f : \{0,1\}^n \longrightarrow \{0,1\}$ and that f is guaranteed to be either constant or balanced. With the classical approach, in order to determine whether f is balanced or constant we need to evaluate it at most on one plus half of the inputs. This is because as soon as we detect two different output values, we can conclude that the function is balanced, but if f is constant, we cannot be sure of it until we check that more than half of its outputs are the same. So the worst case scenario requires $1 + \frac{2^n}{2} = 1 + 2^{n-1}$ evaluations, which is exponential with respect to the size of the input.
Let's now delve into the quantum world and contemplate its power.
Since we are given the possibility to evaluate f, assume there is a unitary matrix U_f that fulfills the task:



Here we denoted the top input $|\mathbf{x}\rangle = |x_0x_1 \dots x_{n-1}\rangle$ in bold and the wire mark $\overset{n}{\not-}$ indicates that it is a bit sting with length n. The circuit that solves the problem is the following:



We can notice that it is very similar to the Deutsch circuit, with the only difference that there are n qubits as the top input, and we denoted the fact that we are applying Hadamard to each one of them with the gate $H^{\otimes n}$, that is the tensor product of n Hadamard matrices.
This circuit is capable to tell us if f is balanced or constant in a single run. In other words, we have an exponential speedup compared to the classical algorithm for the same problem. There are many other problems that benefit of the quantum speedup, and some of them are of a practical nature. This is why quantum computing is so powerful.
In order to design and execute quantum algorithms, we need a language that allows us to describe them in terms of gates and classical instructions. In the next chapter we will explore OpenQASM, which is a powerful and compact language that achieves the goal.

Chapter 3

OpenQASM

OpenQASM is an imperative programming language designed for quantum algorithms and applications. It has a dual nature as an assembly language and as a hardware description language [**qasm**]. There are multiple versions of OpenQASM, but we will describe a simplified version of OpenQASM 3.0.
The formal grammar of the language is presented using the ANTLRv4 syntax, and starts with:

```
program
    : header (globalStatement | statement)*
    ;
```

So we first describe the header and then we proceed to discuss the other aspects of the language, which also include statements. In each section we will present the formal grammar and comment it providing informal semantic and context.

3.1 Header

```
header
    : version? include*
    ;

version
    : 'OPENQASM' (Integer | RealNumber) SEMICOLON
    ;

include
    : 'include' StringLiteral SEMICOLON
    ;
```

The header consists in an optional version string followed by zero or more ‘include’ statements, that are used to import gates and functions from other files at compile time. A typical example of what the header looks like is:

```
// Header
OPENQASM 3.0;
/*
comment
*/
include "stdgates.qasm";
```

We can see here that comments are C-like, and string literals are wrapped by double quotes.
From now on we will proceed in a bottom-up fashion, describing all the features of the language that are used in more complex rules, starting from lexicon, types and index identifiers up to global and non-global statements.

3.2 Lexer rules

```
LBRACKET: '[';
RBRACKET: ']';

LBRACE: '{';
RBRACE: '}';

LPAREN: '(';
RPAREN: ')';

COLON: ':';
SEMICOLON: ';';

DOT: '.';
COMMA: ',';

EQUALS: '=';
ARROW: '->';

PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
```

```
Constant: ( 'pi' | 'tau' | 'euler' );

Whitespace: [ \t]+ -> skip;
Newline: [\r\n]+ -> skip;

fragment Digit: [0-9];
Integer: Digit+;

fragment ValidUnicode: [\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}];
fragment Letter: [A-Za-z];
fragment FirstIdCharacter: '_' | '$' | ValidUnicode | Letter;
fragment GeneralIdCharacter: FirstIdCharacter | Integer;

Identifier: FirstIdCharacter GeneralIdCharacter*;

fragment SciNotation: [eE];
fragment PlusMinus: PLUS | MINUS;
fragment Float: Digit+ DOT Digit*;
RealNumber: Float (SciNotation PlusMinus? Integer)?;

LineComment: '//' ~[\r\n]* -> skip;
BlockComment: '/*' .*? '*/' -> skip;

StringLiteral
    : '"' ~["\r\t\n]+? '"'
    | '\'' ~['\r\t\n]+? '\''
    ;
```

3.3 Types

In OpenQASM, types are divided in two categories: quantum types and classical types.

3.3.1 Quantum types

```
quantumType
    : 'qubit'
    | 'qreg'
    ;
```

We will describe how to make a quantum declaration in the 'Global statements' section.

3.3.2 Classical types

```
classicalType
```

```
    : singleDesignatorType designator
    | doubleDesignatorType doubleDesignator
    | noDesignatorType
    | bitType designator?
    ;

bitType
    : 'bit'
    | 'creg'
    ;

singleDesignatorType
    : 'int'
    | 'uint'
    | 'float'
    | 'angle'
    ;

doubleDesignatorType
    : 'fixed'
    ;

noDesignatorType
    : 'bool'
    ;

designator
    : LBRACKET expression RBRACKET
    ;

doubleDesignator
    : LBRACKET expression COMMA expression RBRACKET
    ;
```

A `designator` is simply a parameter for the type, for example `int[16]` indicates a 16-bit integer, where `16` is the designator. The single designator for the `float` and `angle` types is for specifying the precision. Another example could be `fixed[7, 24]`, which indicates a $24 + 7 + 1 = 32$-bit fixed point signed number having 7 integer bits and 24 fractional bits.
`bitType` has a single designator that stands for the number of bits of the array (or register, equivalently), while only the boolean type has no designator.

3.4 Index identifiers

```
indexIdentifier
    : Identifier rangeDefinition
    | Identifier (LBRACKET expressionList RBRACKET)?
    | indexIdentifier '||' indexIdentifier
    ;

indexIdentifierList
    : (indexIdentifier COMMA)* indexIdentifier
    ;

indexEqualsAssignmentList
    : (indexIdentifier equalsExpression COMMA)* indexIdentifier
      equalsExpression
    ;

rangeDefinition
    : LBRACKET expression? COLON expression? (COLON expression)?
      RBRACKET
    ;
```

These rules are used to refer to specific bits or qubits in both classical and quantum registers. For better understanding, we provide some examples given two registers `x` and `y`, each having 8 bits:

- `x || y`: the 16-bit concatenation of `x` and `y`
- `x[0]`: the first bit of `x`
- `y[-1]`: the last bit of `y`
- `x[0,3,5]`: bits 0, 3 and 5 of `x`
- `x[0:6]`: the first 6 bits of `x`
- `y[0:2:7]`: every second bit of `y` from 0 to 7 (stops at 6)
- `y[-4:-1]`: the last three bits of `y`

3.5 Generical statements

```
statement
    : expressionStatement
    | assignmentStatement
    | classicalDeclarationStatement
    | branchingStatement
    | loopStatement
    | quantumStatement
    ;
```

In our simplified version of OpenQASM we consider six types of statement: expressions, assignments, classical declarations, branching if-then-elses, for and while loops and quantum statements. We now go over these statement types one by one, and we will be commenting only those that require further clarification, since most of them have similar syntax and semantic to other well-known programming languages.

