Qwak
A New Quantum Programming Language

Gabriel Stella[1] and Jonathan Beechner[1]

[1]*Department of Computer Science,*
Texas A&M University, College Station

Abstract

We introduce the Qwak quantum programming language, its design process, and related components. Qwak is a simple, intuitive language for higher-level quantum programming. As opposed to previous work, it attempts to separate the domains of quantum and classical programming by allowing the user to interact with the Qwak system through any classical language of their choice; in this way, Qwak is a *quantum-first* programming language.

1 Introduction

While quantum computers are currently difficult to implement physically, progress is being made in engineering quantum computers with more and more qubits. However, modern quantum programming languages are lacking in several ways. Most notably, Quipper allows the programmer to write invalid quantum programs, while Q# is simply quantum-enabled classical computing. In this project, we developed Qwak, a programming language that focuses solely on quantum computation in order to remain simple and intuitive, leaving classical computation to a language of the programmer's choice.

2 Qwak and its Components

This project involved the design of the Qwak language and the implementation of three projects: first, the Qwality C++ quantum simulator; second, a C++ implementation of a Qwak parser and runtime environment; third, a command-line Qwak interpreter called Qwaket. The state of these projects can be viewed on our GitHub repository at `https://github.com/GabrielRStella/Qwak`.

2.1 Qwality

Qwality is a simple C++ library for quantum simulation. It currently supports the quantum circuit model of computation with pure quantum state vectors. While it supports realistic quantum functionality, e.g. mutable quantum state operations, it also supports a wide variety of simulation-enabled functions such as state cloning, measurement without state collapse, and more. It is based on the GiNaC C++ library for symbolic computation, allowing expressions to be represented with perfect accuracy.

2.2 Qwak

Qwak is a simple language for quantum computation. It is designed to be an intuitive *quantum-first* language that does not attempt to provide extensive classical computation abilities. In order to bridge this gap, Qwak can be controlled from a classical language of the user's choice, or from the Qwaket interactive shell.

The current implementation of Qwak relies on a C++ runtime environment, which links directly to the Qwality simulator. However, our goal for the future design of Qwak is for compilation of Qwak programs to be done in two stages, as seen in Figure 1. The first stage compiles a Qwak program to a simpler intermediate format, similar to bytecode. The second state compiles this intermediate code to a quantum assembly language, such as QASM. The motivation for this two-stage process is that Qwak allows for variables, control flow, functions, and other high-level language constructs that a quantum assembly language may not. The intermediate language, while simpler for a program to compile, must still support these features in some limited way. Thus, the process of compiling this intermediate language can actually be seen as taking a quantum program, along with user-given variable values, to generate a fixed quantum circuit.

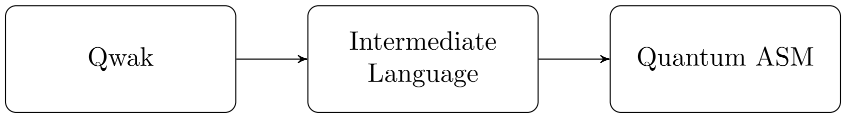

Figure 1: Qwak compilation stages

The ASM itself can then be run on a quantum computer or using a quantum simulator; the Qwak system would not require any knowledge of its use. The separation of the compilation into two stages may also make it easier for interactive shell utilities like Qwaket to be implemented.

As stated before, Qwak is meant to be controlled by a classical programming language. For this to be done, the classical programming language requires a library that can interact with a quantum computer (or simulator) to execute quantum assembly code; it then also requires in-language bindings for compiling Qwak code to quantum assembly to instantiate and execute a circuit. The process of classical control is depicted in Figure 2, where the Quantum Runtime may be a simulator or a quantum computer.

