Queasy/KetPsi

Maharshi Kadeval
Rajiv Chitale
Rishit D
Vedant Bhandare

Language Specifications for DSL
CS3423: Compilers - 2
Department of Computer Science & Engineering
Indian Institute of Technology, Hyderabad

October 2023

Supervisor Dr. Jyothi Vedurada

Contents

Chapter 1

Basic Quantum Computing

1.1 Background

As computer science takes root in all aspects of our lives over the last few decades, computer scientist all over the world have begun to explore new paradigms. One of these promising paradigms includes Quantum-Computing; using the postulates of Quantum-Mechanics to possibly achieve machines that are more powerful than conventional Turing-Complete machines. Our DSL aims to provide a gateway to understand simple Quantum-Circuits and consequences of these circuits on a set of input quantum-bits, better known as qubits.

1.2 Qubits & Quantum-Gates

In the classical computing paradigm, we use classical bits that deterministically take 0 or 1 as its value. Qubits on the other hand, considers a *superposition* of 0 and 1. This can be represented as follows:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1.1}$$

$$= \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{1.2}$$

Classical-Gates like **AND**, **OR**, **NOT**, **XOR** perform operations on a certain set of classical bits. Similarly, Quantum-Gates perform certain transformations on the vector representing a qubit. As a result, all Quantum-Gates can be represented through

unitary matrices statisfying the following property:

$$UU^\dagger = U^\dagger U = \mathbb{1} \tag{1.3}$$

Unlike Classical-Gates, Quantum-Gates always have equal number of input-qubits and output-qubits, i.e., $fanin = fanout$.

1.3 Single Qubit Gates

Similar to a **NOT** gate in the classical computing paradigm, we have the **X** gate which serves as the quantum analgoue. We represent the **X** gate through the following 2×2 matrix as follows:

$$\mathbf{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.4}$$

When **X** acts on a state-vector $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, it switches the coordinates of the vector as follows.

$$\mathbf{X}\,|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} \tag{1.5}$$

Similarly, you have the **Y**, **Z** and other associated rotation gates along different axes. One other single qubit gate that is commonly used is the **Hadamard** gate, represented as follows:

$$\mathbf{H} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \tag{1.6}$$

1.4 Multiple Qubit Gates

Consider a **CNOT/CX** quantum-gate. This gate takes two qubits as input: control-qubit and target-qubit. This gate acts on the target qubit only when the control qubit is in state $|1\rangle$. Its matrix representation is as follows:

$$\mathbf{CX} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.7}$$

Similarly one can define any **CA** quantum-gate which performs the **A** quantum-gate

on the target-qubit based on the control-qubit. Similarly, we have doubly-controlled **NOT** gate also known as the **Toffoli** gate, represented as follows:

$$\mathbf{CCX} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.8}$$

1.5 Measurement

Upon measurement of any qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, we obtain a classical bit 0 with probability $|\alpha|^2$ and 1 with probabililty $|\beta|^2$. As always, we can measure a certain qubit out of a vector of qubits. For instance consider a 2-qubit state as follows:

$$|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle \tag{1.9}$$

The probability of getting 0 while measuring the first qubit would be $|\alpha|^2+|\beta|^2$; whereas the probability of getting 0 while measuring the second qubit would be $|\alpha|^2+|\gamma|^2$. After measurement the qubit is lost and only the normalized version of the remaining qubits remain. Say we measure the second qubit and get 1 as our measurement, our collapsed state is as follows:

$$|\psi\rangle \xrightarrow[\text{Obtain 1}]{\text{Measure } 2^{nd} \text{ Qubit}} \left(\frac{\beta\,|0\rangle + \delta\,|1\rangle}{\sqrt{|\beta|^2 + |\delta|^2}} \right) \tag{1.10}$$

Chapter 2

Aim & Program Flow

2.1 DSL Objectives

This program helps interested users define a Quantum-circuit, composed of:

- Quantum-Registers
- Classical-Registers
- Quantum-Gates
- Measurement Boxes

Users can construct circuits of their choice using predefined gates and also define blocks of gates for further usage. Users can also simulate measurements and accordingly store them in necessary Classical-Registers. Users can also use Classical-Registers to condition the use of gates and control flow of the circuit.

