QuantC: A C-Like Language for Quantum Computing

Tyler Burkett

June 30, 2022

1 Introduction

In this project, I contribute two major deliverables. First, I define a new programming language QuantC, which aims to provide a single uniform interface to write code that executes on both a classical and quantum computer. Second, I implement a compiler for QuantC in C++ with the aid of several standard compiler-writing tools. I used Flex and Bison to implement a lexical analyzer and parser. I also used the LLVM library to generate the classical portions of code, with the `gcc` compiler serving as both a preprocessor and linker, in order for the compiler to create executable programs that can run on a standard command line on a Linux computer. To test the quantum components of the code generated, I ran a quantum virtual machine (QVM) provided as a part of the ForestSDK from Riggetti Computing as a local server on the same machine.

Quantum computing appears to be the next big paradigm in computation, given the growing body of quantum algorithms and their ability to provide significant computational speed-up compared to classical algorithms [7]. Programmers will need programming languages and toolkits that help them write semantically correct code for new quantum computing systems. By establishing clear distinctions between classical and quantum code alongside additional compile-time checks, QuantC aims to be a user-friendly language for developing programs on future quantum-enabled systems.

2 Background on Quantum Computing

This section does not intend to fully discuss quantum mechanics and quantum information theory but rather to provide an overview of several terms and concepts discussed throughout this paper. For exploring these topics further, a source like [8] is better suited.

Qubits are the basis of data storage for a quantum computer, analogous to a bit for a classical computer. Qubits have the property that they can take on more state values than the 0 or 1 values of a bit. Instead, a qubit's state can take on various values, which a vector within a Hilbert space better describes. A **Hilbert space** is a vector space equipped with an inner product and norm. The Hilbert space for a qubit is a 2-dimensional complex vector space, and any state of the qubit is represented as a linear combination, also called a superposition, of two basis states $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Due to physical constraints on qubits, we add the additional constraint that the weight of $|0\rangle$ in any qubit state is real and non-negative. Additionally, qubit state vectors are normalized. Specifically, the state of a qubit $|\psi\rangle$ is then described as $|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\phi}\sin(\theta/2)\,|1\rangle$, where $0 \leq \theta \leq \pi$ and $0 \leq \phi < 2\pi$ [5]. These constraints lead to a representation of a qubit's state space by what is called a Bloch sphere. Figure 1 illustrates this space. On the other hand, the only state value a bit can take on corresponds to $|0\rangle$ and $|1\rangle$. The Bloch sphere helps illustrate how qubits differ from bits' capacity to hold information.

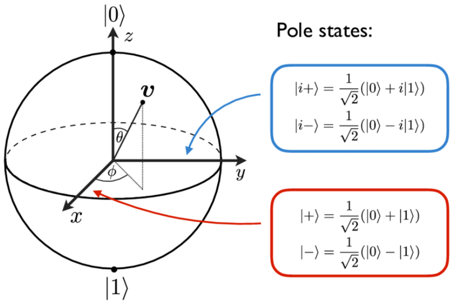

$|0\rangle$
z
Pole states:
$|i+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$
$|i-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$
$\boldsymbol{v}$
θ
ϕ
y
x
$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$
$|1\rangle$

Figure 1: A visual representation of the state space of a qubit [5]

The notation used to describe the base states of a qubit is Bra-Ket notation, also known as Dirac notation. A ket ($|\rangle$) refers to a 2^n-element complex column vector and is used to represent the state of n qubits, whereas a bra ($\langle|$) refers to the adjoint (conjugate transpose) of a ket. Consequently, a bra refers to a linear functional and is used to describe operations like measurement. In combination, the notation allows us to know what operation is implicitly performed between them; $|\rangle\,|\rangle$ is a tensor product, $\langle|\,|\rangle$ is an outer product, and $|\rangle\,\langle|$ is an inner product.

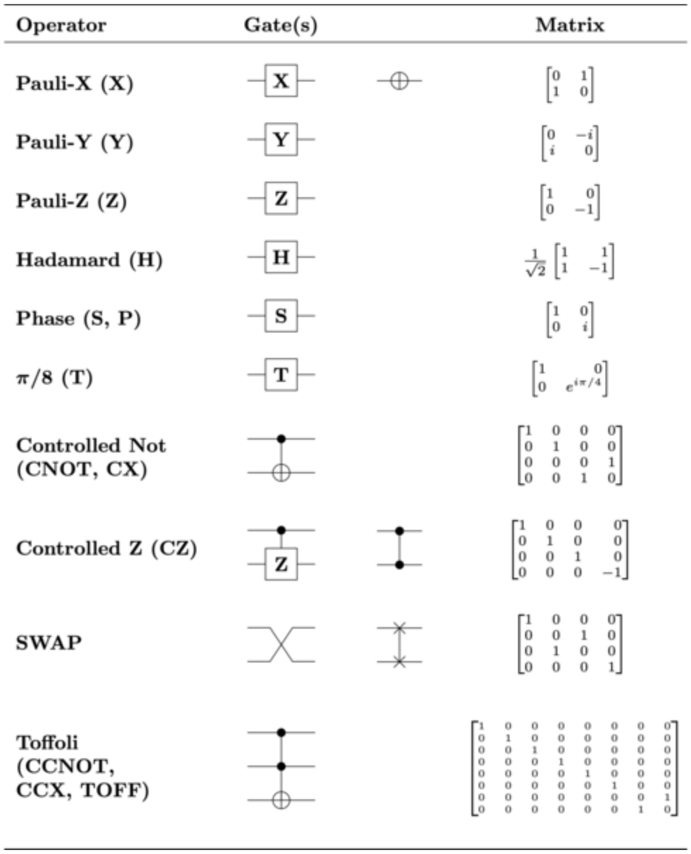
Although the state vector formalism is sufficient to describe qubit states, it becomes less effective when talking about qubits whose states are not entirely known. For example, consider applying one of two operations onto a qubit q_2 based on the measured result of another qubit q_1. If the measurement result from q_1 was unknown for some reason, the exact state of the qubit q_2 is not known. Instead, q_2 would statistically be in one of two states dependent on the measured value of q_1. We refer to the possible states that q_2 can be in as **pure states**. We refer to q_2 being in this statistical ensemble of pure states as being in a **mixed state**. In this case, we could use the density-matrix formalism to describe the state of q_2 more easily. Formally, consider the set of pure states $\{\ |\,\psi_i\rangle\,\}$ that q_2 could be in with probability p_i. The density matrix ρ that represents the mixed state of q_2 is defined as $\rho \equiv \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right|$. A mixed state represented by a density matrix does not necessarily correspond to a qubit's physical state. In the previous example, the density matrix for q_2 corresponds to its mixed state, but q_2 is only ever physically one of the possible pure states. Still, the resultant density matrix directly encodes information about the probability of the qubit collapsing to a specific base state when measured, even though the qubit's pure state is unknown. Both the density-matrix and state-vector formalism are mathematically equivalent [8], so which one to use is often a matter of preference.

