QLAM: An interpreter for Quantum Lambda Calculus

Wil Cram
University of Wisconsin-Madison

December 4, 2025

1 Introduction

The Quantum Lambda Calculus (QLC) is an extension of untyped lambda calculus that offers support for operating on terms in quantum superposition. Originally proposed by van Tonder [3] and further developed by Selinger [2], these models have been shown to be computationally equivalent in power to quantum Turing Machines, and by proxy quantum circuits. While van Tonder gives an operational semantics for such a language, no public implementation exists. In this writeup, we present QLAM, an interpreter written in Rust for a QLC similar to that of [3]. We additionally present a procedure for translating quantum circuits into equivalent QLC terms in linear time and space.

1.1 Background

We first describe how QLC extends the untyped lambda calculus, largely following the model given in [3]. The syntax of the language has been extended with "nonlinear" lambda abstractions $\lambda!x.\, e$ and terms $!(t)$, while the usual lambda abstractions $\lambda x.\, e$ now have linear semantics. That is, the bound variable must be used exactly once in the body. The bound variable of a nonlinear abstraction can be used any number of times in the body just as in classical lambda calculi; note that nonlinear abstractions can only take nonlinear values as input, and likewise for linear abstractions. Nonlinear forms $!(t)$ are considered values, and hence their contents cannot be evaluated further until passed to an abstraction. The calculus is call-by-value, so the input to an abstraction must be fully evaluated before it can be substituted in the body. Evaluation is left-to-right. The language also contains some quantum primitives like the basis states $|0\rangle$ and $|1\rangle$ along with some universal set of quantum gates. The set used by QLAM is $\{\mathrm{H}, \mathrm{T}, \mathrm{CNOT}\}$. The semantics of the quantum primitives is what one would expect: single-qubit gates act on basis states to form superpositions, so that

$$(\mathrm{H}\ \ |0\rangle) \to \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

There are two key things to note from this example: amplitudes are first-class in this model, and superpositions are only created upon application of quantum gates. This language also enjoys a 'congruence' property: no matter how evaluation of a term proceeds, all branches of a superposition will differ only in the values of the basis states.

Two-qubit gates act on pairs of qubits to produce another pair, where we use the Church encoding

$$\texttt{pair} \triangleq \lambda xy.\, \lambda f.\, f\ x\ y.$$

We use the usual syntax (x, y) as a shorthand for such pairs. For example,

$$(\text{CNOT } (|1\rangle, |0\rangle)) \to (|1\rangle, |1\rangle).$$

This definition be extended to n-tuples, which will be important in section 3. An important property to mention here is how our pairs use linear lambda abstractions. This ensures that the components of pairs are not thrown away or duplicated; when supplied with qubits, these pairs will thus abide by the no-cloning and no-deletion principles.

2 Implementation

We now describe some interesting implementation details of QLAM. At the highest level, the objects of the language are `Exprs`. An `Expr` is either a `Term` in the above language containing no superpositions, or a `Superpos` which is a mapping from `Terms` to complex amplitudes. A `Superpos` is created by application of quantum gates to basis states, and can be eliminated by a measurement operator M. This simply picks a branch from the superposition based on the amplitudes; for example,

$$M \ (\text{H } |0\rangle)$$

will evaluate to $|0\rangle$ or $|1\rangle$ with equal probability. Beyond the application of primitives, there is also the question of what happens when a `Term` acts on a `Superpos` and vice versa. This induces m different `Term` $-$ `Term` applications, where m is the number of terms in the superposition. These are already defined, and the output `Exprs` are collected into a new `Superpos`. If the output `Exprs` are all `Superpos` (and if one is, all must be by congruence), the nested superpositions are 'flattened' back into a single `Superpos` by multiplying the joint probabilities. Structurally equal terms also have their amplitudes added during this process. Similarly, for `Superpos` $-$ `Superpos` applications there are $m \cdot m'$ `Term` $-$ `Term` applications, which are then flattened into a single `Superpos` by multiplying joint probabilities.

A consequence of the flattening process along with the above rules is that there is always at most one superposition present during evaluation. Combined with the congruence property, this means that an `Expr` can be described completely by its structure along with a normalized assignment of statevectors to amplitudes. For example,

$$(|\cdot\rangle, |\cdot\rangle) \qquad \{|00\rangle \to 0.5, |01\rangle \to 0.5, |10\rangle \to 0.5, |11\rangle \to 0.5\}$$

determines a maximally-mixed two-qubit state.

