QUARK

QUantum Analysis and Realization Kit

A High Level Programming Language for Quantum Computing

Team

In lexicographical order:

Name	UNI	Role
Daria Jung	djj2115	Verification and Validation
Jamis Johnson	jmj2180	System Architect
Jim Fan	lf2422	Language Guru
Parthiban Loganathan	pl2487	Manager

Introduction

In the early 1980's, Richard Feynman observed that certain quantum mechanical effects could not be efficiently simulated using classical computation methods. This led to the proposal for the idea of a "quantum computer", a computer that uses the effects of quantum mechanics, such as superposition and entanglement, to its advantage.

Classical computers require data to be encoded in binary bits, where each unit is always in a definite state of either 0 or 1. Quantum computation uses qubits, a special unit that can be 0 and 1 at the same time, i.e. a superposition of base states. Measuring a qubit will force it to collapse to either 0 or 1, with a probability distribution determined by its amplitude.

Qubits effectively operates on exponentially large number of entangled states simultaneously, though all of them will collapse as soon as we make a measurement. With carefully designed quantum algorithms, we are able to speed up certain classical problems dramatically by tapping into such massive computational resource. It is not unlike parallel computing, but powered by quantum mechanical laws.

Though quantum computing is still in its infancy, the last two decades have witnessed two ingenious algorithms that produced much inspiration and motivation for quantum computing research. One is Shor's algorithm (1994) for integer factorization, which yields exponential speedup over the best classical alternative, and the other is Grover's search algorithm (1996), which provides quadratic speedup for unsorted database search. Once realized, the former would have significant impact on cryptography, while the latter would have great implication on NP-hard problems.

Proposal

We would like to propose QUARK, a domain-specific imperative programming language to allow for expression of quantum algorithms. The purpose of QUARK is to define quantum computing algorithms and describe quantum circuits in a user-friendly way. In theory, our language can produce quantum circuit instructions that are able to run on actual quantum computers in the future.

Most quantum algorithms can be decomposed into a quantum circuit part and a classical pre/post-processing part. Recognizing this, QUARK is designed to integrate classical and quantum data types and controls in a seamless workflow. Built in types like complex numbers, fractions, matrices and quantum registers combined with a robust built-in gate library make QUARK a great starting point for quantum computing researchers and enthusiasts.

A relatively efficient quantum circuit simulator is included as part of the QUARK architecture. Source codes written in QUARK are compiled to C++, which can then be passed onto our quantum simulator.

Syntax

Comments

```
% single line comment
%{
  multi-line comments
}%
```

Variable Declarations Variables are declared in an imperative style with dynamic typing. There is no need to declare the type of the variable or demarcate a new variable with a keyword. The variable name is on the left and it is assigned a value using `=` operator to the result on the right side of the assignment. Also, every line ending is indicated by a `;` like in Java or C. We also suggest naming variables using underscores.

```
some_variable = "variable";
some_other_variable = 10;
```

Types QUARK supports the following types:

- Numbers
- Fractions
- Complex Numbers
- Booleans
- Strings
- Lists
- Matrices
- Quantum Registers

Numbers

All numbers are floats. There is no distinction between integers and floats.

```
num = 2;
pi = 3.14;
```

Fractions

Fractions can represent arbitrary precision and are represented using a `$` sign to separate the numerator and denominator.

```
% pi represents 22/7
pi = 22$7;
1/pi; % returns 7/22
pi + 1$7; % returns 23/7
```

Complex Numbers

Complex numbers can be represented using the notation `a+bi` where `a` and `b` are Numbers.

```
complex = 3+1i % This represents 3+i. We need b=1. It can't be omitted
complex * -.5i; % Arithmetic operations on complex numbers. This returns .5-1.5i
(-.5 + .3i) ** 5; % Use **n to raise to the power n
complex = 2.6 - 1.3i;

% The following assertions reurns true
norm(complex) == 2.6**2 + (-1.3)**2; % use norm() to get the norm
complex[0] == 2.6; % get real part
complex[1] == -1.3; % get imaginary part
abs(complex) == sqrt(norm(complex)); % use abs() to get the absolute value
```