Two categories of operations affect qubits. First, square unitary matrices represent operations manipulating the state of a qubit. These unitary operations are reversible. Because of reversibility, one can think of these operations as quantum gates, similar to logical gates in classical circuits with the same number of inputs as outputs. Second are non-unitary measurement operations. Measurement operations cause a given qubit to collapse into a basis state with a certain probability. For a specific example, consider the state of a qubit $\left|\psi\right\rangle = \alpha \left|0\right\rangle + \beta \left|1\right\rangle$. For a measurement that causes this qubit to collapse into either $\left|0\right\rangle$ or $\left|1\right\rangle$, the probability that this qubit is in one of those states after the measurement is $\|\alpha\|^2$ and $\|\beta\|^2$, respectively. Measurements are non-reversible and usually appear at the end of some chain of unitary operators to obtain a binary value from qubits. These measurement operations describe special, non-reversible gates applied to one or more qubits.

An interesting consequence of the nature of qubits leads to the concept of **entanglement**. Entanglement refers to the situation where two or more qubits become associated in such a way that each state cannot be described independently of one another. For example, consider that the state of a 2-qubit system could be $\frac{1}{\sqrt{2}} \left|00\right\rangle + \frac{1}{\sqrt{2}} \left|11\right\rangle$, where $\left|\alpha\beta\right\rangle$ is shorthand for $\left|\alpha\right\rangle \left|\beta\right\rangle$. In this 2-qubit system, there is no way to factor this state description into a product of two independent qubit states. Furthermore, measuring one qubit in this described configuration forces the other qubit to have the same value. In other words, although only one qubit is measured, entanglement causes the other qubit's state to be modified as well, in this case measured. This entanglement property of qubits opens up possibilities for developing new algorithms, such as communication or encryption. However, entanglement can also lead to problems if, for example, it is between **ancilla** qubits, additional qubits introduced to convert an otherwise irreversible computation to a reversible one, and result qubits in a computation. Unwanted entanglement sometimes forces us to **uncompute** ancilla qubits before measuring, applying a set of unitary operations on ancilla and result qubits that are the reverse of a previous step of computation that entangled them.

Understanding unitary operators and measurements to be gates allows for a description of quantum computations as circuits. Figure 2 shows some common unitary operations in both their matrix and symbol gate representations. These gates, taken together, can be used to perform various operations, even classical operations like addition.

However, some constraints make computation with qubits more restrictive than with bits. Specifically, these constraints are known as the no-cloning and no-deleting theorem. These theorems state that there exists no unitary operation U such that $U(|\phi\rangle\,|0\rangle) \rightarrow |\phi\rangle\,|\phi\rangle$ (cloning) or $U(|\phi\rangle\,|\phi\rangle) \rightarrow |\phi\rangle\,|0\rangle$ (deleting) for an arbitrary quantum state $|\phi\rangle$. These constraints mean that storing quantum states is not nearly as easy as classical bits, leading to the circuit model being an appropriate choice for modeling quantum computation.



Operator	Gate(s)	Matrix
Pauli-X (X)	$\boxed{\mathbf{X}}$ $\oplus$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)	$\boxed{\mathbf{Y}}$	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)	$\boxed{\mathbf{Z}}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)	$\boxed{\mathbf{H}}$	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)	$\boxed{\mathbf{S}}$	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)	$\boxed{\mathbf{T}}$	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	$\boxed{\mathbf{Z}}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Figure 2: Common logic gates [9]

The circuit model of quantum computation is just the starting point of quantum computation. The abstract quantum computation model typically involves a classical computer extended with quantum instructions and access to registers of qubits, referred to as a QRAM [6]. One way to extend the classical computer is by communicating with a coprocessor: a quantum computer that implements only the quantum instructions and qubit registers. The classical computer executes the classical instructions as expected and sends quantum instructions to the quantum computer. A quantum computer's instructions correspond to the unitary and measurement operations mentioned above, along with **state preparation** operations that set qubits into at least one base state, usually $|0\rangle$. These quantum instructions thus also describe quantum circuits. After finishing execution, the quantum computer returns any resulting classical values to the classical computer. When performing operations that involve classical memory, different desgins for quantum computers vary. We can expect the classical computer to handle all classical operations and store any classical values needed by the quantum computer itself, transmitting them when needed. However, it is also possible that the quantum computer has the capacity to handle some classical operations and memory on its own [10] to reduce the cost of multiple accesses to the quantum computer.

3 Programming Language Features and Semantics of QuantC

My design of QuantC aims to achieve several goals. The first goal is to make a clear separation between classical and quantum code. A programmer should be able to identify what sections of code are meant to run on a quantum computer versus a classical computer. The distinction will hopefully reduce programming errors by making these sections of code easy to distinguish.

The second goal is to minimize the overhead of learning a new language. The primary method of achieving this goal is making QuantC a dialect of a preexisting language rather than a new language entirely. Not only does the similarity reduce the chances of programming errors, but it also eases the burden of modifying preexisting code from the base language to QuantC.

The final goal of QuantC is to abstract communication between the classical and quantum computer while still providing a clear mapping between code and machine instructions. Given the receny and current scale limitations of quantum computers [4], code written in QuantC needs to map closely with lower-level instruction sets as much as possible. How quantum instructions are communicated is less important than ensuring that the instructions perform as expected. Abstracting communication between the classical and quantum computers eases the burden on the programmer during development without affecting code correctness.

3.1 A Superset of C Inspired by CUDA

The design of CUDA, a C dialect implemented by Nvidia to write programs that can utilize a GPU, inspires the design of QuantC. In CUDA, a function marked with a `__global__` keyword encapsulates code meant to execute on the GPU. Both the code inside the `__global__` function and the invocation of that function inside other regular C/C++ functions mostly resemble their regular counterparts. CUDA also includes additional syntax to accommodate differences in code execution on a GPU versus a CPU, namely SIMD code execution. QuantC adopts these broader aspects of CUDA in its language design. Any valid C program should also be a valid QuantC program. Changes to the syntax and grammar of C are only extensions. The semantics of classical portions of code remains unchanged, only affecting areas of code specific to QuantC.

3.2 Assumptions about the Quantum/Classical Architecture

QuantC makes several assumptions about the relationship between quantum and classical computers. The quantum computer operates as a co-processor to the classical computer. The classical computer sends the quantum computer a set of instructions to run. The quantum computer, in turn, executes the instructions and returns some classical value(s) to the classical computer. The quantum computer and the classical computer do not share memory. The quantum computer contains some classical memory and can perform some classical instructions on that memory. QuantC does not expect the quantum computer to execute classical instructions with the same performance as the classical computer. The communication between the two types of computers is synchronous. This last assumption is primarily due to the lack of syntax for asynchronous operations in C. Given that current quantum computing systems usually necessitate remote access, future work may relax this assumption, as detailed in Section 6.1.

3.3 The `quantum` Keyword

In the QuantC grammar, the **`quantum`** keyword is a **storage class specifier**, associated with type information to define how a function or variable is stored in memory and linked during the linking step. A **quantum function** is a function denoted with the **`quantum`** keyword, which represents a set of quantum operations to be executed on a quantum computer. Likewise, a **quantum variable** is a variable denoted with the **`quantum`** keyword, which represents a collection of qubits in a quantum computer. Quantum variables can only appear inside quantum functions. For reference later in the document, a **classical function** and **classical variable** refer to the same constructs without the **`quantum`** keyword.

