Structure

October 24, 2023

1 Introduction

This document specifies the overall structure of a program in `KetPsi`. Later, we also describe the lexical conventions of the language (based on the Kernighan & Ritchie C Programming - Reference Manual).

2 Structure

2.1 `init`

This section is demarcated by `\begin_init` and `\end_init`. We now describe the possible statements & definitions that can come under this section.

1. **#registers_quantum = ⟨uint⟩**:
   This mandatory directive assigns the number of quantum registers in our program. The right hand side of this expression must be an unsigned integer.

2. **#registers_classical = ⟨uint⟩**:
   This mandatory directive assigns the number of classical registers in our program. The right hand side of this expression must be an unsigned integer.

3. **#iters = ⟨uint⟩**:
   This mandatory directive assigns the number of iterations of quantum observations in a program.

4. **#set quantum states →** (a_1, b_1)**,** (a_2, b_2)**, ...,** (a_n, b_n):
   This directive initializes the values in the quantum registers to the values on the right hand side. The values on the right hand side must be a complex-number or a 2-tuple of float-type numbers.

5. **#set classical states →** c_1**,** c_2**, ...,** c_n:
   This directive initializes the values in the classical registers to the values on the right hand side. The values on the right hand side must be a boolean (`True` & `False`) or a integer 0 or 1.

6. **Block Definitions**:
   This functionality allows us to reuse sections of the quantum-circuits by abstracting quantum registers to identifiers. We describe the block-calls later in the `main` section. Our block definitions occur as follows:

```
block <identifier>(q1, q2, ..., qn)->(o1, o2, ..., om)
[
  $ circuit statement calls $
]

```

Listing 1: Block Definitions

   Our definition always starts with `block` followed by the identifier given to the block. We now define the identifiers for the quantum registers that will be used in the block. This is optionally followed by an arrow and the specific quantum registers which are targeted. The main set of circuit statements would be placed within square brackets. These circuit statements are the same type of statements used in the `main` section (without barriers).

7. **Gate Definitions**:
   This functionality defines a single-qubit gate customized by the user through two ways - matrix representation or state mapping. This gate can be used in a controlled fashion at any later stage in the `main` section. Our gate definitions occur as follows:

```
gate <identifier1> = [[c1, c2], [c3, c4]]
gate <identifier2> = [[(a1, b1), (a2, b2)], [(a3, b3), (a4, b4)]]
gate <identifier3> = {(c1, c2), (c3, c4)}
gate <identifier4> = {((a1, b1), (a2, b2)), ((a3, b3), (a4, b4))}

```

Listing 2: Gate Definitions through Matrix Representation

   The first two definitions define the row-major ordering of the matrix-representation of the single-qubit gate - one through complex constants and the other through 2-tuple of float constants. The next two definitions define the mappings from $|0\rangle$ and $|1\rangle$ to their respective mappings - again through complex constants or float constants.

2.2 `main`

This section is demarcated by `\begin_main` and `\end_main`. We now describe the possible statements & definitions that can come under this section.

1. **Pre-defined Gate Calls**:
   We define 4 major single-qubit gates **X**, **Y**, **Z** and **H** representing Pauli-X, Pauli-Y, Pauli-Z and Hadamard gates. One can call these single-qubit gates in their simplest form or as multi-qubit controlled gates. These forms are illustrated below:

```
X -> 3
Y : 1 -> 2
Z : (1, 2, 3) -> 5

```

Listing 3: Predefined Gate Calls

   The first call applies the Pauli-X gate on the 3^{rd} quantum register. The second call applies the Pauli-Y gate on the 2^{nd} quantum register only when the 1^{st} quantum register are in the state $|1\rangle$. The third call applies the Pauli-Z gate on the 5^{th} quantum register only when 1^{st}, 2^{nd}, 3^{rd} quantum registers are in the state $|1\rangle$.

2. **User-defined Gate Calls**:
   We can again call user-defined gates via the same syntax above by replacing pre-defined gate names wiith user-defined gate names.

3. **Block Calls**:
   We call blocks by specifying quantum registers to be used in the block, with an optional check for output quantum registers.

```
BellGen : (3, 5)
BellGen : (3, 5) -> (5)

```

Listing 4: Block Calls

   The first call specifies that the call `BellGen` is called on the quantum registers 3 and 5. The second call does the same except it specifies the output quantum register for sake of clarity.

4. **Classical-Controlled Gate Calls**:
   We use ternary operators to control a particular action based on a classical register. This is achieved as follows:

```
3 ? X -> 5
!2 ? Y : 0 -> 1

```

Listing 5: Classical-Controlled Gate Calls

The first call only works when the classical-register 3 is set to 1. Similarly, the second call only works when the classical-register 2 is set to 0. (Should we have multiply-controlled classical registers??)

5. **Barrier**:
   We can specify a barrier to block optimizations joining the circuit before and after the location of the barrier. This is done by calling `\barrier`.

6. **Measure**:
   We specify measurement calls through `measure` calls specifying the classical register where we store the result of the measurement. An example call is illustrated below.

```
1       measure : 3 -> 0
2
```

Listing 6: Measurement

This call measures the 3^{rd} quantum register and stores the result in the 0^{th} classical register.

2.3 `output`

This section is demarcated by `\begin_output` and `\end_output`. We now describe the possible statements & definitions under this section.

2.3.1 Directives & Reserves

1. **c_output**: This stores the frequencies of occurences of all classical registers in a `list<int>` of unsigned integers. Consider a program with 1000 iterations and 2 classical registers. Then a possible `c_output` would be $[235, 276, 245, 244]$. This represents the following distribution:

C1	C2	Output
0	0	235
0	1	276
1	0	245
1	1	244

   We can access the values in `c_output` through standard indexing, for instance, `c_output[2]`.

2. **q_output**: This stores the final `state`s in a list of final quantum states in each quantum register. Note that we "zero" out quantum registers where we completed our measurement. This output comes up from only a single iteration. (Could we print all state-iterations out to a file??) We can access the `state` of the i^{th} quantum register via indexing, i.e., `q_output[i]`.

3. **Save**: This directive writes output to a given `csv` file. The syntax for the directive is as follows: `\save <filename>.csv`.

4. **Echo**: This directive writes output to `stdout`. The syntax for the directive is as follows: `\echo`.

2.3.2 Data Types

1. **int**
2. **float**
3. **string**
4. **complex**
5. **matrix**
6. **state**
7. **list⟨type⟩**

2.3.3 Control Statements

1. **condition(...){ ... }otherwise{ ... }**
2. **for i in (range) { ... }**: Range could be of the form `start:end` or `start:end:step`
3. **for_lex (i,j,k,...) in (range_i, range_j, range_k, ...)**
4. **for_zip (i,j,...) in (range_i,range_j)**

2.3.4 Data Mainpulation for `list`

The following operations only work for applicable types in `list`.

1. **Add/Sub**: Adds/Subtracts two lists component-wise.
2. **Dot**: Computes dot product of lists.
3. **StdDev**: Computes standard-deviation of a single list.
4. **Avg**: Computes average of a single list.
5. **Condense**: Condenses frequencies by ignoring certain bits. The syntax is `condense(A, (0,1))`. Assume A = [1, 1, 1, 1, 1, 1, 1, 1]. We assume positions written in binary as 000, 001, 010 and so on. Then condensing the first two bits adds up all even and positions to yield [4, 4].