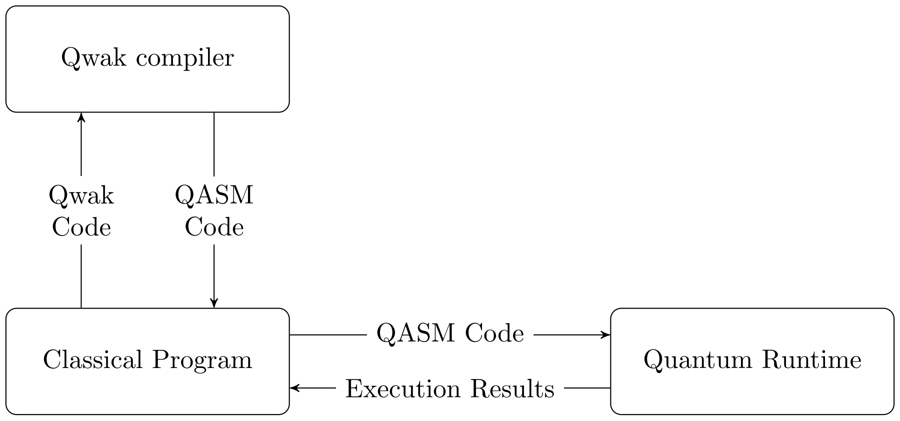

Figure 2: Qwak information flow

Figure 3 shows two short code snippets comparing another quantum programming framework, LIQUi$|\rangle$, to Qwak. We aimed to make Qwak simple and intuitive to those who have worked with quantum algorithms. Its goal is to be a truly quantum language, as opposed to a classical language with quantum capabilities. More details on the Qwak language can be found in the appendices.

2.3 Qwaket

The Qwaket command-line shell, based on our Qwality simulation library, allows the user to interactively experiment with Qwak programming via line-by-line execution and Qwak file parsing. It also supports several convenience features for testing such as the ability to print the global quantum state, the list of variables, LIQUi|⟩

```
let Entangle1(entSiz:int) =
  let ket = Ket(entSiz)
  let _ = ket.Single()
  let qs = ket.Qubits
  H qs
  let q0 = qs.Head
  for i in 1..qs.Length-1 do
    let q = qs.[i]
    CNOT[q0;q]
  M >< qs
```

Qwak

```
function entangle(n) {
  state = |0^n>
  H(state)[0]
  xx = X^**(n-1)
  cx = control(xx)
  cx(state)
  return measure(state)
}
```

Figure 3: Entangling n qubits in LIQUi|⟩ and Qwak.

and the list of loaded functions. Qwaket currently comes with “stl.qwak”, which demonstrates some basic features of the Qwak language.

3 Future Work

There are two general classes of work to be done on the Qwak project. The first relates to the Qwak language; as discussed more in Appendix B, the Qwak language is currently not fully expressive, and more work needs to be done to support more general programs. We also currently have no specification for the Qwak intermediate language. The second class of improvements is to the implementation of Qwality and Qwak.

3.1 Qwality

3.1.1 Improved symbolic algebra computations

Qwality currently uses the GiNaC C++ library for symbolic computation. While it is useful, the library has drawbacks when applied to quantum computation; for example, it is not able to properly simplify very common quantum expressions involving $e^{i\theta}$. We would like to implement a quantum-specific symbolic math library that will enable better handling of the terms that often appear in quantum algorithms.

3.1.2 Noise simulation

Physical implementations of quantum computers experience complicated errors at every step of computation. However, our simulator preserves perfect precision by using symbolic expressions. In future work, artificial noise may be an optional feature of our quantum simulator to help programmers experiment with a more realistic scenario.

3.2 Qwak

The current Qwak implementation is a basic prototype that relies fully on the Qwality simulation library; a proper implementation must be made that follows the compilation process outlined previously in this paper.

3.3 Other Work

3.3.1 Qubit sharing

In future work, we would like to explore the possibility of shared quantum programming; that is, programming "circuits" that span some nontrivial geographic distance, including simulations that run between several computers. This could allow better experimentation with quantum teleportation, super dense coding, and quantum cryptography. This could be facilitated by a quantum state server that allows users to interact with a small substate of the globally-shared quantum state.

Appendices

A Qwality Overview

The Qwality simulation library is composed of two C++ classes: `QuantumState` and `QuantumGate`. As a general design principle, Qwality supports both in-place (marked with an underscore in the source) and out-of-place operations on all objects. For a complete list of supported functions, refer to the Qwak GitHub repository.

A.1 QuantumState

The QuantumState class represents an n-qubit quantum state vector. All qubits within a `QuantumState` are fully connected, and `QuantumState` objects can be combined via the tensor product.

`QuantumState` objects support a general group of important operations: measurement (including partial measurement), tensor product, and gate application. All of these operations include in-place (mutating) and out-of-place (cloning) versions.

One important feature of `QuantumState` is the `applyPartial` function. This allows the programmer to apply an m-qubit gate to a substate of an n-qubit state, where $n > m$, without having to tensor the gate with the identity matrix $n - m$ times. This function also includes automatic swapping of qubits given the programmer's desired order of application.