3.4 Semantics of Quantum Functions

A quantum function can take both classical and quantum variables as arguments. The **`quantum`** keyword in arguments denotes them as quantum values, identical to how quantum variables are declared. However, quantum arguments can only be passed by reference, denoted with `&`, similar to how pass-by-reference is denoted in C++. Only quantum arguments can use the pass-by-reference syntax. Classical arguments for quantum functions operate identically to regular C function arguments.

```
quantum void basic_function(int value1, quantum int &value2) {
    ...
}
```

Figure 3: An example of a permissible quantum function signature in QuantC

Any quantum function can contain a call to any other quantum function, excluding recursive calls. Forward declarations of quantum functions are disallowed in the translation unit they appear in, so circular calls are by necessity also disallowed. However, classical functions can only contain calls to quantum functions with exclusively classical arguments. None of these classical arguments may be pointers. A quantum function can also contain classical control structures like `if/else` and `while` statements. A quantum function may only return a classical value like a classical function, regardless of which kind of function the call appears in.

All return types, arguments, and variables defined for a quantum function can only be of a built-in type. These built-in types include `bool`. This restriction is primarily due to how limited classical resources may be for a quantum computer, which may prevent structured types from being stored inside the quantum computer. In addition, quantum computers may have addressing restrictions that make structured types impossible. Due to the no-cloning theorem, a quantum computer must ensure that non-unary quantum operations do not reference the same qubit multiple times. Again, it may be impossible to enforce this rule at compile time in the presence of structured types. Local quantum variables are implicitly reset at the end of a quantum function to prevent entanglement with qubits outside the function, because these qubits are reused.

Quantum functions compile to a quantum intermediate representation (IR) suitable to send to a quantum computer. The QuantC compiler stores this quantum IR as a part of the final executable program in a format that the quantum computer can receive and process into the final machine instructions before execution. QuantC leaves the specific quantum IR format(s) undefined. For example, a compiler could store the quantum IR in either a machine-specific binary or text-based machine-independent format.

At calls to quantum functions inside classical code, rather than generating a regular C function call, a QuantC compiler inserts a call to a library function to handle communication between the classical and quantum computers. This function takes at minimum the generated IR, the name of the quantum function called at the corresponding location in code, and any values passed as arguments to the quantum function. Any additional parameters or implementation details of this built-in function are left undefined.

3.5 Semantics of Quantum Variables

Quantum variables represent a collection of qubits inside the quantum computer. Quantum variables cannot be assigned a value after declaration. Quantum variables can accept classical integer values as initializers. QuantC defines the order of significance and size of integer values and literals to match those properties of the classical C types, usually MSB first and 32 bits long. The i_{th} classical bit of the value being either 0 or 1 initializes the i_{th} qubit of the variable to either $|0\rangle$ or $|1\rangle$, respectively. An uninitialized quantum variable is equivalent to the variable being zero-initialized. These semantics ensure that every quantum variable always refers to the same set of qubits throughout the lifetime of a quantum function. Quantum variables also accept non-classical values as initializers, though the semantics are necessarily different from the prior case.

Section 3.6 covers this case of initializers in detail.

Array variables can also be declared as quantum, but their declared size and any values used to index them must be constant at compile time. Pointer variables cannot be declared quantum. This restriction is due to addressing limitations by quantum computers discussed in Section 3.4. Quantum arrays accept initializer lists of classical values like C arrays. An entry in the quantum array follows the same initialization semantics as the corresponding entry in the initializer list. Any entries not defined in the initializer list are zero-initialized.

3.6 Quantum Operations and Expressions

Quantum operations correspond to applying a quantum gate to a set of qubits. Some quantum operations also take an additional classical value. These operations correspond to **parametric gates**, generic gates whose effects are partially defined by a classical value representing an angle (in radians). Consequently, these classical values must be floating-point numbers. Parametric gates are often generalized versions of other non-parametric gates. For parametric gates, the terms “unary” and “binary” are used only concerning their quantum arguments; quantum gates do not operate on gate parameters.

```
// Unary Gates
%I myQubits; // Identity (No change)
%X myQubits; // X-Gate (Bitwise NOT extended to qubits)
%Y myQubits; // Y-gate
%Z myQubits; // Z-Gate
%H myQubits; // Hadarmard
%PHASE(angle) myQubits; // Phase
%S myQubits; // S-Gate
%T myQubits; // T-Gate
%RX(angle) myQubits; // Rotate-X Gate
%RY(angle) myQubits; // Rotate-Y Gate
%RZ(angle) myQubits; // Rotate-Z Gate
%M myQubits; // Measurement

// Binary Gates
myQubits ^ myOtherQubits // Controlled NOT (X) gate
myQubits <> myOtherQubits // Swap Gate
myQubits <I> myOtherQubits // Imaginary Swap Gate
myQubits <P>(angle) myOtherQubits // Parameterized Swap Gate
```

Figure 4: The set of quantum operations defined in QuantC

A **quantum expression** is a combination of quantum variable identifiers and quantum operations. Quantum expressions can only appear in quantum functions. Since the value of qubits cannot be copied, the resulting value of a quantum expression is the set of (possibly value-modified) qubits involved in the expression. Quantum operations correspond to applying a quantum gate to the qubits operated on. The one exception to this rule is the measurement operation. The **measurement** operation takes a quantum expression and results in a classical integer value corresponding to the collapsed states of the qubits inside the quantum expression. Quantum operations apply a side effect (the corresponding quantum gate) to quantum variables contained in the expression. However, instead of evaluating to the value of the quantum variables (which cannot be copied or measured without modifying them), quantum operations return "references" to those quantum variables. The compiler handles theses "references" during code generation.

When the initializer of a quantum variable declaration is a quantum expression, the semantics of the declaration differs from that of a classically-valued initializer. In this case, the initialization follows a move semantics; the new quantum variable defined in the declaration takes ownership of the qubits associated with the quantum variable used in the initializer. Consequently, the quantum variable in the quantum expression cannot appear more than once inside the expression, and further statements within the current scope cannot reference that variable. Classical values in the initializer, such as parameters to certain quantum operations, remain unaffected by declarations; later statements can reference these values. The type of the newly declared quantum variable must match the type of the variable used in the expression. These semantic rules prevent the case of multiple identifiers referring to the same set of qubits and later used together in the same quantum expression.

```
quantum bool a = 0;
quantum bool b = 0;

quantum bool c = %H a; // Allowed; "a" is moved to "c".
quantum bool d = %H a; // Error; "a" in no longer available.
```

Figure 5: An example of declarations with simple quantum expressions.