Users have the option to manipulate their data collected in the Clasical-Registers over multiple iterations over the circuit with specified input qubits. Users can compare these values with their assumed distributions and describe error, variance and other parameters. Users can also do basic arithmetic operations on these output-vectors.

2.2 Program

The basic program consists of three sections:

- `init`

- `main` denoted by `\begin` and `\end`
- `output`

Please find below a sample program that illustrates quantum teleportation.

```
\begin_init
    #registers quantum = 3
    #registers classical = 3
    #iters = 1000

    #set quantum states -> (1,0), (1,0), (1,0)
    #set classical states -> 0, 0, 0

    block BellGen(i, j) -> (j) [
        $ Start with Hadamard $
        H: i
        $ Apply C-X $
        X: i -> j
    ]
\end_init

\begin
    $ Generate Bell State $
    BellGen : (2, 3) -> (3)

    \barrier

    $ Init Transformations $
    X: 1 -> 2
    H: 1

    $ Measurements $
    measure: 1 -> 1
    measure: 2 -> 2

    $ Classical Controlled Ops $
    2 ? X : I
    1 ? Z : I

    $ Final Measurement $
    measure: 3 -> 3
\end

\begin_output
    $ Copy Output $
    int sat_alpha = 0;
    int sat_beta = 0;
```

```
43        for i in (0:7)
44        {
45            condition(i%2 == 0)
46            {
47                sat_alpha = sat_alpha + c_output[i];
48            }
49            otherwise
50            {
51                sat_beta = sat_beta + c_output[i];
52            }
53        }
54
55        $ Comparision $
56        list final = [sat_alpha, sat_beta];
57        list dist = [1000, 0];
58        list c = diff(final, dist);
59        echo(c);
60
61    \end_output
```

Listing 2.1: Quantum Teleportation

Chapter 3

Structure

3.1 `init` Section

This section contains three necessary statements that provide the compiler with necessary information about qbits

```
1       #registers quantum = 3
2       #registers classical = 2
3       #iters = 1000
```

to set number of quantum registers, set number of classical registers and set number of iterations, respectively.

Optional statements include initializing states of quantum registers:

```
1    #set quantum states -> (2,3), (0,1), (5,5)
2    #set classical states -> 0, 1
```

This sets the state of first quantum register as $2 + 3i$, second one as i and third one as $5 + 5i$ and sets the state of first classical register as 0 and second one as 1.

This section will also have user defined code block definitions (refer to block definitions section)

```
1    block (i,j,k) -> (p,q,r) {
2        $
3            statement calls
4        $
5    }
```

3.2 `main` Section

\begin marks the beginning of the main section and \end marks the end of the main section. This section includes

- calls to pre-defined or user-defined *blocks* (refer to block definitions section)
- \barrier: It's a directive to the compiler to not combine blocks before and after \barrier
- *measure* calls block that reads the value of a register and stores it in another register (both registers provided by the programmer)
- condition-otherwise for conditional statements
- *for*, *for_lex* and *for_zip* loops for repetitive calls to blocks.

3.3 `output` Section

This section contains the features of any general programming language to help interpret the results of the quantum experiment further and better understand the results from repeated runs of the circuit.
This section mostly contains C-like features but syntactic sugar for certain operations and loops. We also support more datatypes for vector and array manipulation.

Chapter 4

Lexemes

4.1 Comments

- $ `this is a valid comment` $
- $
  `this is a valid`
  `multiline comment`
  $

4.2 Whitespaces

We ignore all whitespaces in the `\output` section; although we consider newlines in the `\init` and `\main` sections.

4.3 Reserved Keywords

Some of the keywords used in the language are listed down below:

- `registers`
- `quantum`
- `classical`
- `iters`
- `set`
- `states`

- `block`
- `for`
- `for_lex`
- `for_zip`
- `condition`
- `otherwise`
- `begin`
- `end`
- `output`
- `barrier`
- `X`, `Y`, `Z`, `H` (gate names)

Apart from the above keywords, we also include reserved keywords for datatypes like `int`, `list`, `float`, `complex`, `matrix` etc.

4.4 Punctuations

Some of the punctuations in our language include $[,]$ $\{, \}, (\,) \,;, \rightarrow$.

4.5 Identifiers

We again standard conventions for identifiers; they can start with _ or a character from the alphabet, followed by an alphanumeric sequence.