A.2 QuantumGate

A Qwality `QuantumGate` is simply a square matrix with support for typical quantum functions: tensor product, gate concatenation, inverse, and conjugate transpose. Most importantly, the `QuantumGate` class provides static functions for creating some common quantum gates:

`I`	The single-qubit identity operation.
`H`	The single-qubit Hadamard gate.
`X`	The single-qubit Pauli X gate.
`Y`	The single-qubit Pauli Y gate.
`Z`	The single-qubit Pauli Z gate.
`R(k)`	The single-qubit phase rotation gate R_k.
`control(U)`	Creates a quantum boolean-controlled gate from the unitary U.
`kcontrol(U, m)`	Creates the integer-controlled gate $C_m(U)$.
`swap(qubits)`	Creates a swap gate using the given vector of positions.
`makeUnitary(f)`	Creates a reversible unitary from the given classical function.

B Qwak Language Definition

B.1 Types

QWAK currently supports 3 types: quantum state references, quantum gates, and integer values. In the future, we will work on the integration of classical states, list types, and arbitrary scalar expressions.

Types	
`x = |0>`	A quantum state variable.
`U = H`	A quantum gate variable.
`n = 5`	An integer variable.
Future[1]	
`c = |0>`	A classical state variable.
`xs = [|0>, |1>]`	A list variable.
`ns = [0, 3..6, n..m]`	A list variable defined using ranges.

[1] It is not decided how these will be integrated into the language as of yet. Thus, there may be some ambiguity right now.

B.2 Boolean Values

The truth value of a QWAK variable is essentially like an override of the C++ operator bool(). It is used for things like while loops.

Boolean Values	
Quantum state	Always true.
Quantum gate	Always true.
Integer (or scalar) `n`	True if `n != 0`.
Classical state `c`	True if `c == |1>` (or `c != |0^n>` for `n`-bit states).
List `ls`	True if `dim(ls) != 0`.

B.3 Variables

Variables in QWAK have a type, however they are not *explicitly* typed; variables are created at their first assignment, and deleted when out of scope. Their type is based on the type of their most recent assignment, and can change at any time.

Variables	

`x = |0>`
`x = H`
	Example of a variable's initial assignment, to type *quantum state*, and subsequent reassignment to type *quantum gate*.

Default Variables	
`I`	Single-qubit identity operator.
`H`	Single-qubit Hadamard gate.
`X`	Single-qubit Pauli X gate.
`Y`	Single-qubit Pauli Y gate.
`Z`	Single-qubit Pauli Z gate.
`STATE`	A pseudo-variable that represents the program's entire quantum state.

B.4 Functions

Functions in QWAK are identified by their name, their argument list (count and names), and whether they take a secondary argument of a list of qubits. See Appendix C for more examples.

Functions	

`function name(args)[qubits] {`
`  //function body`
`}`
	General form of a QWAK function.

B.5 Built-in Operators

Builtin operators work just like other QWAK functions, except that they are supported by the QWAK runtime.

Functions	

`dim(x)`
`dim(U)`
`dim(n)`
`dim(c)`
`dim(xs)`
	Retrieves the dimensionality of the given object. The returned value is an integer.

`measure(x)`
`measure(x)[qubits]`
	Measures a substate of the quantum state `x`, in the given order. The measured value is returned as a classical state, and the measured substate of `x` is collapsed.

`zero(x)`
`zero(x)[qubits]`
	Has the effect as if calling `measure(x)`, but the result is always `0`; essentially restores the given substate of `x` to `0`.

`swap(ns)`	Creates a swap gate based on the integer arguments in the list `ns`. The returned gate `U` will be such that `dim(U) == dim(ns)`. The swap is constructed so that the `i`-th qubit inputted will be swapped to the `ns[i]` position in the output.

`control(U)`
`control(U, i)`
	Returns a single-qubit controlled version of the quantum gate `U` such that the `i`-th qubit of the returned gate is the control qubit.

`kcontrol(U, m)`	Returns an integer-controlled version of `U` with `m` control qubits. The control qubits will be the first `m` qubits of the returned gate.
`R(k)`	Returns the phase rotation gate R_k, where k is an integer.

B.6 Statements

Statements form the body of a QWAK function. There are several different types.

Statements	
`return expression`	A return statement. There should be at most one per function, and no other statements after it.
`expression` `identifier = expression`	An assignment. The actual assignment is optional, while the expression is not; many meaningful operations, such as quantum gate application, have no need for an assignment.
`if expression {` `  //if body` `}`	A simple *if* statement. The body is executed if expression evaluates to `true`.
`if expression {` `  //if body` `} else {` `  //else body` `}`	A simple *if-else* statement. If expression evaluates to `true`, the *if body* is executed; otherwise, the *else body* is executed.
`for identifier in expression {` `  //for body` `}`	A simple *for* loop over each element in the result of `expression`. Note this implies that `expression` should evaluate to a list.
`while expression {` `  //while body` `}`	A simple *while* loop. Will execute as long as `expression` evaluates to `true` (according to the truth value of its result).