QuantC provides a new declaration syntax for expressions containing multiple quantum variables to move each quantum variable to a new name. QuantC borrows this syntax from the C++ structured binding assignment. Figure 6 depicts this syntax. The number of identifiers in the declaration must match the number of unique quantum variable identifiers in the initializer expression. QuantC bases the mapping of an old variable to a new one on their order in the initializer; the leftmost variable becomes the first declaration, the next leftmost variable becomes the second declaration, and so on. The types of each newly declared quantum variable correspond to the types of the quantum variables they map to in the initializer. This declaration syntax otherwise follows the same semantics as declarations using quantum expressions.

```
quantum bool a = 0;
quantum bool b = 0;

// a <> b => a = 1, b = 0
// qubits of "a" and "b" are moved to "c" and "d", respectively
// c renames a = 1, d renames b = 0
auto [c, d] = a <> b;
```

Figure 6: An example of the new declaration syntax. Note the move semantics of the last declaration; the *values of* `a` and `b` have swapped, but the *qubits associated with* `a` and `b` go to `c` and `d`.

For initializers containing quantum arrays, the constant index counts as part of a unique identifier in a quantum expression. However, the newly declared quantum variable refers to the entire array after the declaration. This choice of semantics with arrays prevents a compiler from needing to track invalid indices in quantum arrays. Otherwise, the semantics of initializing with quantum arrays remains identical to non-array quantum variables.

```
quantum bool a[2];
quantum bool b[2] = a[1] <> a[2]; // Allowed;
quantum bool c[2] = a[1] <> a[2]; // Error; "a" is no longer available
```

Figure 7: An example of array declarations with quantum expressions.

3.7 Operation Modifiers

QuantC also defines some **operation modifiers** that change how quantum operations affect quantum variables. When prepended to a quantum operation, the inverse modifier `>` causes the operation applied to be the inverse of the given operation. The inverse of a given unitary matrix U always exists; the inverse of U is the conjugate transpose of U. When appended to a quantum operation, the control modifier `+C(cQubits)` causes the quantum variable *cQubits* to control the operation. For the classical analog to a controlled gate, given a reversible classical gate G operating on bits b with control bits c, the controlled version

CG on applies G to b if all bits of c are 1.

For a given unitary matrix U, we can construct the corresponding controlled unitary matrix CU controlled by a single qubit as a composition of the following matrices

$$CU = \begin{bmatrix} [I] & [0] \\ [0]^{\dagger} & [U] \end{bmatrix}$$

where $[I]$ is the 2×2 identity matrix and $[0]$ is a $2 \times dim(U)$ matrix of zeros.

```
%H myQubits; // Regular Hadamard gate
~>%H myQubits; // Inverse of the Hadamard gate
%H +C(myOtherQubits) myQubits; // Hadamard gate controlled by other qubit(s)
```

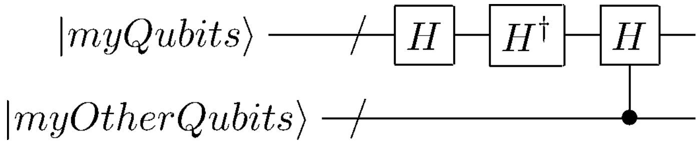


Figure 8: The operation modifiers available in QuantC and their corresponding circuit diagram.

4 Sample Programs and Outputs

```
int printf ( const char * format, ... );

quantum bool hello_quantum() {
    quantum bool my_quant = 0;
    return %M (%H my_quant);
}

int main() {
    return printf("%d\n", hello_quantum());
}
```

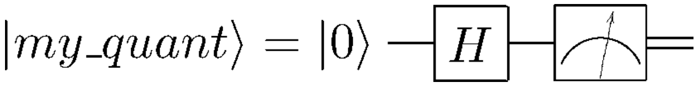


Figure 9: The “Hello, Quantum!” program, which uses a single qubit as a random number generator. The output of the program should be either “0” or “1” with equal probability.

```
int printf ( const char * format, ... );

int quantum bell_state() {
    quantum bool q1 = 0, q2 = 0;
    return %M (%H q1 ^ q2);
}

int main() {
    return printf("%d\n", bell_state());
}
```

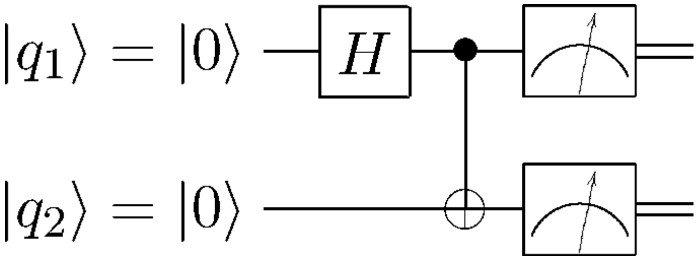


Figure 10: The “Bell state” [8] program, which demonstrates a simple form of entanglement. The output of the program is either “0” or “3” with equal probability. These values correspond to the values 0x00 and 0x11 respectively; the qubit values can never be measured to be 0x01 or 0x10.

```
// Qubits may only be passed by reference.
void quantum toffoli_001(quantum bool(&x)[3], quantum bool &y) {
       // Negate qubits 0 and 1
       // Precedence: [] before %X
       %X x[0]; %X x[1];

       // "+C" is an quantum operation modifier on %X (NOT gate);
       // it connects the qubits of x as control qubits to NOT
       %X +C(x) y;
       %X x[0]; %X x[1]; // Uncompute; same precedence as first line
} // toffoli_001

// The other Toffoli gate functions follow a similar pattern, except
// for how the qubits of x control the NOT gate
// (i.e. are negated before control)
void quantum toffoli_101(quantum bool(&x)[3], quantum bool &y) {
       %X x[1];
       %X +C(x) y;
       %X x[1]; // Uncompute
} // toffoli_101

void quantum toffoli_010(quantum bool(&x)[3], quantum bool &y) {
       %X x[0]; %X x[2];
       %X +C(x) y;
       %X x[0]; %X x[2]; // Uncompute
} // toffoli_010

void quantum toffoli_100(quantum bool(&x)[3], quantum bool &y) {
       %X x[1]; %X x[2];
       %X +C(x) y;
       %X x[1]; %X x[2]; // Uncompute
} // toffoli_100
```

Figure 11: The "Simon's Problem" program (part 1). Simon's Problem [7] is as follows: given a classical function $f : \{0,1\}^n \to \{0,1\}^n$ with the property $f(x) = f(y) \iff (x = y \vee y = x \oplus s)$ for a fixed unknown $s \in \{0,1\}^n$, determine s with as few queries as possible. This section depicts some of the quantum functions used to build the function f for the quantum computer.

```
// Example Oracle: s = 110 (binary)
void quantum black_box(quantum bool(&x)[3], quantum bool(&y)[3]) {
       toffoli_001(x, y[0]);
       toffoli_101(x, y[0]);
       toffoli_010(x, y[1]);
       toffoli_100(x, y[1]);
       toffoli_001(x, y[2]);

       %X +C(x) y[2];
} // black_box

int quantum simons_algorithm_step() {
       quantum bool x[3] = {0,0,0};
       quantum bool y[3] = {0,0,0};
       int result;

       %H x;
       black_box(x, y);
       %M y;
       %H x;

       // Determine result by measuring x
       result = %M x;
       return result;
} // simons_algorithm_step
```