3.5.1 Expression statements

```
expressionStatement
    : expression SEMICOLON
    ;

expressionList
    : (expression COMMA)* expression
    ;

expression
    : expressionTerminator
    | unaryExpression
    | xOrExpression
    | expression '|' xOrExpression
    ;

xOrExpression
    : bitAndExpression
    | xOrExpression '^' bitAndExpression
    ;

bitAndExpression
    : bitShiftExpression
    | bitAndExpression '&' bitShiftExpression
    ;

bitShiftExpression
    : additiveExpression
    | bitShiftExpression ('<<' | '>>') additiveExpression
    ;

additiveExpression
```

```
    : multiplicativeExpression
    | additiveExpression (PLUS | MINUS) multiplicativeExpression
    ;

multiplicativeExpression
    : expressionTerminator
    | unaryExpression
    | multiplicativeExpression (MUL | DIV | MOD) (
       expressionTerminator unaryExpression)
    ;

unaryExpression
    : unaryOperator expressionTerminator
    ;

expressionTerminator
    : Constant
    | Integer
    | RealNumber
    | Identifier
    | StringLiteral
    | MINUS expressionTerminator
    | LPAREN expression RPAREN
    | expressionTerminator LBRACKET expression RBRACKET
    | expressionTerminator incrementor
    ;

unaryOperator
    : '~' | '!'
    ;

incrementor
    : '++'
    | '--'
    ;
```

The grammar for expressions is defnined hierarchically to implement the conventional evaluation order of operators.
Expression statements do not affect the memory unless incrementors are present, in which case the respective variables are updated accordingly to the incrementor sign.
Even if they are not included in the `expressionStatement` rule, we show the grammar for boolean expressions here, because they will be used in other statement types from now on.

```
booleanExpression
    : membershipTest
    | comparisonExpression
    | booleanExpression logicalOperator comparisonExpression
    ;

comparisonExpression
    : expression  // if (expression)
    | expression relationalOperator expression
    ;

relationalOperator
    : '>'
    | '<'
    | '>='
    | '<='
    | '=='
    | '!='
    ;

logicalOperator
    : '&&'
    | '||'
    ;

membershipTest
    : Identifier 'in' setDeclaration
    ;

setDeclaration
    : LBRACE expressionList RBRACE
    | rangeDefinition
    | Identifier
    ;
```

3.5.2 Assignment statements

```
assignmentStatement: (classicalAssignment |
   quantumMeasurementAssignment) SEMICOLON;

classicalAssignment
    : indexIdentifier assignmentOperator (expression |
        indexIdentifier)
    ;

assignmentOperator
    : EQUALS
    | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '~=' | '^=' | '
        <<=' | '>>='
    ;

quantumMeasurement
    : 'measure' indexIdentifierList
    ;

quantumMeasurementAssignment
    : quantumMeasurement (ARROW indexIdentifierList)?
    | indexIdentifierList EQUALS quantumMeasurement
    ;
```

Assignments are defined in the usual way, with the addition of quantum measurements, that simply tell the hardware to perform a measurement on certain qubits and store the result in classical variables. As we explained in the previous chapter, we remark that after being measured, qubits collapse to the observed state, so measurements are usually done at the end of the algorithm.

3.5.3 Classical declarations

```
classicalDeclarationStatement
    : (classicalDeclaration | constantDeclaration) SEMICOLON
    ;

classicalDeclaration
    : singleDesignatorDeclaration
    | doubleDesignatorDeclaration
    | noDesignatorDeclaration
    | bitDeclaration
    ;

singleDesignatorDeclaration
    : singleDesignatorType designator (identifierList |
        equalsAssignmentList)
    ;

doubleDesignatorDeclaration
```

```
    : doubleDesignatorType doubleDesignator (identifierList |
       equalsAssignmentList)
    ;

noDesignatorDeclaration
    : noDesignatorType (identifierList | equalsAssignmentList)
    ;

bitDeclaration
    : bitType (indexIdentifierList | indexEqualsAssignmentList)
    ;

constantDeclaration
    : 'const' equalsAssignmentList
    ;
```

3.5.4 Branching statements

```
programBlock
    : statement
    | LBRACE statement* RBRACE
    ;

branchingStatement
    : 'if' LPAREN booleanExpression RPAREN programBlock ('else'
       programBlock)?
    ;
```

3.5.5 Loop statements

```
loopStatement: loopSignature programBlock;

loopSignature
    : 'for' membershipTest
    | 'while' LPAREN booleanExpression RPAREN
    ;
```

3.5.6 Quantum statements

```
quantumStatement
    : quantumInstruction SEMICOLON
    ;

quantumInstruction
```

```
    : quantumGateCall
    | quantumMeasurement
    ;

quantumGateCall
    : quantumGateName (LPAREN expressionList? RPAREN)?
      indexIdentifierList
    ;

quantumGateName
    : 'CX'
    | 'U'
    | 'reset'
    | Identifier
    ;
```

Here it is worth mentioning that quantum gates can be called on multiple qubits, and this is the only possible cause of entanglement.

3.6 Global statements

After the header, the program consists in a sequence of statements and global statements. Global statements are defined by the rule:

```
globalStatement
    : subroutineDefinition
    | quantumGateDefinition
    | quantumDeclarationStatement
    ;
```

3.6.1 Subroutine definition

```
subroutineDefinition
    : ('#qubits' expression)? 'def' Identifier ( LPAREN
      classicalArgumentList? RPAREN )? quantumArgumentList
      returnSignature? subroutineBlock
    ;

classicalArgumentList
    : (classicalArgument COMMA)* classicalArgument
    ;

classicalArgument
```

```
    : classicalType association
    ;

quantumArgumentList
    : (quantumArgument COMMA)* quantumArgument
    ;

quantumArgument
    : quantumType designator? association
    ;

association
    : COLON Identifier
    ;

returnStatement: 'return' statement;

returnSignature
    : ARROW classicalType
    ;

subroutineBlock
    : LBRACE statement* returnStatement? RBRACE
    ;
```

Subroutines have two sets of arguments: classical arguments and quantum arguments. They also have an optional return signature, which is used to specify the return type of the function. Before the `def` keyword, the programmer can annotate the expression of the upper bound on the number of qubit that she wishes to assert, so that the analyzer can compare it to the estimated number of effectively used qubits. Here is an example of a subroutine definition:

```
#qubits 4
def f2 (int[5]:x) qreg[5]:q, qreg[5]:u -> int[8] {
    int[5] r;
    if (x * x < 0) {
        r = measure q;
    } else {
        CX q[0:3], u[1:4];
        r[0:2] = measure q[2:4];
    }
    int[5] s = r + x;
    return s;
}
```

Notice that the above example obviously passes the qubits check, since the branch where `x * x < 0` is never taken.

3.6.2 Quantum gate definition

We will not delve into the actual grammar of quantum gate definitions, since the analysis that we are going to discuss in the next chapters does not take them into account. In fact, our analyzer treats them as black-boxes that entangle all the input qubits together.

3.6.3 Quantum declaration statements

```
quantumDeclarationStatement: quantumDeclaration SEMICOLON;

quantumDeclaration
    : quantumType indexIdentifierList
    ;
```

As we can notice, a quantum declaration is simply the declaration of a qubit (or an array of them). It must be done globally, since qubits cannot be allocated dynamically.

Having clear how the language is defined, we can now discuss the tool that performs analysis on QASM programs and estimates the number of qubits.