4.6 Operands

All standard operands in most common & general programming languages are available. Apart from these operands, we also use $\rightarrow$ indicating control from one qubit to another. Similarly, we have ternaries to indicate a Classical-Register's control over a qubit. This is illustrated in the following section.

Chapter 5

Statements

5.0.1 Save Command

```
1    \save filename.csv
```

This command saves the output from main section of the code into an output csv file which can be used for further data manipulation

5.0.2 Echo Command

```
1    \echo
```

This prints the output from the main section of the code to the terminal

5.1 Output Statements

This section mainly involves mainpulating the data received from the main section. These are basic arithmetic operations and also allows graphical interpretation of the data received.
It allows basic programming constructs any other programming language can offer:

- Declaration statements
  - It follows C style syntax for variable declaration.
- Expression statements
  - This may involve using variables, carrying out arithmetic operations on them and assigning to other variables.
- Conditional statements

- It follows C-style conditional statements but with a different flavour:

```
condition(x > 3){
    ...
}
otherwise{
    ...
}
```

- Loop statements
  - Three kinds of for loops are provided:
    - for: it initializes an integer (i in this case). The integer loops through values from start to end (exclusive). The step in values can be provided optionally. These parameters are provided as a tuple. This is followed by the body of statements enclosed by curly braces.

```
for i in (start:end)
{
    X: i
}

for i in (start:end:step) {
    Y: i
}
```

    - for_lex: loops through all permutations of multiple integers in lexicographical order, in their respective ranges

```
for_lex (i,j,k) in (0:5,5:10,10:15) {
    X: i -> j
    Y: j -> k
}
```

    - for_zip: loops through multiple variables simultaneously.

```
for_zip (i,j) in (0:3,9:4:-2) {
    X: i -> j
}
```

      Here (i,j) take values (0,9), (1,7), (2,5)

Some additional features for data-manipulation are provided: **(more to be added on the go)**

- Difference or Sum of arrays

```
x = diff(A,B)
y = sum(A,B)
```

- Standard deviation of a list of numbers

```
a = std_dev(A)
```

- Variance of a list of numbers

```
a = var(A)
```

5.2 Operators

1. **X**:2 - Swap operation
   Here 2 is the target register and X is the operand
2. **X**: $2 -> 3$
   Here 2 is the control register, 3 is target register and X is the operand.
3. All other common operands in general programming languages are also included with similar operator precedence.

5.3 Binary If

This statement consists of a classical register and a quantum operator with a ? in between. The quantum operation only takes place if the classical register contains a 1.

```
2 ? X : 4->5
```

5.4 Measure

This reads from a quantum register (left) and stores the observation to a classical register (right). An arrow is used as punctuation. The qubit ceases to exist after this operation.

```
measure: 4->1
```

5.5 Block

A collection of statements can be abstracted and reused like an operation. A block is analogous to a user defined function.

- *Definition*:
  This consists of a call signature followed by a body of statements enclosed in square braces. The block-name consists of alphanumeric characters and underscore. It must start with an uppercase letter.

```
block MyBlock: (a,b,c) [
    $ statements $
]
```

  A user may also define a function with an arrow operator to make the control and target registers more explicit. Note that a block may contain statements other than quantum operations, such as for.

```
block MyBlock_2: (i,j,k) -> (p,q) [
    for t in (i..j)
    {
        Y: t->p
        k? X: q
    }
]
```

- *Call*: A block operator is fed arguments similar to operands in a quantum operation.

```
MyBlock_2: (2,9,0) -> (0,1)
```

Chapter 6

Possible Optimizations

6.1 Motivations

- Certain Quantum-Gates are self-invertible matrices or are commutable with other gates. Hence, rearranging such gates can yield easy optimizations.
- Moreover, certain combinations gates can reduce easily to other Quantum-Gates. This can lead to further optimizations through hard-coded sequences.
- A system of n-qubits would be represented through a 2^n state-vector and the corresponding matrix equivalent of the circuit through a $2^n \times 2^n$ matrix of complex entries. This represents a space explosion whose effect can be reduced by introduction of parallelization in matrix and tensor products.

6.2 Optimization Levels

The proposed optimization levels for the language are prescribed below:

- *Level 0:* No Optimization
- *Level 1:* Quantum-Gate Simplification & Elimination
- *Level 2:* Parallelization of Matrix and Tensor Products
- *Level 3:* Levels 1 + 2