QClang: High-Level Quantum Computing Language

*Klint Qinami** *Connor Abbott**
*Columbia University in the City of New York
kq2129@columbia.edu, cwa2112@columbia.edu

ABSTRACT
QClang [/klæŋ/] is a high-level, imperative programming language intended for the implementation and rapid prototyping of quantum algorithms. Its features include user-defined functions, unitary gates, quantum and classical data built-ins, and vectorized operators. The QClang compiler semantically enforces quantum restrictions, like the no-cloning theorem, through substructural typing. QClang code is compiled to the Open QASM intermediate representation and thus can be freely executed on various simulators and physical quantum machines, including the IBM Quantum Experience. Its syntax closely resembles that of the C programming language and other modern languages. These features, coupled with its higher-level control flow and abstraction, allow for a large reduction in the number of statements required to implement sophisticated quantum computing algorithms when compared to an equivalent implementation in Open QASM. The compiler, along with its regression test suite, is available on GitHub.

KEYWORDS: compilers, quantum computing, Open QASM, imperative languages.

1 Background

1.1 Motivation

Programming directly in a low-level language like Open QASM can be exceptionally difficult. While developing minimal programs is possible, generating hand-written Open QASM implementations of more sophisticated programs quickly becomes unwieldy. This challenge has already led to the development of SDKs like the Quantum Information Software Kit (QISKit), which allows for the generation of Open QASM IR through a higher-level language like Python. This additional abstraction naturally allows for quick prototyping, the generation of larger, more sophisticated programs, and greater ease-of-use. Additionally, outside of Open QASM, Q# is a recent high-level quantum computing language developed by Microsoft aiming at a similar goal by supporting simple procedural programming.

Simply put, QClang intends to be both a QISKit replacement and extension, allowing programmers to keep the benefits of the higher-level abstraction, while getting rid of the dependencies and baggage that arise from using an external,

purely classical, language. QClang also offers additional features, like semantic checking of quantum programs and arbitrary vectorized operators, which QISKit does not support. Ultimately, QClang aims to unify both its quantum and classical dialects in a way that is intuitive and easy to use.

1.2 Philosophy

In some sense, QClang is a language for old dogs who can't learn new tricks. It adopts a view of quantum computation as a classical program which manipulates various amplitudes. Classical data in QClang can be read, written, duplicated, and discarded as usual. It offers familiar and dearly held features of classical languages, like loops, ifs, and nots. However, along with these come powerful quantum primitives, like qubits, hadamard transformations, and general unitary gates. Quantum data supports unitary transformations and measurements as primitives, and cannot be copied and duplicated as classical data. It is not the place of the programmer to worry about such matters, however. Rather, it is the QClang compiler which enforces these constraints.

2 Language

2.1 Lexical Conventions

Tokens in QClang are similar to C. There are four kinds of tokens: identifiers, keywords, literals, and expression operators like (,), and *. QClang is a free-format language, so comments and whitespace are ignored except to separate tokens.

2.2 Types

QClang supports the types

- `int`: 32-bit two's complement signed integer.
- `bool`: boolean.
- `float`: single precision 32-bit floating point number.
- `qubit`: quantum bit.
- `bit`: classical bit.
- `tuple`: ordered set of values.
- `array`: fixed and variable size arrays.

2.3 Expressions

QClang supports the expressions

- Type Constructors: a type constructor is syntactically similar to a function call, except that instead of an identifier for the function to call there is a type.
- Function Calls: the function to call must be an identifier.
- Operators:

Precedence	Operator Class	Operators	Associativity
1	Parenthetical Grouping	`( )`	Left-to-right
2	Array Subscript Function Call	`[ ]` `( )`	Left-to-right
3	Unary Operators	`-, !`	Right-to-left
4	Multiplicative Operators	`*, /`	Left-to-right
5	Additive Operators	`+, -`	Left-to-right
6	Relational	`<, >, <=, >=`	Left-to-right
7	Equality	`==, !=`	Left-to-right
8	Assignment	`=`	Right-to-left

- Vectorized operators and Functions: unary operators, multiplicative operators, additive operators, hadamard, measure, U.
- Implicit Type Conversion: bools for bits.