Chapter 4

QASM Analyzer

QASM Analyzer is a tool written in Python 3 that performs static analysis on QASM programs, more specifically it classifies all the subroutines (based on the presence of loops, recursive calls, branching statements and so on) and simulates their execution by substituting formal parameters with immutable symbols, that are then involved in assignments, branching conditions and loop guards. By invoking an SMT solver, the analyzer determines all the paths that the execution will *effectively* take, and estimates the number of qubits that are *effectively* used. We remark the word "*effectively*" because there may be regions of code that are never executed, like an unsatisfiable condition in an if-then-else statement.
The source code is available on GitHub [**lusvelt**] and it is split into multiple modules:

- **Parser**: takes the QASM program as input and outputs a parse tree.
- **Types and variables**: utility classes to manage types, variables and subroutine arguments
- **Subroutine classifier**: navigates the parse tree and builds a data structure containing information about all the subroutines, each wrapped by an object of class 'Subroutine'.
- **Registers and ranges**: utility classes, helpful for working with arrays of bits or qubits
- **Expression manipulator**: takes an expression node of the parse tree and builds an Abstract Syntax Tree, which is then used to evaluate that expression in a certain context.
- **SMT solver**: takes a symbolic boolean expression as input and checks whether it is satisfiable or not.
- **Symbolic execution engine**: takes a subroutine object and builds a symbolic execution tree by simulating the instructions of that subroutine. At the end of the

simulation it checks if each branch of the execution does not exceed the number of qubits declared by the programmer as annotation in the QASM program.

In this chapter we provide details about each of them and explain how symbolic execution works.

4.1 Parser

The module we start with is the parser, since it provides the data structure that will be used throughout all the other modules: the parse tree.
Having defined the grammar of the language in the `qasm3sub.g4` file, we can invoke the ANTLRv4 tool [**antlr**] to generate the actual parser:

```
$ antlr4 -Dlanguage=Python3 qasm3sub.g4
```

This command generates a series of files which are stored in the `qasm3sub` folder. One of these files contains the `Listener` class, that allows us to navigate the raw parse tree built by ANTLRv4 and pick only the information that we need to build a simplified version of the parse tree, which we will be using in the other modules of the analyzer.
Our parse tree is composed by interconnected instances of the `Node` class.

4.1.1 Node

This class represents the core of the parse tree, and it encapsulates a grammar rule.

Attributes

- `type`: the name of the grammar rule (e.g. 'assignmentStatement', 'classicalDeclaration'), *None* if it is a lexer token.
- `children`: an array containing all the nodes obtained by expanding the rule with its actual production.
- `parent`: the node whose rule the current rule is obtained from.
- `text`: the space-trimmed string of the input program corresponding to the current rule
- `rules`: a dictionary that maps a rule type with the (ordered) list of all child nodes matching that type
- `position`: the index of the current Node in `parent.children` list
- `index`: the index of the current node in `parent.rules[self.type]`

Methods

- `__init__(type, text)`: sets the `type` and `text` attributes
- `appendChild(child)`: used internally for the ANTLRv4 tree walker to build the custom parse tree.
- `getChildByType(type, index=0)`: returns the `index`-th child matching `type`.
- `getChildrenByType(type)`: returns the array of all children matching `type`.
- `getChild(index=0)`: returns the `index`-th child.
- `getDescendantsByType(type)`: returns an array containing all the nodes matching the `type` which appear in the subtree using DFS visit.
- `getLastChild()`: returns the last child in the `children` array.
- `hasChildren()`: returns whether the current node has any child or not.

The parse tree root node can be obtained by calling the static function passing the program path string as argument:

```
Parser.buildParseTree(filePath)
```

4.2 Types and variables

This is an utility module for encapsulating all the information regarding types and variables. We describe these classes providing details about what their attributes and methods are.

4.2.1 ClassicalType

This class represents a classical type as defined in the QASM grammar, with zero, one or two designator expressions.

Attributes

- `node`: the parse tree node which the type comes from
- `typeLiteral`: the string literal of the type, it can be one of 'bit', 'creg', 'int', 'uint', 'float', 'angle' and 'fixed'.
- `designatorExpr1`: the expression node of the first designator (*None* if it is a no-designator type).
- `designatorExpr2`: the expression node of the second designator (if present, None otherwise).

Methods

- `__init__(typeLiteral, designatorExpr1, dedsignatorExpr2, node)`: if `node` is not *None*, instantiates a `ClassicalType` and reads the information in the `node` to fill the other attributes; otherwise they must explicitly be passed as arguments.
- `hasLimitedDomain()`: returns a boolean value indicating if the current type has a limited ('bit', 'creg' or 'bool') or unlimited domain.
- `getTypeForSolver()`: returns the mathematical numerical set of the current type ('Bool', 'Integer', 'Real' or 'BitVector') in order to allow the SMT solver to evaluate satisfiability according to the right domains.

4.2.2 Symbol

Symbolic expressions (which are different from our parse expression trees) are managed by the `sympy` library, which allows us to define symbols and automatically performs simplifications on expressions involving them.
This class acts as a static symbol provider, and has both static attributes and static methods.

Attributes

- `nextIndex`: since new symbols can be defined at any time, they are given an incremental index in order to make them unique. This attribute keeps track of what the index of the next symbol should be.
- `symbolTypes`: a dictionary that associates each symbol to its classical type.

Methods

- `getNewSymbol(type)`: invokes `sympy` to create and return another symbol, whose label is the character `$` concatenated to the `nextIndex` stringified value (`$0`, `$1`, `$2`, ...). Additionally, it stores the newly-created symbol type in the `symbolTypes` dictionary.
- `getSymbolType(label)`: returns the classical type associated to the symbol matching `label`.

4.2.3 Variable

This class represents a classical variable in an expression AST (which will be described in the next section).

Attributes

- `identifier`: the variable identifier.
- `type`: the variable classical type.

Methods

- `__init__(identifier, type)`: instantiates a new variable named `identifier` encapsulating its `type`.

Child classes `ClassicalVariables(Variable)` and `QuantumVariable(Variable)` are also defined, with the only abstraction that the former has a constructor that accepts a `typeNode` argument and automatically calls the `ClassicalType` constructor to set the `type` attribute.

4.2.4 Value

This class represents a hard-coded value in an expression AST.

Attributes

- `typeLiteral`: a string representing the type of the value, which determines its domain.
- `value`: the actual value.

Methods

- `__init__(value, typeLiteral)`: instantiates an element having the specified `value` and `typeLiteral`. If the latter is *None*, it is automatically detected.

4.3 Subroutine classifier

This module has four classes: `ClassicalArgument`, `QuantumArgument`, `Subroutine` and `SubroutineClassifier`. The first two are extensions of `ClassicalVariable` and `QuantumVariable` respectively.

4.3.1 ClassicalArgument

This class extends `ClassicalVariable`, inheriting attributes and methods. The utility of this class resides merely on the constructor.

Methods

- `__init__(node)`: takes a 'classicalArgument' node and instantiates a classical variable based on the information that the node carries.
- `hasLimitedDomain()`: returns a boolean indicating whether the encapsulated argument has a limited domain or not.

4.3.2 QuantumArgument

This class extends `QuantumVariable`. Again, the utility resides on the constructor.

Methods

- `__init__(node)`: takes a 'quantumArgument' node and instantiates a quantum variable based on the information that the node carries.

