QClang: High level quantum computation

Connor W. Abbott [*] **Klint Qinami** [*]

[*] *Columbia University in the City of New York, New York, NY 10027 (e-mail: cwa2112@columbia.edu, kq2129@columbia.edu).*

Abstract: QClang will be a prototype higher-level language for quantum computing designed to compile down to QASM. It includes user-defined functions, higher-level control flow, and syntax similar to classical languages to make it easier to use for non-experts.

1. INTRODUCTION

While creating small programs in QASM is possible, large programs can become unwieldy. Additional control and data abstraction can allow for quick prototyping, larger and more complex programs, and greater ease-of-use. Towards this end, Q# is a recent high-level quantum computing language developed by Microsoft, supporting simple procedural programming.

This project aims to develop a new high-level programming language called QClang. QClang will be an imperative language supporting a small but expressive set of built-in types, with a syntax similar to other modern high-level languages.

2. BASICS

In addition to the builtin `qubit` type representing qubits, QClang has the following built-in types:

- 32-bit integers (`int`)
- Classical boolean bits (`bit`)
- Arrays of any of these types (`[]`)
- Tuples of any of these types (`()`)

To prevent cloning quantum bits, QClang enforces an affine typing constraint on the `qubit` type: each qubit must be used at most once. For example, `!` takes in a qubit (or classical bit) and returns its logical inverse. `!` is implemented by performing the QASM X gate on the qubit, and then returning the original qubit. Because the compiler ensures that the argument to `!` is never used anywhere else, no other part of the program can observe that it has been mutated. This allows us to unify the classical and quantum "dialects" of the language in a way that makes sense.

Another built-in feature that takes this unification farther is the `if` keyword. Used with a classical `bit`, `if`/`else` does more-or-less what you'd expect. But `if` statements can also have a qubit argument, in which case they perform the appropriate controlled operation. For example, to implement CX, one could do:

```
if (control) {
    target = !target;
}
```

Because controlled unitary gates preserve their control qubit, this use of `control` is not counted for in the affine typing rule. That is, we could use `control` somewhere else.

QClang also offers `for` loops, but since they are unrolled at compile time, their length must be bounded by a compile-time constant. Similarly, recursion depth must be bounded.

3. EXAMPLES

While the CX gate isn't one of the builtin gates in the language, we could implement it as follows:

```
(qubit, qubit) CX(qubit control,
                  qubit target) {
    if (control) {
        target = !target;
    }

    return control, target;
}
```

The quantum teleportation experiment can be transrcibed as follows:

```
(qubit, qubit) epr_pair()
{
    qubit a = hadamard(0), b = 0;
    if (a) {b = !b;}
    return a, b;
}

(bit, bit) meaure_bell(qubit a, qubit b)
{
    if (a) {b = !b;}
    a = hadamard(a);
    return measure(a), measure(b);
}

qubit teleport(qubit alice)
{
    qubit shared, bob;
    shared, bob = epr_pair();
    barrier();
    bit meas1, meas2;
    meas1, meas2 = measure_bell(alice,
                                shared);
    if (meas1) { bob = !bob; }
    if (meas2) { bob = Z(bob); }
    return bob;
}
```