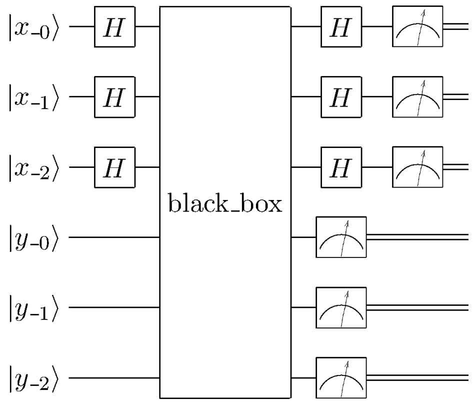
Figure 12: The “Simon’s Problem” program (part 2). This section depicts the quantum function for f (`black_box`)and the steps to query it.

```c
// From this point on, program is classical code and should
// thus follow standard C precedence
int classic_black_box(int x) {
        switch(x) {
                case 0: case 6:
                        return 0;
                case 1: case 7:
                        return 1;
                case 2: case 4:
                        return 2;
                case 3: case 5:
                        return 4;
                default: return -1; // error
        }
} // classic_black_box

// Attempt to solve the system of equations as a mtrix using
// Gaussian Elimination and back substitution
void solveMatrix_mod2(int a[2][3]) {
    int m = 2, n = 3;

    // Gaussian Elimination mod 2
    int row, column, rows_after;
    for(row=0; row<m-1; row++){
        for(rows_after=row+1; rows_after<m; rows_after++) {
            int term = a[rows_after][row];
            for(column=0; column<n; column++) {
                int value = (a[rows_after][column] - term * a[row][column]) % 2;
                a[rows_after][column] = (value == -1) ? 1 : value;
            } // column
        } // rows_after
    } // row

    // Back substitution mod 2
    for(row=m-1; row>=0; row--) {
        for(rows_after=row-1; rows_after>=0; rows_after--) {
            int term = a[rows_after][row];
            for(column=n-1; column>0; column--) {
                int value = (a[rows_after][column] - term * a[row][column]) % 2;
                a[rows_after][column] = (value == -1) ? 1 : value;
            } // column
        } // rows_after
    } // rows

} // solveMatrix_mod2()

#define THIRD_BIT_MASK 0x4
#define SECOND_BIT_MASK 0x2
```

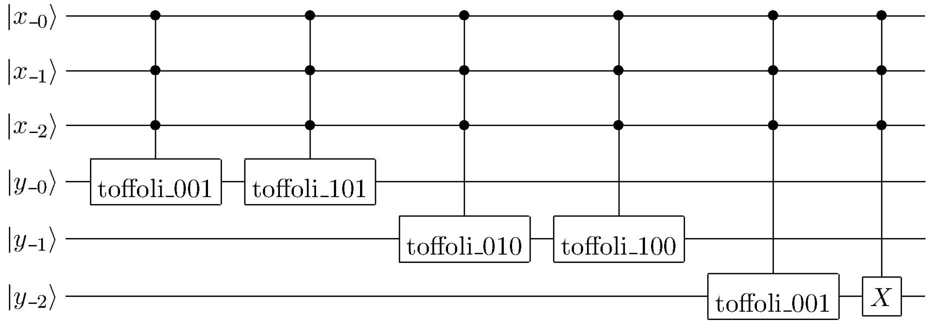
Figure 12: The “Simon’s Problem” program (part 3). This section depicts the function f as classical code and a function for solving systems of equations. The classical section of code calls `simons_algorithm_step` multiple times to form a system of equations to solve for s. For this particular f, $s = 110_2 = 6$.

```c
#define FIRST_BIT_MASK 0x1

int main() {
        // Since we are using a 3 bit in/out oracle, perform the quantum step twice
        int s = 0;
        int i;

        int step1_result = simons_algorithm_step();
        int step2_result = simons_algorithm_step();

        // Solve system of equations to determine all but the last bit of s
        int matrix[2][3];
        if (step2_result > step1_result) { // Swap rows
            int temp = step1_result;
            step1_result = step2_result;
            step2_result = temp;
        }
        matrix[0][0] = (step1_result & THIRD_BIT_MASK) >> 2;
        matrix[0][1] = (step1_result & SECOND_BIT_MASK) >> 1;
        matrix[0][2] = step1_result & FIRST_BIT_MASK;
        matrix[1][0] = (step2_result & THIRD_BIT_MASK) >> 2;
        matrix[1][1] = (step2_result & SECOND_BIT_MASK) >> 1;
        matrix[1][2] = step2_result & FIRST_BIT_MASK;
        // Current compiler causes SEGFAULT when this functin is ran
        solveMatrix_mod2(matrix);

        // For this case, easiest guess is top row of solved matrix
        // Take all but the last bit
        for (i = 0; i < 1; i++) {
                s = (s ^ matrix[0][i]) << 1;
        }
        s <<= 1;

        // Check result with classical version of oracle to decide guess for last bit
        if (classic_black_box(s) != classic_black_box(0)) {
                s += 1;
        }

        // Print guess (probability of guessing right > 1/2)
        printf("guess for ’s’ = %d\n", s);
        return 0;
}
```

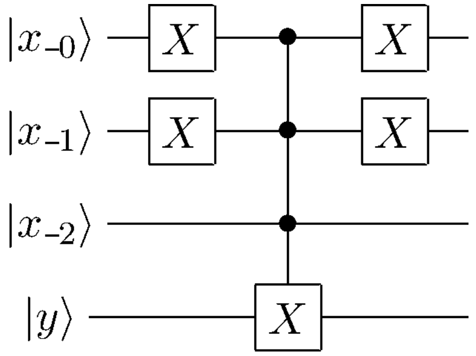
Figure 12: The “Simon’s Problem” program (part 4). This section depicts the classical code to query the quantum version of f and attempt to guess s. We note that f only needs to be queried $\mathcal{O}(n)$ number of times in order to form a guess for s. The output should be `guess for ’s’ = 6` more than half of the time.



(a) The circuit diagram for the `simons_problem_step` function.



(b) The circuit diagram for the `block_box` function.



(c) The circuit diagram for the `toffoli_001` function. The other similarly named functions have a similar circuit diagram, with each qubit of `x` having different combinations of `X` gates applied before and after the central gate.

Figure 12: The "Simon's Problem" program (part 5). This section depicts the circuit diagrams for the quantum components of the program. The last step of converting the values of measured value of `x` into an integer is omitted.

5 Implementation of a Compiler for QuantC

I wrote all code for this project in C++. I compiled the code with `gcc` using the C++17 standard. For the most part, I designed the code to avoid requiring platform-specific features, relying on libraries and compiler tools to abstract the process away from the machine as much as possible.

When run on the command line, the compiler takes the path to the QuantC file to compile. The compiler assumes that the programmer includes a `main` function in the program. At each step, the compiler generates intermediary files in the `/tmp/Quantc/` directory. These intermediary files include the preprocessed QuantC program, the generated IR, the object files generated from the IR, and the final executable `a.out`. Currently, the compiler does not have any other command-line arguments to modify the behavior of any of these steps.