4.3.3 Subroutine

This class holds all the information about a subroutine. By calling its constructor on a 'subroutine' node, it basically fills up all the attributes that tell us what the properties of the subroutine are.

Attributes

- `identifier`: the subroutine identifier string.
- `classicalArgument`: an array containing a `ClassicalArgument` instance for each classical argument of the subroutine.
- `quantumArgument`: an array containing a `QuantumArgument` instance for each quantum argument of the subroutine.
- `symbolicExecutionTree`: the root state of the symbolic execution tree (it will be discussed later).
- `qubitUpperBound`: the symbolic expression annotated by the programmer to check against the number of qubits in each branch of the execution tree.
- `qubitChecks`: an array of symbolic expressions and *True* values. The former ones represent violations of the upper bound on the number of qubits, while the latter ones represent successful checks. It is filled during the last stages of the symbolic execution.

Other than these, there are a bunch more of attributes wich are actually *properties* of the subroutine:

- `hasOnlyLimitedArgs`: a boolean value indicating if the only arguments that the subroutine has have a limited domain.
- `hasInfiniteDomainArgs`: the logical negation of `hasOnlyLimitedArgs`
- `returnType`: an instance of `ClassicalType`, indicating the return type of the subroutine. *None* if there is no return value.
- `hasBranchingStatements`: a boolean value indicating whether the subroutine contains any if-then-else statement in its body.
- `hasLoops`: indicates if there are any loops in the subroutine body.
- `hasSubroutineCalls`: indicates if there are any subroutine calls in the subroutine body.
- `isPlainSequential`: indicates if the subroutine has no statements other than assignments and expressions in its body.
- `hasOnlyBranchingStatements`: indicates if the subroutine has no other statement than assignment, expression and branching statements.
- `isRecursive`: indicates if the function makes any recursive call.

Methods

- `__init__(node, parseTree)`: takes a 'subroutineDefinition' node and the root parse tree node, and fills the `classicalArguments` and `quantumArguments` arrays, and all the *property* attributes with the right values. This last thing is accomplished by traversing the subroutine parse tree and checking for the existance of if-then-else statements, loops and subroutine calls.
- `addQubitCheck(qubitCheck)`: takes a symbolic expression representing an upper bound violation, or *True* and pushes it into the `qubitChecks` array.
- `respectsQubitBound()`: returns whether all the branches of the execution respect the upper bound on the number of qubits or not by checking if there are any non-*True* values in the `qubitChecks` array.

4.3.4 SubroutineClassifier

This class is simply a wrapper for `Subroutine`. Its constructor takes `parseTree` as only argument, it goes over all global statements and for each subroutine that it encounters it instantiates a `Subroutine` object, which automatically performs all the analysis. A dictionary attribute, namely `subroutines`, associates each subroutine object to its identifier, so that it is possible to retrieve information about a particular subroutine by simply looking for it in that dictionary.

4.4 Registers and ranges

This module contains multiple utility classes that are used to work with both classical and quantum registers, and to simplify array ranges manipulation.

4.4.1 BitRange

This class encapsulates a range of bits in a classical register `creg`, it has a `start` and an `end` expressions (which can be either numerical or symbolic), a `symbol` to represent it in symbolic expressions, and a `value`, which can also be *None* if not specified.

Attributes

- `creg`: the classical register that has the current range as content.
- `start`: the range's starting index with respect to the parent `creg`.
- `end`: the range's ending index with respect to the parent `creg`.
- `symbol`: a `sympy` symbol used to represent the current range.
- `value`: the actual value of the range (either numerical or symbolic).

Methods

- `__init__(creg, start, end, symbol)`: sets the range attributes according to the parameters.

4.4.2 Bit

A bit is a particular case of bit range, where the start and end expression coincide. So this class inherits all from `BitRange` and its constructor makes sure that the range is one bit long.

Methods

- `__init__(creg, index, value)`: sets the `start` attribute to `index` and the `end` one to `index+1`. Sets the other attributes according to parameters.

4.4.3 CReg

CReg stands for "Classical Register", which is an array of bits.

Attributes

- `identifier`: the classical register identifier.
- `size`: the number of the register bits.
- `symbol`: the symbol which is used in symbolic expressions to represent the register value. More about symbolic expressions and registers will be discussed in the 'Expression manipulator' section.
- `content`: an array containing instances of the `BitRange` and `Bit` classes, which symbolically represent the actual content of the register.

Static methods

- `fromStringLiteral(stringLiteral)`: returns a `CReg` having as content as many `Bit`s as the length of `stringLiteral`, with the former ones matching the `0`s and `1`s of the latter.

- `fromSymbolAndSize(identifier, symbol, size)`: returns a `CReg` having a certain `size`, represented by `symbol` and identified by `identifier`.

- `concat(lreg, rreg)`: takes two `CReg`s and returns a `CReg` having size `lreg.size + rreg.size`, and being the concatenation of `lreg` and `rreg`, in the order.

Instance methods

- `findBit(index)`: returns the first bit in the `content` array having `start == index`, *None* if not present.
- `getBit(index)`: returns the first bit in the `content` array having `start == index`, creating it if not present.
- `findRange(rangeDefinition)`: returns the range in the `content` array matching the `rangeDefinition`, *None* if not present.
- `getRange(rangeDefinition)`: returns the range in the `content` array matching the `rangeDefinition`, creating it if not present.
- `getList(expressions)`: returns the list of `Bit`s specified in the `expressions` array, creating them if not present.
- `setRange(rangeDefinition, value)`: sets the value of the range matching the `rangeDefinition`, creating it if not present.
- `setList(expressions, values)`: sets each `Bit` having `index` in `expressions` to the correspondent expression in `values` (which is a `BitRange`)

- `getSymbolicExpression()`: returns an expression that represents the content of the register. If there are no symbolic indexes, the actual integer value is returned (by converting the content from binary to integer).

4.4.4 QubitRange

This class encapsulates a range of qubits in a quantum register `qreg`. Like `BitRange`, it has `start` and `end` attributes, but neither value nor symbol.

Attributes

- `qreg`: the quantum register that has the current range as content.
- `start`: the range's starting index with respect to the parent `qreg`.
- `end`: the range's ending index with respect to the parent `creg`.

Methods

- `__init__(creg, start, end)`: sets the range attributes according to the parameters.
- `__hash__()`: returns the hash of `qreg.identifier + start + end` for the range, in order to create sets of `QubitRange`s.
- `__eq__(qubitRange)`: defines the equality for `QubitRange`s. Returns *True* if `qreg.identifier`s, `start` and `end` all match.

4.4.5 Qubit

Like the case for bit ranges and bits, a qubit is a particular case of qubit range, where the start and end expression coincide. So this class inherits all from `QubitRange` and its constructor makes sure that the range is one qubit long.

Methods

- `__init__(qreg, index)`: sets the `start` attribute to `index` and the `end` one to `index+1`. Sets the `qreg` parent attribute according to the parameter.

4.4.6 QReg

QReg stands for "Quantum Register", which is an array of qubits.

Attributes

- `identifier`: the quantum register identifier.
- `size`: the number of the register qubits.
- `content`: an array containing instances of the `QubitRange` and `Qubit` classes, which symbolically represent the actual content of the register.