Booleans

It’s simply `true` and `false`.

```
is_this_an_awesome_language = true;
```

Strings

Strings can be represented within double quotes. There are no characters. Characters are just strings of length 1. Escape a double quote with `\` as in `\"`. Get the string length with the `len()` function. Concatenate strings with `+=`. Access parts of string using `[]`.

```
some_string = "Hello World";
len(some_string); % returns 11
some_string += "!"; % It's now "Hello World!"
some_string[4]; % returns "o"
```

Lists

We make it easy to use lists, kind of like Python.

```
some_list = [1:5]; % returns list {1, 2, 3, 4, 5};
another_list = {"a", "b", "c", "d", "e", "f"}; % can be used to explicitly define elements i
another_list[-1]; % returns "f"
another_list[2:4]; % returns {"c", "d", "e"}
len(lis); % len() returns the size of the first dimension of the list
```

Matrices

Matrices are also easy to use. Similar to Matlab.

```
mat = [[2, 3],[5, 6],[-1, 2]];
mat'; % transpose a 2D matrix
mat2 = [[-2, 3],[0, 6]];
len(mat[0]) == 2; % returns true. This is the column dimension
len(mat) == 3; % returns true. This is the row dimension
mat[0][1]; % get element at position (0,1)
```

Quantum Registers

Quantum registers are the essential containers for qubit states and entanglement. Our language (and simulator) supports two modes of a quantum register: dense and sparse mode. Note that they are used for simulation only. Please use sparse mode if you know in advance that your quantum state vector will have relatively few non-zero entries.

Measuring a register, partially or totally, will force certain states to collapse probabilistically. A measurement is a non-reversible intrusive operation on a physical quantum computer, but our simulator supports an unrealistic mode of repeated measurement to facilitate simulation.

The measurement operator is `?`, while the unrealistic non-destructive measurement is `?'`

```
size = 5;
% quantum register  construction
qr1 = <size, 1>; % dense quantum register with initial state 1
qr2 = <size, 0>'; % sparse quantum register with initial state 0

% measurement
q ? 1; % measure a single qubit. Returns either 0 or 1
q ? 2:5; % measure qubit 2 to 4. Returns an integer from 0 to 31 that represents the resulta
q ? :5; % from qubit 0 to 4.
q ? 3:; % from qubit 3 to the last qubit.
q ?; % measure the entire register. The result will range from 0 to 2^size - 1

%{
same as above, except that you can repeatedly measure
without disrupting the quantum states.
Use this mode with caution because it is unrealistic.
}%
q ?' 1;
q ?' 5:;
q ?';
```

Control Flow if

`if` behaves similar to C. If the body of `if` has only one line, the brackets are optional. The condition is terminated by a colon.

```
l = [1, 2];
if len l == 0:
    return l;

if 1 == 1:
{
    x = 2;
    x += 1;
}
else
{
    x = 6;
    x ++;
}
```

for

Our `for` is similar to Python's for

```
for number in numbers:
    number += 1
```

Example of iterating over half of a list

```
for val in list[0:len(list)/2]:
    val = val*10;
```

You can also iterate over qubits in a quantum register

```
q = <10, 0>
for qbit in [: len(q)]:
    had q qbit;
```

while

`while` is python style and continually runs the block until the boolean expression becomes false

```
while val != 0:
    val -= 1;
```

Function definition

`def` keyword defines a new function. The parentheses at the line of `def` are optional.

Optional arguments are denoted by `arg_name = default_value`

```
def f1 a, b = 3
{
    c = a ** b + 1i;
    return c * (2 - 3i) + a;
}
```

Lambda

There are two styles of lambda.