5.1 Preprocessor

Since QuantC does not modify the preprocessing directives, the compiler calls `gcc` to take the initial QuantC program and preprocess it. Along with its use as a linker (Section 5.7), `gcc` enables QuantC programs to link with standard C library code that classical sections of code can use.

```
std::string preprocess_command = "gcc -E -P ";
std::string preprocessed_file = cd + "/" + file_name + ".i";

preprocess_command += std::string(argv[1]) + " -o " + preprocessed_file;

if (system(preprocess_command.c_str()) == EXIT_FAILURE) {
    std::cout << "Error preprocessing file";
    return -1;
}
```

Figure 13: Compiler code illustrating how the compiler calls `gcc`

5.2 Lexical Analyzer

To create the scanner, I defined the token regular expressions and actions in a Flex scanner description file, then generated the scanner tables with Flex. The token regular expression and actions are a modified version of the Lex specification for C provided by Jutta Degener [2]. The scanner description file also contains additional modifications to have Flex generate the scanner as a C++ class. Due to how the parser generated by Bison (detailed in the next section) expects to call the scanner for the next token, I added a shim for the scanner to operate with the parser. This shim consists of a scanner class that inherits from the one generated by Flex, with a token-generation method having a signature that the parser expects.

5.3 Parser

To create the parser, I defined the grammar rules and actions in a Bison grammar file, then generated the parser tables with Bison. The grammar rules and actions are a modified version of the Yacc specification for C provided by Jutta Degener [3]. The grammar file also contains additional modifications to have Bison generate the parser as a C++ class. I also utilized the Bison-defined "variants" class with the parser to allow the scanner to pass near-arbitrary semantic values, such as the values of literals, back to the parser in addition to the type of token the scanner generates. Although it is useful, this "variants" class results in the need for the shim mentioned in the previous section. Specifically, the parser expects the scanner to have a method with the following signature.

```
int Scanner::lex(Parser::semantic_type *yylval);
```

The scanner `lex` method does not typically take an argument. By providing the scanner shim for the parser, the scanner can return additional semantic values for tokens via ‘yylval‘. The scanner peforms this action by calling the ‘emplace‘ method on the variant as follows.

```
// The semantic value be of any type or class defined in parser specs
auto semanticValue = SemanticValueClass(...);
yylval->emplace<SemanticValueClass>(semanticValue);
```

The variant functions like a C union, allocating a shared memory location to store values of multiple types in a single object. The parser specification defines the semantic value of each token, which in turn specifies all the expected data types the variant can store. The following example illustrates how the parser defines these semantic values and how they can be used to build an abstract syntax tree.

```
// Both terminal (tokens) and non-terminal symbols can have
// semantic values associated with them
%token <TokenTypeA> TOKEN_A
%token <TokenTypeB> TOKEN_B
%token <TokenTypeC> TOKEN_C
%nterm <TreeNodeA> ntermA
%nterm <TreeNodeB> ntermB
...
%%
...
// Grammar rule
ntermA : TOKEN_A TOKEN_B
{
    // Grammar action
    // $1, $2 are the semantic values of the tokens
    // produced by the scanner.
    // $$ is the value of the non-terminal symbol
    $$ = TreeNodeA($1, $2);
}
;

ntermB : ntermA TOKEN_C
{
    // Other grammar rules access the semantic value of a non-terminal
    // identically to tokens' semantic value.
    $$ = TreeNodeB($1, $2);
}
;
```

Figure 14: An extract of an example Bison grammar file illustrating how grammar rules can build AST nodes using the variant semantic values.

5.4 Abstract Syntax Tree (AST)

I implement the abstract syntax tree (AST) as a set of C++ classes. Each class represents a node of the AST. These nodes take advantage of inheritance to create categories of nodes that mirror QuantC's grammar. There are six main categories of nodes that inherit from a generic AST node class (`ASTNode`): translation units (`TranslationUnitNode`), declarations (`DeclNode`), function arguments (`ArgDeclNode`), types (`TypeNode`), statements (`StmtNode`), and expressions (`ExprNode`). All other nodes that represent more concrete parts of grammar, such as for loops and binary expressions, directly or indirectly inherit from one of these node categories.

The constructors of these nodes take smart pointers to other nodes to form the AST. By taking smart pointers to generic category nodes, AST nodes take advantage of method overriding when invoking a sub-node's methods for later compilation steps. Smart pointers also trivialize memory management for arbitrary trees, making the parser's task of building the AST easier than if I used regular pointers. Specifically, I designed the AST with C++ `shared_ptr` smart pointers to allow for multiple pointers to a single object. Since some structures in the grammar have optional components, I defined node classes representing the lack of a specific class of node, such as `NullExprNode`. Each category node instantiates the corresponding null node as a singleton, justifying the need for `shared_ptr` over other smart pointers.

The base node `ASTNode` defines several static methods and members used by nodes in later steps. Additionally, `ASTNode` defines the interface nodes interact with to perform the later steps. Since all nodes inherit from `ASTNode`, they can access these static methods and members. For semantics checking, `ASTNode` maintains a symbol table and a set of helper methods to assist in discovering semantic errors. For code generation, the `ASTNode` maintains several LLVM data structures, such as an IR builder.

The AST nodes take advantage of a feature of the LLVM library to enable a form of class introspection. The nodes achieve this introspection by requiring the base class to define an enumerated type `ASTNodeKind` and a property of that new type `nodeKind`. Each subclass then defines a static method `classof`, which takes a pointer to an `ASTNode` and compares the `nodeKind` property to one or more of the enumerated values of `ASTNodeKind`. By defining these components in the AST nodes, the LLVM template function `isa` uses the `classof` method of the provided AST node class to determine class information about an AST node object at run-time.

The final AST always has the `TranslationUnitNode` as the root node for a valid program. `TranslationUnitNode` serves as the main interface that the compiler interacts with to perform later steps of compilation. Figure 15 illustrates how the compiler combines the scanner, parser, and `TranslationUnitNode` to build the AST of a QuantC program.

```
std::cout << "Creating scanner\n";
Scanner test_scanner(quant_c_file, std::cout);
std::cout << "Creating parser\n";
TranslationUnitNode ast;
Parser test_parser(test_scanner, ast);
int parse_result = test_parser.parse();
```

Figure 15: Compiler code for building a QuantC program's AST

5.5 Semantics Checking

I implement semantics checking as a method called `checkSemantics` in each AST node. Semantics checking begins when the compiler calls the `TranslationUnitNode` node's `isSemanticsCorrect` method. This method calls the `TranslationUnitNode` node's `checkSemantics` method, which in turn calls the sub-nodes' own `checkSemantics` method. The process of semantics checking continues recursively.

If information needs to flow to different parts of the AST, the nodes provide two methods to achieve this flow. First, nodes store information about identifiers, such as variable or function names, in the symbol table `semanticTable`, which maps the identifier to a `SymbolInfo` object. The `SymbolInfo` object stores the identifier's type and kind information. Kind refers to the purpose an identifier serves in the code. For example, whether the identifier is a variable value, function name, or label.