2.4 Statements and Control Flow

QClang supports statements according to the following context-free grammar:

$$
\begin{aligned}
\text{stmt\_list} &\rightarrow \epsilon \mid \text{stmt\_list stmt} \\
\text{stmt} &\rightarrow \text{expr;} \\
&\rightarrow \text{typ ID;} \\
&\rightarrow \text{typ ID} = \text{expr;} \\
&\rightarrow \{ \text{ stmt\_list } \} \\
&\rightarrow \text{RETURN expr;} \\
&\rightarrow \text{IF (expr) stmt} \\
&\rightarrow \text{IF (expr) stmt ELSE stmt} \\
&\rightarrow \text{WHILE (expr) stmt} \\
&\rightarrow \text{FOR (expr; expr; expr) stmt}
\end{aligned}
$$

2.5 Built-in Functions

QClang supports the following built-in functions:

- `qubit hadamard(qubit q)`
- `qubit U(float x, float y, float z, qubit q)`
- `(qubit, qubit) CX(qubit control, qubit taget)`
- `bit measure(qubit q)`
- `barrier(variable number of qubits)`
- `length(array or tuple of any type)`

3 Testing

The compiler comes with a regression test suite. All tests can be run by executing the `testall.sh` script included in the topmost directory. The compiler can be made by executing `make` in the home directory. Any particular QClang source file can be compiled by running `./qclang.native sourcefile`. A python script `run_qasm.py` is included, which can execute the Open QASM output of the QClang compiler. It may be edited to change the number of shots, or the backend used. Its usage is `python3 run_qasm.py qasmsource`.

4 Examples

The QClang code and Open QASM circuits described in this section are included as an appendix.

4.1 Affine Typing Constraint

The algorithm 1 source is an example of a purposefully incorrect program in QClang, and shows an example of the kinds of errors the compiler will give if the affine typing constraint on qubits is violated. Qubit `b` is assigned to `a`, then assigned to `c` using an assignment of tuples. The compiler catches this and throws an error.

4.2 U-Gates

The algorithm 2 source shows the ability for a user to create functions and arbitrary unitary gates in QClang. Note that the first five lines of output of the compiler are a minimal header needed for any Open QASM program to run. This is done so that any compiled QClang program compiles to an executable Open QASM program.

4.3 Entangle-Unentangle

The algorithm 3 source displays tuple indexing and manipulation through an overly complicated no-op. Terminal session shows the pipeline of QClang → Open QASM → execution with 100 shots.

4.4 Quantum Teleportation

The algorithm 4 source shows the QClang implementation of the quantum teleportation experiment detailed in Bennet et al.

4.5 Deutsch-Josza Algorithm

The algorithm 5 source creates an array of `strlen` number of qubits in state `0`. The array is hadamarded at once through vectorization. A superposition qubit `answer` is created in state true and hadamarded in a single line. The oracle is a constant function returning zero. Querying it only requires flipping the `answer` qubit. Through vectorization, the test bit array is hadamarded and measured into a bit array in two lines.

4.6 Ripple-Carry Adder

The algorithm 6 source shows an implementation of a quantum Ripple-Carry adder in QClang.

5 Implementation

Our compiler was based on the `microc` compiler made for the PLT course here. Starting with this base allowed us to avoid reinventing the wheel, focusing on things actually relevant for a quantum language. After parsing and type-checking the source code, the compiler essentially acts as an interpreter, evaluating statements and spitting out Open QASM when it sees a new qubit, bit, or gate. While running the program, the interpreter knows the value of every boolean, integer, etc., but bits and qubits only contain a string with the QASM name of the emitted qubit/bit. Both qubits and bits have a special “invalid" value which is given to qubits/bits that aren’t initialized, or qubits that have already been read. This implements the affine typing constraint, as well as guaranteeing that qubits and bits aren’t used uninitialized.

6 Limitations and Future Work

One current limitation of the compiler is that Open QASM measurement outcomes and QClang measurement outcomes are not unified through an integrated namespace. This would require a consolidation with the backend used for execution. Presently, the backend reports on the measurement outcomes in its own order that may or may not correspond well with the QClang implementation.

As it currently stands, the Open QASM code generated from a QClang program is not optimal. There are many extraneous qubits and bits generated that can be optimized away by any decent optimizing compiler pass. This can be done by adding an internal Open QASM representation in the compiler, or through the development of a separate optimization program. Although we have not implemented such optimizations, adding them to the present compiler is a straightforward task.