Methods

- `addItem(item)`: appends `item` to the `content` array.
- `findQubit(index)`: returns the first qubit in the `content` array having `start == index`, *None* if not present.
- `getQubit(index)`: returns the first qubit in the `content` array having `start == index`, creating it if not present.
- `findRange(start, end)`: returns the range in the `content` array matching `start` and `end`, *None* if not present.
- `getRange(start, end)`: returns the range in the `content` array matching the `start` and `end`, creating it if not present.
- `getAll()`: returns the list of all `Qubit`s and `QubitRange`s in the register.
- `clone()`: returns a deep copy of the register

4.4.7 Range

This class is defined in the 'Symbolic execution engine' module due to dependency issues, but we discuss it here for clearness. It represents a Python-like range.

Attributes

- `node`: the parse tree node that the range is obtained from.
- `start`: the range start symbolic expression (e.g. in [1:4], the `start` would be `1`).
- `end`: the range end symbolic expression (e.g. in `[1:4]`, the `end` would be `4`). The `end` expression is not included in the resulting range.
- `step`: the range step expression (e.g. in `[0:2:8]` the step is `2`, meaning that the range will be `[0, 2, 4, 6]`)

Methods

- `__init__(node, context, size)`: instantiates a `Range` object by inspecting the `node` and evaluating its expressions in the provided `context`. In case one extreme is not specified, the `size` parameter is used to determine the actual range.
- `isSymbolic()`: returns whether the range is determined by actual numbers, or it has symbolic expressions as `start` or `end`.
- `toArray()`: returns the array that the range defines. If the range is symbolic, *None* is returned instead.

4.4.8 SetDeclaration

This class is defined in the 'Expression manipulator' module due to dependency issues, but we discuss it here for clearness. It represents an abstraction over 'setDeclaration' nodes of the parse tree.

Attributes

- `node`: the parse tree node that the set declaration is obtained from.
- `items`: an array containing the elements of the declared set.
- `start`: the start expression, in case the set is a range.
- `end`: the end expression, in case the set is a range.
- `step`: the step integer, in case the set is a range.

Methods

- `__init__(node)`: constructs a `SetDeclaration` object by inspecting the `node` and setting the attributes accordingly.

4.5 Expression manipulator

The aim of this module is to setup a tool which can evaluate expressions (coming from the parse tree) symbolically. This is done by taking an 'expression' node from the parse tree, building the respective Abstract Syntax Tree (AST) and translating it in a symbolic `sympy` expression.
An AST is composed of `UnaryOperator`s and `BinaryOperator`s as internal nodes, and `Variable`s and `Value`s as leaves.
Operators are simple data structures that hold a string literal for what operation they do represent, and one or two arguments (depending on the operator being unary or binary),

which are AST nodes in turn. Furthermore, they have the `applyTo` method, which takes symbolic operands as arguments and returns the symbolic expression of them combined with the operator.

4.5.1 UnaryOperator

This class constitutes a possible internal node for an expression AST.

Attributes

- `literal`: the operator string literal (e.g. ‘-’ for the negative sign, ‘!’ for the logical NOT, ‘ ’ for the bitwise NOT).
- `arg`: the AST node the operator must be applied to

Methods

- `__init__(literal, arg)`: returns an instance of `UnaryOperator` with the attributes set accordingly to parameters.
- `hasLeafArgument()`: returns *True* if the argument is a leaf (`Value` or `Variable`), or *False* if it is another operator in turn.
- `applyTo(operand)`: takes a symbolic `sympy` expression as argument and returns the symbolic expression with the current operator applied to the operand (e.g. `UnaryOperator(‘-’).applyTo(x+y)` returns `-(x+y)`, which is simplified to `-x-y` automatically by `sympy`).

4.5.2 BinaryOperator

This class constitutes another possible internal node for an expression AST.

Attributes

- `literal`: the operator string literal (e.g. ‘+’ for the addition, ‘-’ for the subtraction, ‘*’ for the multiplication, etc.).
- `arg1`: the AST node that the operator takes as left argument
- `arg2`: the AST node that the operator takes as right argument

Methods

- `__init__(literal, arg1, arg2)`: returns an instance of `BinaryOperator` with the attributes set accordingly to parameters.
- `hasLeafAsFirstArgument()`: returns *True* if `arg1` is a leaf, *False* if it is another operator in turn.
- `hasLeafAsSecondArgument()`: returns *True* if `arg2` is a leaf, *False* if it is another operator in turn.
- `applyTo(operand1, operand2)`: takes a two symbolic `sympy` expressions as arguments and returns the symbolic expression with the current operator applied to those operands (e.g. `BinaryOperator('*').applyTo(x+y, x-y)` returns `(x+y) * (x-y)`).

4.5.3 Expression

This is the core class of this module, and it is used for both arithmetical and boolean expressions.

Attributes

- `node`: the parse tree node the expression comes from
- `tree`: the root of the expression AST
- `isBoolean`: whether or not it is a boolean expression

Methods

- `__init__(node, tree, isBoolean)`: returns an `Expression` object having attributes set as specified by the parameters.
- `buildExpressionAST(node)`: builds the AST for the expression by inspecting the `node` and recursively instantiating `UnaryOperators`, `BinaryOperators` until `Values` and `Variables` are reached as base cases. The root of the AST is assigned to the attribute `tree`.
- `applyBinaryOperator(literal, secondOperand)`: modifies the expression by creating a new `BinaryOperator`, setting it as `tree` root. The operands are, in the order, the old expression and the `secondOperand`.
- `clone()`: returns a deep copy of the expression.
- `evaluate(context)`: recursively calls the `applyTo` methods of the operators in the AST in order to build the corresponding `sympy` symbolic expression. When a `Variable` base case is reached, the evaluator asks the `context` what the actual

value of that variable is; while when a `Value` is reached, no calls to the `context` are required since its actual value is hard-coded. The `context` argument is an instance of `Store`, which will be discussed in the 'Symbolic execution engine' section.

4.6 SMT solver

SMT stands for 'Satisfiability Modulo Theory', which means that an SMT solver is able to check formulas of a certain theory for satisfiability. In this case, the theory we are using is non-linear arithmetic, since our boolean expressions can include comparisons, equalities and inequalities between complex symbolic arithmetic expressions involving multiplications, powers, fractions and so on.
Building an SMT solver is a long and complex task, so we use the `PySMT` library. However, since our symbolic expressions are stored in `sympy` format, we need to convert them to allow `PySMT` to check for satisfiability.
This module simply converts a `sympy` expression to `PySMT` format and checks for satisfiability or unsatisfiability modulo non-linear arithmetic theory.

4.6.1 Solver

Static methods

All of the following methods take a `sympy` symbolic expression as argument.

- `getConvertedExpression(symbolicExpression)`: returns the `PySMT`-formatted expression by visiting the `symbolicExpression` `sympy` tree and progressively building the correspondent `PySMT` tree.
- `isSat(symbolicExpression)`: returns *True* if the `symbolicExpression` is satisfiable, *False* otherwise.
- `isUnsat(symbolicExpression)`: returns *True* if the `symbolicExpression` is unsatisfiable, *False* otherwise.

4.7 Symbolic execution engine

This is the core module of the project, it is responsible for the simulation of the QASM program, and during the symbolic execution it calls the SMT solver to determine whether certain execution paths are taken or not. Based on this, some regions of code that contain potential entanglements and measurements are detected to be unreachable, so they are not considered in the count of the effectively used qubits. In this section we are going to describe the APIs of the classes that this module is composed of, and we defer the actual explaination of the implementation to the last chapter.