Classes for type and kind classes encapsulate type information for identifiers. The `Kind` class stores an enumerated value for the identifier's kind. The `Type` class is more complicated. The `Type` class stores three enumerated values for the type, corresponding to the simple type (e.g. integer, character, float), the storage type (e.g. **`extern`**, **`static`**, **`quantum`**), and a quality type (specifically if it is volatile and/or constant). The `Type` class defines multiple constructors for different categories of variables, namely scalar values, arrays, functions, and pointers. In addition to the previous enumeration values, constructors for the non-scalar types also take additional parameters, such as size and a smart pointer to an element type for array types.

For the second method of semantic information flow, nodes pass other kinds of semantic information between themselves via public members defined in each class. In many cases, semantics checking does not need these class-specific members. More complicated cases like `BlockNode` need to pass information about expected `return` statement types to other sub-blocks and `return` statements, justifying the need for this ad hoc method of information flow. In this second method, nodes inspect and create more class-specific pointers if necessary using the `isa` and `cast` functions provided by LLVM (Section 5.4). The class-specific pointers allow a parent node to interact with their sub-nodes' semantic-specific members accordingly.

5.6 Code Generation

I implement code generation in a similar manner to semantics checking. Given that the compiler needs to generate two different IRs in a combined manner, each node defines both a `codegen` and `quantum_codegen` method to handle each type of code generation separately. The compiler always starts code generation by calling the `codegen` method of the root `TranslationUnitNode` in an AST. The code-generation methods, like `checkSemantics`, proceed recursively through the AST.

5.6.1 Classical Code Generation

For classical code generation, each node has a `codegen` method. I take advantage of the LLVM core libraries to ease defining these methods.

The `ASTNode` class stores static members to several LLVM data structures. First is the `LLVMContext`, which manages data such as type tables for the other data structures. Next is the `LLVMModule` object, which acts as a container for other LLVM objects that ultimately represent the code the compiler will generate. Finally, the `Builder` object handles the operations to insert instructions into the `LLVMModule`. `ASTNode` also maintains a static table of `Value` pointers `codegenTable` for methods to easily access values associated with identifiers.

The `codegen` method optionally takes two pointers as arguments: one to an LLVM `Value` object used for parent structures, such as blocks, and the other to an LLVM `Type` object for type information if needed. The latter argument comes into play when generating instructions to allocate memory for variables due to how nested the syntax can get for variable declarations. The `codegen` returns a pointer to a LLVM `Value` object, corresponding to the resulting LLVM instruction objects returned when calling the methods of `Builder`. Expression nodes commonly use this return value for code generation since instructions commonly refer to the results of instructions generated by nested expressions. Using these data structures, code generation in most cases exclusively involves a node calling the `codegen` method(s) of its sub-nodes, then calling methods on `Builder` to generate additional instructions.

The resulting code generated is LLVM IR, an architecture-independent instruction language written in a static single-assignment (SSA) format. This format enables the LLVM libraries to include predefined optimization routines. However, this project does not currently take advantage of those optimizations. Figure 16 illustrates the format the LLVM IR the compiler generates. Once the compiler generates the LLVM IR, it calls the LLVM tool `llc` to generate an object file from the LLVM IR. Figure 17 details these code-generation steps in the compiler.

```
int add(int a, int b) {
    return a + b;
}
```

```
define i32 @add(i32 %0, i32 %1) {
    %3 = alloca i32
    %4 = alloca i32
    store i32 %0, i32* %3
    store i32 %1, i32* %4
    %5 = load i32, i32* %3
    %6 = load i32, i32* %4
    %7 = add nsw i32 %5, %6
    ret i32 %7
}
```

Figure 16: An example of LLVM IR for a simple C function.

```
// Generate LLVM IR for code
ast.setName(std::string(original_path.stem()));
ast.codegen();

// Pull IR from Module into string stream, then write to file
std::string IR_file_name = cd + "/" + file_name + ".ll";
std::string output_content;
llvm::raw_string_ostream instruction_stream(output_content);
instruction_stream << *(ASTNode::LLVMModule);
instruction_stream.flush();
std::fstream output_file(IR_file_name, std::fstream::out);
output_file << output_content;
output_file.close();

// Generate an object file from the IR
std::string obj_command = "llc -relocation-model=pic --filetype=obj ";
obj_command += IR_file_name;
if (system(obj_command.c_str()) == EXIT_FAILURE) {
   std::cout << "Error generating object file";
   return -1;
}
```

Figure 17: Compiler code for code generation.

5.6.2 Quantum Code Generation

For quantum code generation, each node has a `quantum_codegen` method. I created all the data structures in these sections of code. Whenever code generation reaches a `FuncDefNode` (a function definition), that node's `codegen` method first checks the function's type definition to see if its storage type is **`quantum`**. If the function has quantum storage, the `codegen` method performs an entirely separate code generation sequence.

In this alternate sequence, the method creates a `QuantumContext` object, a class meant to operate similar to the LLVM `Builder` class to handle code generation for quantum functions. The function `codegen` method then calls methods on the `QuantumContext` object to provide information about the function name and arguments. The function `codegen` method then calls the `quantum_codegen` method of the body node, passing the `QuantumContext` object by reference to allow lower-level statement and expression nodes to use built-in methods to build the quantum instructions. The `quantum_codegen` method returns a custom `QuantumValue` object, serving an analogous purpose to the return type for `codegen`. The `quantum_codegen` method also maintains its own version of `codegenTable`. Once the body `quantum_codegen` method returns, the `codegen` method concludes by calling the `genQUIL` method on the `QuantumContext` object to get a string value for the generated quantum IR. At this point, the `codegen` method also copies information from `QuantumContext` about the number of qubits allocated for local variables, for use when generating calls to quantum functions. The `codegen` method concludes by using the `LLVMModule` object to store the quantum IR string as a string literal inside the final LLVM IR. In the case of multiple quantum functions, the code generation methods all store their generated quantum IR into one string value.

The generated quantum IR is Quil [10], an architecture-independent quantum instruction language. Figure 19 provides an example of the kind of Quil IR generated by the compiler.

```
quantum int example_func(quantum bool (&qubits)[2]) {
    %H qubits;
    qubits[0] ^ qubits[1];
    return %M qubits;
}
```

```
DECLARE example_func_ro BIT[2];

DEFCIRCUIT example_func qubits_0 qubits_1:
    MOVE example_func_ro 0;
    H qubits_0; H qubits_1;
    CNOT qubits_0 qubits_1;
    MEASURE qubits_0 example_func_ro[0];
    MEASURE qubits_1 example_func_ro[1];
    RESET 0;
    RESET 1;
```

Figure 18: An example of a quantum function and its generated Quil IR.

5.6.3 Handling Classical Calls to Quantum Functions

Similar to how I handle code generation with quantum functions, function call expressions handle quantum functions as a separate case in code generation. In `CallExprNode` (function call expression), the codegen method recursively calls the `codegen` method of its argument expressions. Then it checks if the function to be called has a **`quantum`** storage type. If so, instead of generating a call to this function in LLVM IR, it generates a call to a classical function `quant_com`, which takes the name of the function, the generated quantum IR string, the number of qubits allocated by quantum functions, and the argument expressions. The compiler links in the `quant_com` library code when constructing the executable (detailed in Section 5.7).