The short version has a single statement as its body:

```
square = lambda x : x * x;
```

The long version has its body enclosed in brackets and can have more complicated control flow. A `return` statement is required if the lambda wants to return a non-void value.

```
square = lambda x : { return x * x; };
(lambda x : {if x < 2: return -1; else return 100;})(30) % return 100
```

Built-in Functions bit

Viewing the bit value of an integer is also critical to QC. `bit` takes an integer and bit position. The position is counted from the least significant bit.

```
bit(15, 0); % returns 1
```

len

`len` is short for length. `len` returns the length of a list, the number of characters in a string, the number of qubits in a quantum register, or the number of rows in a matrix.

```
list = [1,2,3,4,5];
len list; % returns 5
str = "hi earth";
len str == 8; % returns true
q_reg = <10,0>;
len(q_req); % returns 10
mat = [[1,2,3],[4,5,6]];
len mat; % returns 2
```

Sample Code

Classical algorithms GCD

```
def gcd(a, b)
{
    while a != 0:
    {
        c = a;
        a = b mod a;
        b = c;
    }
    return b;
}
```

Bitwise dot product

```
def bit_dot a, b
{
    limit = 1;

    c = a & b;

    counter = 0;
    i = 0;

    while limit <= c:
    {
        counter += bit c i++;
        limit <<= 1; % same as limit *= 2
    }

    return counter mod 2;
}
```

User-defined quantum gates Multi-qubit hadamard gate operation

```
def had_multi(q, lis)
{
    %{ if the list is empty,
      we apply had() to all bits
    }%

    if len lis == 0:
        lis = [: len q];

    for i = lis:
        had q, i;
}


def had_range(q, start, size = 1)
{
    had_multi q, [start: start+size] ;
}
```

Quantum Fourier Transform

`qft_sub` is the recursive subroutine used by `qft`

```
def qft_sub q, start, size
{
    if size == 1:
    {
        had(q, start);
        return;
    }

    % recurse
    qft_sub(q, start, size-1);

    last = start + size - 1;
    for t = [start : last]:
    % control gates are prefixed with 'c_'
        c_phase_shift q, PI / 2**(last - t), last, t;

    had q, last ;
}

def qft(q, start = 0, size = len q)
{
    qft_sub q start size;

    % reverse the qubits
    for tar = [start : start + size/2]:
        swap q, tar, tar+tarSize-1-tar;
}
```

Sample quantum algorithms Almost all quantum algorithms consist of a classical part and a quantum part. Classical part typically involves pre- or post-processing on a normal computer. Quantum part involves qubits and quantum circuits.

Deutsch-Josza Parity algorithm

An efficient $O(1)$ quantum algorithm to solve the Deutsch-Josza parity problem. The theoretical lower bound of a classical algorithm for this problem is $O(n)$.

```
import myutil; % user-defined libraries
import mygate;

% keyboard input
secret = int(
    input("Secret 'u' for Deustch_Josza parity algorithm"));

% quantum oracle
```

```
ocfun = lambda x : bit_dot(x, secret);

nbit = 5;

% dense mode, 6 qubits, initialize to state 1
q = <nbit+1, 1>;

% or sparse mode.
q = <nbit+1, 1>';

% apply hadamard gates
for i = [: len q] :
    had q, i;

% apply oracle
oracle q, ocfun, nbit;

% using a library function from mygate.qk
had_multi q, 0, nbit;

% measurement
result = q ? 0:nbit;
```

Grover's Search

This is one of the most celebrated quantum algorithms ever invented. Grover's search can efficiently find the needle in an unsorted haystack in $O(\sqrt{n})$ time, while the trivial lower bound for classical search algorithms is $O(n)$.

```
key = int(input("The key to search"));

% the search oracle
ocfun = lambda x : { return x == key; };

nbit = 5;
N = 2 ** nbit;
q = <nbit, 0>;

for i = [:nbit] :
    had q, i;

for iter = [: floor(sqrt N)]:
{
    % apply search oracle
    oracle q, ocfun, nbit;
    had_multi q, [:nbit];

    % define a diffuse matrix
    % initialize to all zeros
    diffuse = zeros N, N;
    diffuse[0][0] = 1;
    for i = [1: len diffuse]:
        diffuse[i][i] = -1;

    % apply this unitary gate
    generic_gate q, diffuse;

    had_multi q, [:nbit];
}

% measure the whole register,
% then shift one bit to the right.
result = (q ?) >> 1;
```