The main concern of this module is to build a symbolic execution tree. Such a tree is composed of `SymbolicState`s.

4.7.1 SymbolicState

Each `SymbolicState` holds the following information:

- The **memory configuration** in a `Store` object
- The **constraints** that state assertions on the symbols defined that far.
- The set of all **potential entanglements** that are active and **quantum measurements** that have been detected that far (in the `QRegManager`).

Attributes

- `subroutine`: the `Subroutine` that the symbolic execution is performed on.
- `node`: the parse tree node of the instruction that generated the state.
- `store`: the `Store` object (discussed later).
- `constraints`: a `sympy` boolean expression made up by a conjunction of multiple assertions on the symbols defined that far. Those assertions are made by the execution engine whenever a certain condition occurs in the program. The details will be discussed in the next chapter.
- `qregManager`: the `QRegManager` object (discussed later).
- `children`: the children `SymbolicState`s. Only one in case of a sequential instruction, two in case of a branching statement or a loop.

Methods

- `addChild(childState)`: adds `childState` (which is a `SymbolicState`) to the array of `children`.
- `clone(newNode)`: returns a deep copy of the symbolic state.
- `addConstraint(booleanExpression)`: adds a constraint to the state by making a conjunction with the previous `constraint` expression.
- `symbolizeAll()`: substitutes all the variable values in the memory with fresh symbols. This is used when loops are simulated and there is no information on the post-condition of those variables.

4.7.2 Store

This class represents the memory configuration in a `SymbolicState`.

Attributes

- `store`: a dictionary that maps variable names to memory items. Each item is a dictionary that has two keys, 'value' and 'type', that represent respectively the symbolic expression (or `CReg` object, in case it is a register) associated to the variable, and the `ClassicalType` of it.

Methods

- `__init__(store)`: initializes the `store` with the parameter, or an empty dictionary if it is *None*.
- `clone()`: returns a deep copy of the store
- `set(key, value, type)`: sets the `value` and `type` of the item associated to the variable named `key` according to parameters.
- `get(key)`: returns the item associated to `key`.
- `getValue(key)`: returns the symbolic expression associated to the variable named `key`. *None* if the variable is not defined.
- `setValue(key, value)`: sets the `value` of the variable named `key`.
- `setType(key, type)`: sets the `type` of the variable named `key`.
- `getType(key)`: returns the `ClassicalType` of the variable named `key`. *None* if the variable is not defined.
- `evaluate(indexIdentifierNode)`: returns the symbolic expression associated to the `CReg` associated to the variable referenced by `indexIdentifierNode`, according to the indexes that appear in the latter.
- `assign(indexIdentifierNode, value)`: sets the `value` of the register referenced by `indexIdentifierNode`, according to the indexes that appear in the latter.
- `evaluateType(indexIdentifierNode)`: returns the `ClassicalType` associated to `indexIdentifierNode`, with the `designatorExpr1` set according to the size of the referenced portion of register.
- `symbolizeAll()`: substitutes all 'value' fields of each item with a fresh symbol.

4.7.3 QRegManager

This class implements a tool to keep track of potential entanglements and quantum measurements that occur in the program, so that the estimation of the number of qubits is reduced to a simple count of all the qubits that are entangled with a measured one, or are themselves measured.

Attributes

- `quantumRegisters`: dictionary that holds information about all the globally declared qubits.
- `potentialEntanglements`: an array of sets. Each one contains a certain number of `Qubit`s and `QubitRange`s that are potentially entangled.
- `effectiveQubits`: a set of `Qubit`s that have been measured, or were entangled to a measured one. The number of qubits in this set is the estimated upper bound to the number of effectively used qubits.

Static methods

- `fromParseTree(parseTree)`: returns a `QRegManager` with the `quantumRegisters` attribute already initialized with the ones that are declared in the program code.

Instance methods

- `__init__(quantumRegisters, potentialEntanglements, effectiveQubits)`: creates a new `QRegManager` object having the specified attributes.
- `addQReg(identifier, size)`: inserts a new `QReg` named `identifier` having the specified `size` in the `quantumRegisters` dictionary.
- `markAsMeasured(indexIdentifierNode, context)`: searches for the qubits specified in the `indexIdentifierNode` (evaluating its indexes in the specified `context`) and adds them to the `effectiveQubits` set.
- `addPotentialEntanglement(indexIdentifierNodes, context)`: searches for all the qubits referenced in the `indexIdentifierNodes` array (evaluating the indexes in the specified `context`), and merges all the already-existent potential entanglements containing one of them. If a qubit is not present in any of the potential entanglements, a new one is created and immediately merged with the others. At the end of the procedure, all the specified qubits will be in the same entanglement set and there will be no duplicates in the other potential entanglements.
- `clone()`: returns a deep copy of the `QRegManager`.
- `findEntanglementContainingQubit(qubit)`: returns the element appearing in the `potentialEntanglements` array that contains the specified `qubit`. `None` if there is no match.
- `checkQubitBound(bound)`: returns whether the effective number of qubits in the current state does not exceed the `bound` expression.

4.7.4 ExecutionStack

An instance of this class is a stack that holds the next program statements to be executed. A statement can cause one or more new statements to be pushed on top of the stack, since some of them hold program blocks that have to be expanded.

Attributes

- `sequence`: the array of statements that have to be executed. The last element is the first to be popped out.

Methods

- `pop()`: returns the next statement to be executed, removing it from the stack.
- `appens(block)`: pushes all the statements in the `block` on the stack.
- `isEmpty()`: returns *True* if the stack is empty, *False* otherwise.
- `clone()`: returns a deep copy of the execution stack.
- `addMetaInstruction(instruction)`: pushes a meta-instruction on the stack.

4.7.5 Meta-Instructions

Meta-instructions are instances of the `MetaInstruction` superclass, and they represent instructions for the symbolic execution engine. For example, at the end of a 'for' loop iteration, the engine must increment the value of the loop iterator, so the engine must know the exact moment when this operation needs to be done. This is achieved by pushing a meta-instruction on the stack underlying the loop body statements that reminds the engine to update the value of the iterator before going on with the other program statements.
The classes that represent meta-instructions are the following:

- `AddConstraintInstruction(constraint)`: adds the specified `constraint` to the `SymbolicState`.
- `SetStoreValueInstruction(identifier, value)`: updates the `Store` by updating the `value` of the variable having the specified `identifier`.

4.7.6 SymbolicExecutionEngine

This is a static class that has a single public method, but it wraps all the logic that allows the symbolic simulation to be performed, and the effective number of qubits to be estimated. As already anticipated, the details on how the engine operates are discussed in the next chapter. Here we describe the API of the class.

Static methods

- `analyzeSubroutine(subroutine)`: takes a `Subroutine` object as argument and sets its `symbolicExecutionTree` attribute. It also sets the `qubitChecks` array so that it is possible to determine if the upper bound on the number of qubits is respected.

Chapter 5

Symbolic Execution

This last chapter focuses on the actual symbolic execution of subroutines [**SurveySymExec-CSUR18**], how the various types of statement are simulated, and presents some concrete examples.
We are now going to discuss the implementation of the `SymbolicExecutionEngine`'s `analyzeSubroutine` method.