The `quant_com` function handles packaging the quantum IR into a message to send to a quantum computer. It generates a call to the quantum function in the provided quantum IR based on the name of the quantum function and the number of previously allocated qubits. In principle, `quant_com` would handle various protocols to transmit the IR. For testing, however, `quant_com` only handles a single type of quantum computing setup, as detailed in Section 5.8.

```
quantum int quant_func() {
    .....
}

.....

int result = quant_func();
```

```
@.quant.str = private unnamed_addr constant i8* c"DEFCIRCUIT quant_func:\n ....."
@.str = private unnamed_addr constant i8* c"quant_func\0A\00"
@qubit_alloc = private global i32 .....

.....

%0 = load i32, i32* @qubit_alloc
%result = call i32 (i8*, i8*, i32, ...)
          @quant_com(i8* getelementptr inbounds (i8, i8* @.str),
                     i8* getelementptr inbounds (i8, i8* @.quant.str,
                     i32 %0)
```

Figure 19: An example of the LLVM IR generated to handle a call to a quantum function from classical code.

5.7 Linker

Like preprocessing, the compiler defers the task of linking to `gcc`. Code generation produces object files that are identical to an object file generated by `gcc`. By using `gcc` to link the object file, the compiler can generate executables that include code from the C standard library, in addition to standalone executables.

5.8 Verifying the Implementation

I conducted testing in an *ad-hoc* manner. Testing consists of manually compiling and running a selection of both C code and QuantC programs to verify expected IR outputs and executable behavior. I acquired most of the C test programs from a publicly available database of C compiler test cases [1]. I wrote additional test cases when necessary to test code generation of certain C constructs, such as `if` statements, isolated from others. Section 4 depicts the QuantC programs I tested.

To test the behavior of these QuantC programs, I installed the Riggetti Computing ForestSDK and configured its QVM included to run as a background process. I configured the QVM to set up an HTTP API and connect to the default `localhost:5000` port. I hard-coded the `quant_com` function to process messages to this kind of QVM on that port. The `qpu_driver` function packages the quantum IR into a JSON message and sends it to the QVM. To handle communication, it uses the cURL library to transmit the JSON message and handle the response message. If the response message indicates that the transmitted IR has executed successfully, `quant_com` converts the response into a JSON object with the `jsoncpp` library. It then parses that object to get the resulting binary string if the quantum function returns a value. It concludes by converting the binary string into an integer value, which it returns. At the location of the call to `quant_com`, the code generated handles casting the return value to the appropriate type.

6 State of the Project

At the time of writing, the QuantC compiler is only partially complete. I have completely defined the QuantC scanner and parser to implement the language's necessary syntactical structures. However, the parser only has actions defined to build a complete AST for the examples programs in Section 4, outside of some simple expression substitutions like binary operations. The compiler only defines a minimal set of semantics checks. These semantics checks primarily consist of setting identifiers' type information and checking that functions that return a value always reach a **`return`** statement. The compiler can generate code described in previous sections for the example programs, but I do not expect any program containing other syntactical structures to compile correctly. Quantum functions cannot generate code for most classical language constructs.

Quantum functions also do not handle assignment with quantum expressions.

All the example QuantC programs compile as expected. The example programs also perform as expected, though I need to verify my implementation of Simon's algorithm to ensure the correct output is not due to the small example function used in the program. In particular, I am not sure if the code I wrote to solve the system of equations and derive the guess value is entirely correct, though the intended meaning of the code is straightforward.

6.1 Future Work

First, I want to finish implementing the language features and semantic checks for the QuantC compiler as described in Section 3. This process first entails implementing the code generation for the unimplemented language features. Afterward, I need to implement the semantics checks in full. When implementing the semantics checks, I intend to further research best practices regarding implementing semantics checking with ASTs to improve the quality of code. My plan is to focus on implementing semantics checks and code generation for quantum functions before moving to classical code.

After finishing with the current QuantC compiler as specified, I need to consider how to handle asynchronous communication channels efficiently. Quantum computers currently operate as shared resources similar to computational nodes in HPC systems. Thus, I/O latency is usually the most significant overhead in a quantum program's run time. One solution is to introduce a variation of `async/await` syntax, as implemented by C++ and JavaScript.

QuantC also faces a hurdle with linking quantum functions. As defined, QuantC does not provide a mechanism to link quantum functions from outside the translation unit statically. I need to modify the compiler to export the Quil IR to a separate file to implement this linking. Afterward, I need to create a custom Quil linker for the Quil IR. Quil does have an `INCLUDE` instruction that operates similarly to the C preprocessor statement. However, that instruction alone is insufficient to generate a fully linked Quil IR. Assuming I could implement this Quil linker, I could modify the compiler to read the linked Quil output and insert the string content into the LLVM module. The final steps of writing the LLVM IR to a file and converting it into an object file would proceed as implemented.

At the moment, QuantC limits the ability to reassign a quantum variable's qubits to a new variable to a one-to-one mapping. A possible addition is adding syntax that would let qubits be arbitrarily bound to a new set of quantum variables — for example, combining a set of quantum variables into a single one or dividing a quantum variable between a set of new ones. I need to research more quantum algorithms to determine how useful this new feature would be.

I could also consider the ability to denote quantum functions containing only quantum variables and expressions. If the compiler is able to determine that a quantum function has that property, it could permit operation modifiers on calls to that quantum function. I could achieve this feature by adding a new storage specifier like `quantum_gate` to indicate to the compiler to further check for classical statements within a quantum function. Quil already supports this concept directly with `DEFCIRCUIT` code. This change would therefore entail minimal changes to code generation. The `toffoli` quantum functions in Figure 11 are prime candidates for this feature.

References

[1] *c-testsuite*. 2018. URL: `https://github.com/c-testsuite/c-testsuite`.

[2] Jutta Degener. 1995. URL: `https://www.lysator.liu.se/c/ANSI-C-grammar-l.html`.

[3] Jutta Degener. 1995. URL: `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`.

[4] Marco Fellous-Asiani et al. "Limitations in quantum computing from resource constraints". In: *PRX Quantum* 2.4 (2021), p. 040335.

[5] Andreas Ketterer. "Modular variables in quantum information". PhD thesis. Oct. 2016, p. 209.

[6] Emmanuel Knill. *Conventions for quantum pseudocode*. Tech. rep. Los Alamos National Lab., NM (United States), 1996.

[7] Ashley Montanaro. "Quantum algorithms: an overview". In: *npj Quantum Information* 2.1 (2016), pp. 1–8.

[8] Michael A Nielsen and Isaac Chuang. *Quantum computation and quantum information*. 2002.

[9] Rxtreme. Dec. 2019. URL: `https://commons.wikimedia.org/wiki/File:Quantum_Logic_Gates.png`.

[10] Robert S Smith, Michael J Curtis, and William J Zeng. "A practical quantum instruction set architecture". In: *arXiv preprint arXiv:1608.03355* (2016).