Additionally, there are many instances where extra syntactic sugar would be helpful. For one, the ability to declare and assign in a single statement, increment and decrement operators, and so forth. We had only a few weeks to prioritize an essentially endless number of tasks, so these were niceties that were put on the cutting-floor due to a lack of time, and not due to any inherent difficulty in their implementation. A slightly more important feature that is yet to be added is an exact `pi` constant, similar to the one present in Open QASM. This `pi` constant would produce exact results when used with unitary gates, like `U`, and would need to be carried around internally by the compiler.

One more substantial feature that was cut was the ability to have `if`-statements whose conditions are qubits. That is, something like:

```
qubit a;
qubit b;
// ...
if (a) { b = foo(b); }
```

would do a controlled-`foo` gate. While certainly possible, the actual constructions for arbitrary nesting of control gates are somewhat involved, and unfortunately we didn't have the time to pursue it.

In the future, we'd need a standard library similar to `qelib1.inc` with more complex gates. As this is more "polish", it was left out.

7 Conclusion

QClang is a promising new higher-level language for quantum computation, integrating its quantum and classical dialects through simple language semantics. QClang represents quantum computation as an extension of classical computation, allowing users to utilize unitary transformations of qubits exactly when it is advantageous to do so. This 'superset' philosophy gives greater flexibility, and aims to attract both expert and novice quantum programmers to the emerging field. The QClang compiler uses the Open QASM intermediate representation, which is supported by many physical and simulated machines. Implementations of various known quantum algorithms have shown the utility of QClang, which produces easy-to-read and simple source code.

8 Acknowledgements

We'd like to thank Professor Lior Horesh and Professor John Smolin for an extremely enjoyable semester and for sharing their incredible knowledge and experience with us. Thank you for allowing us to pursue this project. We'd also like to thank the TAs for their assistance throughtout the semester. Thank you.

References

Bennett, C. H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A., and Wootters, W. K. (1993). Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899.

Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). Open Quantum Assembly Language. *ArXiv e-prints*.

Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 439(1907):553–558.

Appendix

Algorithm 1: QClang/tests/fail-qubit1.qc

```
[klint@xps13:QClang]
[klint@xps13:QClang] $ cat tests/fail-qubit1.qc
int main() {
  qubit a;
  qubit b;
  qubit c;
  a = b;
  (c, a) = (b, c);
  return 0;
}
[klint@xps13:QClang] $ ./qclang.native tests/fail-qubit2.qc
Fatal error: exception Failure("qubit b used more than once")
[klint@xps13:QClang] $
```

Algorithm 2: QClang/tests/test-u1.qc

```
[klint@xps13:QClang]
$ cat tests/test-u1.qc
qubit u1(float lambda, qubit q) {
  return U(0., 0., lambda, q);
}

qubit u2(float phi, float lambda, qubit q) {
  return U(3.14159265 / 2., phi, lambda, q);
}

int main() {
  qubit q1;
  qubit q2;

  q1 = false;
  q2 = false;
  u1(1.618, q1);
  u2(3.268, 6.28, q2);
  return 0;
}
[klint@xps13:QClang] $ ./qclang.native tests/test-u1.qc
OPENQASM 2.0;
include "qelib1.inc";
qreg q[1];
creg c[1];
h q;
qreg temp_0[1];
qreg temp_1[1];
U(0., 0., 1.618) temp_0;
U(1.570796325, 3.268, 6.28) temp_1;
```

Algorithm 3: QClang/tests/test-cx.qc

```
[klint@xps13:QClang] $ cat tests/test-cx.qc
int main() {
  qubit q1;
  qubit q2;

  /* entangle q1 and q2 */
  (q1, q2) = CX(hadamard(false), false);
  /* unentangle */
  (q1, q2) = CX(q1, q2);
  q1 = hadamard(q1);
  measure(q1);
  measure(q2);
  return 0;
}
[klint@xps13:QClang] $ cat tests/test-cx.out
OPENQASM 2.0;
include "qelib1.inc";
qreg q[1];
creg c[1];
h q;
qreg temp_0[1];
h temp_0;
qreg temp_1[1];
cx temp_0, temp_1;
cx temp_0, temp_1;
h temp_0;
creg temp_0_mb2[1];
measure temp_0 -> temp_0_mb2;
creg temp_1_mb3[1];
measure temp_1 -> temp_1_mb3;
[klint@xps13:QClang] $ python3 run_qasm.py tests/test-cx.out
COMPLETED
{'0 0 0': 100}
```