5.1 Initial state and execution stack

Suppose we have the following subroutine and we want to build its symbolic execution tree:

```
def foo(int:x, int:y) {
    int t = 0;
    if (x > y) {
        t = x++;
    } else {
        t = x + y;
    }
    t /= 2;
    return t;
}
```

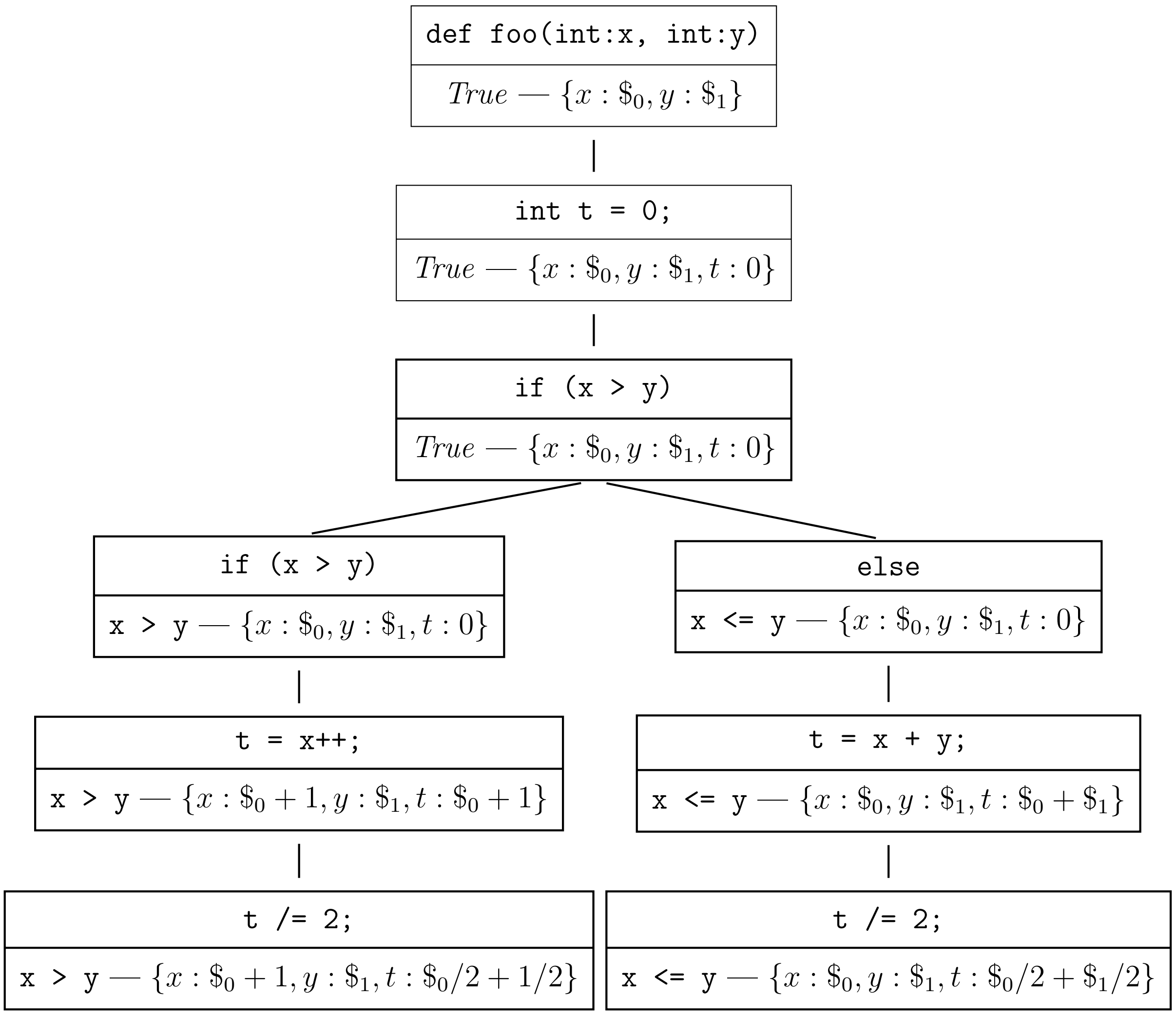
The `analyzeSubroutine` method takes the subroutine as an argument, analyzes its signature and builds the root `SymbolicState` (which we call `initialState`) with the following properties:

- `constraints`: *True*
- `store`: $\{x : \$_0, y : \$_1\}$

As we see, the two arguments of the subroutine have been assigned a fresh symbol each, and since there is no assumption on the state yet, we could set the constraints to *True*, meaning that the state is valid for each possible domain value of the symbols.
After defining the initial state, the `ExecutionStack` must be built, so we take the 'subroutineBlock' node, unravel its children (which are all statements) and push them on a brand new `executionStack`.

5.2 Simulation of statements

The next step is to call the `simulateExecution` private method, which takes a `SymbolicState` (representing the `currentState`), and an `ExecutionStack`. That is, this method performs the simulation of all the statements that are in the `executionStack` starting from the `currentState` parameter and generates a symbolic execution tree like the following one:


def foo(int:x, int:y)
True — $\{x : \$_0, y : \$_1\}$
int t = 0;
True — $\{x : \$_0, y : \$_1, t : 0\}$
if (x > y)
True — $\{x : \$_0, y : \$_1, t : 0\}$
if (x > y)
x > y — $\{x : \$_0, y : \$_1, t : 0\}$
else
x <= y — $\{x : \$_0, y : \$_1, t : 0\}$
t = x++;
x > y — $\{x : \$_0 + 1, y : \$_1, t : \$_0 + 1\}$
t = x + y;
x <= y — $\{x : \$_0, y : \$_1, t : \$_0 + \$_1\}$
t /= 2;
x > y — $\{x : \$_0 + 1, y : \$_1, t : \$_0/2 + 1/2\}$
t /= 2;
x <= y — $\{x : \$_0, y : \$_1, t : \$_0/2 + \$_1/2\}$

In order to generate such a tree, the `simulateExecution` method pops the statement on top of the stack, and it checks whether it is a concrete statement or a meta-instruction. In the first case we need to detect the statement type and simulate its behavior. One or more child states are created (based on the type of statement), and after setting them up (as well as the execution stack, if it needs to be modified) there is a recursive call for each of the child states. The base case occurs when the execution stack is empty, in which case the qubit upper bound is checked against the information that the `QRegManager` holds. Now we describe the operations of the symbolic execution engine for each statement type.

5.2.1 Expression statements

The simulation of expression statements, just like the majority of the others, starts with the cloning of the current state. The expression statement can only modify the state if incrementors are present (i.e. `++` or `--`). The expression node is passed to the `Expression` constructor, and an evaluation is performed on the store of the new state. Since the evaluation procedure automatically updates the values of variables in the store if incrementors are found, the simulation is over after the evaluation. The final value of the expression is discarded, and the engine can proceed with the execution of the next statement.

5.2.2 Assignment statements

There are two types of assignment statements: classical ones and quantum measurements.