A note on statement groups: While the above definitions include curly braces, QWAK should treat a statement group (list of statements within curly braces) as its own type of statement, so that the control structures replace body with some arbitrary statement, which may or may not be a group. This opens up more flexibility in the way code is written.

B.7 Expressions

Expressions	
Binary Operators	
`+`	Scalar addition (or list concatenation)
`-`	Scalar subtraction
`*`	Multiplication; works on scalars or quantum gates (gate concatenation = matrix multiplication). Both operands must be of the same type.
`/`	Scalar division
`^`	State concatenation, e.g. `|0^n>`
`**`	Exponent; right-hand operand must be a scalar. Left-hand operand may be a scalar or matrix.
`^*`	Tensor product; works on quantum gates or quantum states. When two quantum state references are the input, this simply returns the combination of both states.
`^**`	Tensor exponent. Due to the no-cloning theorem, this is only supported by quantum gates. The right-hand operand should be an integer.
`==`	Equality. Both operands must be of the same type. The equality expression will evaluate to a scalar; if true, `1`, else `0`. This means that the truth value of `(a == b)` is the same as whether `a` equals `b`. While this isn't technically allowed for quantum states, a Qwak implementation may or may not allow equality checks of arbitrary quantum states.
Other	
`x[qubits]` `c[bits]`	State subscript; returns another (smaller) state. qubits and bits must be integer-valued lists.
`xs[pos]`	List subscript. Returns a single value.
`U(x)` `U(x)[qubits]`	Gate application. Applies the gate `U` to the given qubits (in order) of state `x`. If qubits is not given, `U` is applied (in default order) to the entirety of `x`.
`example(args)` `example(args)[qubits]`	Applies the function example to the list of arguments given in its parentheses. An optional secondary argument of an integer list can be placed directly after the parentheses, just like with a quantum gate application. If qubits is not given, it is as if an empty list is passed as the secondary argument.

B.8 Representation of Quantum States

The Qwak runtime has several large quantum states in the background. When new qubits are created for a variable assignment, a new quantum state is created, the qubits are put into that state, and a reference to the state is returned. Any quantum state referenced from then on will be a substate of that state or other states. These backend quantum states can be tensored and untensored. The mechanisms for the user (programmer) to do this are not clear yet; the current implementation of Qwak only supports a single backend quantum state, with no untensoring.

This makes it clear why operations such as tensoring two states are trivial: if each substate is represented by a list of qubits, then the tensor product of two states is simply the concatenation of their lists.

B.9 Storage of Variables

The only real data in a QWAK program is the underlying quantum state and any quantum gates. Variables can all be seen as references into this "pool" of actual information. Thus, all variables are essentially pointers (with some extra information), and they can be passed by value and created/deleted using simple scope rules. Note that this would however mean that lists would have to become part of the pool; we could probably use some simple reference-counting pointers to handle list garbage collection.

C Qwak Code Snippets

The following snippets show some ideal QWAK code, along with its meaning. Note that these snippets are for clarity; some could be condensed or simplified.

`function KetPlus() {`
`  return H(|0>)`
`}`
	Creates and returns the single-qubit state $\|+\rangle$.

`function Bell() {`
`  s = |00>`
`  cx = control(X)`
`  H(s)[0]`
`  cx(s)`
`  return s`
`}`
	Creates and returns the bell state $\|\phi+\rangle$.

`function Deutsch(U) {`
`  x=|01>`
`  hh = H^*H`
`  hh(x)`
`  U(x)`
`  hh(x)`
`  return measure(x)[0]`
`}`
	Runs Deutsch's algorithm on the given single-qubit unitary U_f, returning classical $\|0\rangle$ if f is constant and $\|1\rangle$ if f is balanced.

`function Deutsch(U) {`
`  x=|01>`
`  hh = H^*H`
`  (hh * U * hh)(x)`
`  return measure(x)[0]`
`}`
	Alternate implementation of the above. Demonstrates application of a quantum gate that is the result of an expression.

`function Entangle(n) {`
`  x = |0^n>`
`  H(x)[0]`
`  xx = X^**(n-1)`
`  cx = control(xx)`
`  cx(x)`
`  return x`
`}`
	Entangles n qubits into the state $\frac{1}{\sqrt{2}}(\|0^n\rangle + \|1^n\rangle)$.

`function Entangle(n) {`
`  x = |0^n>`
`  first = x[0]`
`  H(first)`
`  cx = control(X^**(n-1))`
`  return cx(x)`
`}`
	Alternate implementation of the above. Demonstrates that the value returned by a quantum gate application is the substate that it acted on.