Algorithm 4: QClang/tests/test-teleportation.qc

```
/* FILENAME: QClang/tests/test-teleportation.qc */

tup(qubit, qubit) epr_pair() {
  return CX(hadamard(false), false);
}

tup(bit, bit) measure_bell(qubit a, qubit b) {
  (a, b) = CX(a, b);
  return (measure(hadamard(a)), measure(b));
}

qubit teleport(qubit alice) {
  qubit shared;
  qubit bob;
  bit meas1;
  bit meas2;
```

```
  (shared, bob) = epr_pair();
  (shared, bob) = barrier(shared, bob);
  (meas1, meas2) = measure_bell(alice, shared);

  /* TODO invert based on measurement */
  return bob;
}

int main() {
  qubit alice;
  qubit bob;

  alice = true;
  bob = teleport(alice);

  return 0;
}
```

Algorithm 5: QClang/tests/test-Deutsch-Josza.qc

```
/* FILENAME: QClang/tests/test-Deutsch-Josza.qc */

/* Create a bunch of either true or false qubits */
qubit[] const_qubit(int len, bool val) {
  int i;
  qubit[] out;

  out = new qubit[](len);
  for (i = 0; i < len; i = i + 1)
    out[i] = val;

  return out;
}

int main() {
  /* QClang implementation of Deutsch-Josza Algorith */
  int i;
  int strlen;

  qubit[] test_bits;
  bit[] measure_bits;
  qubit answer;

  /* Create Qubit array for oracle query */
  strlen = 10;
  measure_bits = new bit[](strlen);

  /* Create superposition state */
  test_bits = hadamard(const_qubit(strlen, false));

  answer = hadamard(true);

  /* Query oracle  */
```

```
  answer = !answer;

  /* Apply hadamard again */
  test_bits = hadamard(test_bits);

  /* Measure */
  measure_bits = measure(test_bits);

  return 0;
}
```

Algorithm 6: QClang/tests/test-adder.qc

```
/* FILENAME: QClang/tests/test-adder.qc */

/* gates transcribed from qelib1.inc */
qubit tdg(qubit a) {
  float pi;
  pi = 3.14159265359;
  return U(0., 0., -pi / 4., a);
}

qubit t(qubit a) {
  float pi;
  pi = 3.14159265359;
  return U(0., 0., pi / 4., a);
}

tup(qubit, qubit, qubit) CCX(qubit a, qubit b, qubit c) {
  (b, c) = CX(b, hadamard(c));
  c = tdg(c);
  (a, c) = CX(a, c);
  c = t(c);
  (b, c) = CX(b, c);
  c = tdg(c);
  (a, c) = CX(a, c);
  b = t(b);
  c = hadamard(t(c));
  (a, b) = CX(a, b);
  (a, b) = CX(t(a), tdg(b));
  return (a, b, c);
}

/* adapted from QISKit/examples/generic/adder.qasm */

tup(qubit, qubit, qubit) majority(qubit a, qubit b, qubit c)
{
  (c, b) = CX(c, b);
  (c, a) = CX(c, a);
  return CCX(a, b, c);
}

tup(qubit, qubit, qubit) unmaj(qubit a, qubit b, qubit c) {
```

```
  (a, b, c) = CCX(a, b, c);
  (c, a) = CX(c, a);
  (a, b) = CX(a, b);
  return (a, b, c);
}

tup(qubit[], qubit) ripple_add(qubit[] a, qubit[] b) {
  qubit carry;
  qubit cout;
  qubit[] out;
  int i;

  out = new qubit[](length(a));
  carry = false;
  for (i = 0; i < length(a); i = i + 1) {
    (b[i], a[i], carry) = majority(carry, b[i], a[i]);
  }

  (carry, cout) = CX(carry, false);

  for (i = length(a) - 1; i >= 0; i = i - 1) {
    (carry, out[i], a[i]) = unmaj(b[i], a[i], carry);
  }

  return (out, cout);
}

int main() {
  qubit[] a;
  qubit[] b;
  qubit cout;

  a = new qubit[](4);
  b = new qubit[](4);

  a[3] = false;
  a[2] = false;
  a[1] = false;
  a[0] = true; /* a = 0001 */
  b[3] = true;
  b[2] = true;
  b[1] = true;
  b[0] = true; /* b = 1111 */
  (b, cout) = ripple_add(a, b);
  meas_array(b);
  measure(cout);
  return 0;
}
```