Classical assignments

Classical assignments involve a LHS (Left-Hand Side) and a RHS (Right-Hand Side) expressions. The RHS can be either an expression or an index identifier. In the first case it is evaluated to a symbolic expression, while in the second case it becomes a `CReg`. The assignment operator can include a binary operator (e.g. `+=`, `-=`, etc.), in which case the LHS is also evaluated and the binary operator is applied to the couple (LHS, RHS). The final step is setting the value in the store, which is done by calling its `assign` method. If the old value in the store is a simple symbolic expression, it is just overwritten, but if it is a `CReg`, its `content` array is updated according to the new value.

Quantum measurements

Quantum measurements return a fresh symbol and set it in the store, since we ignore what happens at quantum level. Furthermore, when a quantum measurement is performed, the measured qubits and all the ones entangled to them are transfered by the `QRegManager` from the `potentialEntanglements` array to the `effectiveQubits` set by invoking the `markAsMeasured` method.

5.2.3 Classical declarations

The main difference between declarations and assignments is that the first have to take the type and domain of the variable into account. This is accomplished by calling the `ClassicalType` class constructor with the arguments obtained by inspecting the statement subtree.

Since declaration can have an immediate assignment, that case is handled like the corresponding statement, after having initialized the store item with the correct type.
Constant declarations are allowed too, but the type does not need to be specified. Thus, the store item will only contain the 'value' field, and this is fine because the type is only needed when treating symbols with the SMT solver.

5.2.4 Branching statements

When simulating if-then-else statements, both paths need to be generated, unless one of them has an unsatisfiable condition. In other words, the first thing to do is evaluate the boolean expression of the condition and check both it and its negation for unsatisfiability. If one of them is unsatisfiable, the corresponding path is not generated.
Once determined which paths are to be generated, a clone state and a clone execution stack is created for each one of them, and the corresponding program blocks are expanded into sequences of statements and pushed on top of their execution stack. This is necessary because a branching instruction hides all the statements that are in the nested blocks, and so they need to be explicitly pushed on the stack for the engine to execute them. Notice that the statements that appear after the end of the if-then-else remain untouched and are present in all the execution stack's clones, just below the expanded blocks.
All child states are appended to the current state's `children` array, and a recursive call to the `simulateExecution` method is done for each one of them, with the corresponding execution stack passed as second argument.

5.2.5 Quantum statements

When a quantum gate call is encountered, all the qubits that appear in the it are considered to be potentially entangled, so the `addPotentialEntanglement` method of `QRegManager` is called on them.

Example 5.2.1. Suppose we have the following quantum registers:

```
qreg q[4];
qreg r[4];
qreg s[4];
```

Here is an example of how the `QRegManager` arrays are affected by quantum statements (we put the array contents as comments between statements):

```
// potentialEntanglements: []
// effectiveQubits: []

CX q[0], q[1];

// potentialEntanglements: [ {q[0],q[1]} ]
// effectiveQubits: []

CX q[2], r[0];

// potentialEntanglements: [ {q[0],q[1]}, {q[2],r[0]} ]
// effectiveQubits: []

CX q[1], q[3];

// potentialEntanglements: [ {q[0],q[1],q[3]}, {q[2],r[0]} ]
// effectiveQubits: []

CX r[1:4], s[1:4];

// potentialEntanglements: [ {q[0],q[1],q[3]}, {q[2],r[0]},
//     {r[1],s[1]}, {r[2],s[2]}, {r[3],s[3]} ]
// effectiveQubits: []

CX r[2], r[3];

// potentialEntanglements: [ {q[0],q[1],q[3]}, {q[2],r[0]},
//     {r[1],s[1]}, {r[2],s[2],r[3],s[3]} ]
// effectiveQubits: []

CX r[0], s[3];

// potentialEntanglements: [ {q[0],q[1],q[3]},
//     {q[2],r[0],r[2],s[2],r[3],s[3]}, {r[1],s[1]} ]
// effectiveQubits: []

measure r[2];

// potentialEntanglements: [ {q[0],q[1],q[3]}, {r[1],s[1]} ]
// effectiveQubits: [ q[2],r[0],r[2],s[2],r[3],s[3] ]

measure q[0];

// potentialEntanglements: [ {r[1],s[1]} ]
// effectiveQubits: [ q[2],r[0],r[2],s[2],r[3],s[3],q[0],
//     q[1],q[3] ]
```

The code above uses only 9 qubits (the size of `effectiveQubits` array) out of 12 that have been declared. The QASM analyzer is therefore able to tell the programmer which qubits are effectively used, so that resources can be saved for other computations.

5.2.6 Loop statements

Simulating loops symbolically is a hard task, so in this version of the analyzer only 'for' loops with a known number of iterations have been considered. When such a loop statement is encountered, the execution stack is updated so that the loop block is repeated the correct number of times, and between two consecutive sequences there is a `SetStoreValueInstruction` that updates the value of the iterator.

Example 5.2.2. Suppose we have the following 'for' loop.

```
for i in [0:4] {
    CX q[i], r[i];
    c[i] = measure r[i];
}
```

Suppose also that the execution stack is already filled with some statements and the `for` loop on the top.

`for i in [0:4] {...}`
⋮

So, when the engine simulates the `for` statement, the execution stack becomes:

`SetStoreValueInstruction('i', 0)`
`CX q[i], r[i]`
`c[i] = measure r[i]`
`SetStoreValueInstruction('i', 1)`
`CX q[i], r[i]`
`c[i] = measure r[i]`
`SetStoreValueInstruction('i', 2)`
`CX q[i], r[i]`
`c[i] = measure r[i]`
`SetStoreValueInstruction('i', 3)`
`CX q[i], r[i]`
`c[i] = measure r[i]`
`SetStoreValueInstruction('i', 4)`
⋮

5.3 Limitations and future development

There are some features that have not been implemented in this simple version of QASM analyzer, such as:

- Function calls in expressions (e.g. `2 + a + getNumber()`)
- Boolean auto-cast in conditional statements (e.g. `if (variable)`): the libraries make it hard to cast from number to boolean.
- Bitwise shift operations (`<<` and `>>`): bit vectors are only available in `PySMT`, not in `sympy`.
- Modulo operation (`%`): not available in `PySMT`
- Alias statements (e.g. `let y = x`): keeping track of all the references could be difficult, especially if there are more levels of indirection.
- Indeterminate loops (e.g. `while (a > 0)`): here some technique should be used to detect loop invariants and use them to make assertions about pre- and post-conditions.
- Control directive statements (`break`, `continue` and `end`): they would complicate indeterminate loop handling even more.
- Symbolic designators in index identifiers (e.g. `a[n]`, where `n` is not known at compile time): every time a symbolic designator is detected, it should be checked against all the previous ones and check for equalities. This is quite time-consuming and it does not always guarantee that the analysis is correct.

In conclusion, we could say that this is the first version of a static analyzer for QASM, and the main concern is to estimate the number of qubits that are measured and/or entangled to measured ones. This could help quantum programmers to care about the resources that they use, and this is fundamental in quantum computing since qubits are really expensive, and the real power of quantum superpositions, that allow multiple states to be computed at the same time, is exponential with respect to the number of qubits.

5.4 Conclusion

The path for Quantum Supremacy is still long, but every step counts. Reducing the number of qubits that a program requires is a tiny step, but together with other innovations and newer technologies for the physical implementation of qubits, could make the difference and lead to a completely new scenario for communications, cryptography, big data and simulations of chemicals and phisical phenomena.