Just as pairs can be defined in terms of our primitives, we can also define let-expressions which make it easier to define some quantum algorithms: `let` $x = e$ `in` e' is simply syntactic sugar for $(\lambda x.\, e')\ e$.

QLAM checks terms for well-formedness. Well-formedness is a syntactic property defined in [3] which is a stronger condition than linearity: there is the additional requirement that variables of a nonlinear suspension $!(t)$ must also refer to bound variables in an enclosing nonlinear lambda abstraction. This property is important, as otherwise terms like

$$\lambda x.\, (\lambda !y.\, y\ y)\ !(x)$$

could duplicate linear variables using a nested nonlinear lambda abstraction. This property is preserved under evaluation steps, and therefore need only be checked once before evaluation.

We finally note that the fixpoint operator as defined in section 6 of [3] also works in QLAM, and can be used to define arbitrary-length lists, `map`, and other recursive functions. However, we did not choose to explore these programs as quantum algorithms typically operate on fixed-size registers, making n-tuples a more applicable representation.

3 Compilation from Quantum Circuits

3.1 Motivation

In light of the methods developed in the previous section, transforming the elements of any n-tuple can be accomplished by applying the tuple to an n-ary function that returns another n-tuple. For example, a cyclic permutation of three elements can be realized as

$$\texttt{cycle} \triangleq \lambda t.\, t\ (\lambda xyz.\, \lambda f.\, f\ z\ x\ y)$$

If we remove the binder for t and keep only the expression in parentheses, we obtain a term $\texttt{cycle}'$ that acts from the right. This allows us to apply cycles sequentially without any extra bracketing due to the left-associativity of function application, such that

$$t\ \texttt{cycle}'\ \texttt{cycle}'\ \texttt{cycle}' \equiv t.$$

A quantum circuit is simply a series of transformations applied to an n-tuple of qubits. Therefore, the example above can be generalized to realize any quantum circuit as a QLC term

$$t\ A_1\ A_2\ ...\ A_k,$$

where t is an n-tuple and each A_i is a transformation of n-tuples representing a layer of a quantum circuit.

3.2 Compilation Algorithm

We now provide an algorithm for transforming a quantum circuit C with n wires into an equivalent QLC term, assuming the circuit is constructed from a set G of one and two-qubit gates. A circuit is treated as a list of layers, where each layer is a tensor product of n elements of G. The idea of the algorithm is to turn each layer into an n-tuple transform, and then apply each transform to the start state sequentially.

For layers consisting of only single-qubit gates, the corresponding transform can be realized as

$$[[G_1 \otimes ... \otimes G_n]] \triangleq (\lambda x_1 ... x_n.\, \lambda f.\, f\ (G_1\ x_1)\ ...\ (G_n\ x_n))$$

The two-qubit gates are slightly more challenging, as can be seen from the following example: the expression CNOT (x_1, x_2) evaluates to another pair (x_1', x_2'), but linearity prevents us from using this twice to project x_1' and x_2' in the output tuple. It appears that there is no way to realize two-qubit gates 'locally,' applying the gate directly in the output.

The issue can be worked around by moving the evaluation of all two-qubit gates to the front using let-syntax: $\texttt{let p} = \text{CNOT}\ (x_1, x_2)\ \texttt{in}\ p\ (\lambda x_1' x_2'.\, e)$ allows for the evaluation of e while having access to both x_1' and x_2' in a linear fashion. Therefore, layers containing one and two-qubit gates can be realized by a nested let-expression for each two-qubit gate, culminating in the collection of all outputs into an n-tuple (and unary gates can transform their inputs directly). For example, the layer CNOT $\otimes$ H transforms into

$$(\lambda x_1 x_2 x_3.\, \texttt{let p} = \text{CNOT}\ (x_1, x_2)\ \texttt{in}\ p\ (\lambda x_1' x_2'.\, \lambda f.\, f\ x_1'\ x_2'\ (H\ x_3)))$$

A formal description of the algorithm is given below.

Algorithm 1 Translation of a quantum circuit C into a QLC term

Require: Circuit C on n qubits, input term t

```
Term ← t
for each Layer ∈ C do
    Vs ← [x_1, ..., x_n]
    Context ← □
    for gate G on wires i, j in Layer do
        x'_i, x'_j ← fresh variables
        Bind ← let p = G(Vs[i], Vs[j]) in p (λx'_i x'_j. □)
        Context ← Context[Bind]          ▷ Fill the hole in the evaluation context with Bind
        Vs[i] ← x'_i; Vs[j] ← x'_j
    end for
    for gate U on wire k in Layer do
        Vs[k] ← (U Vs[k])
    end for
    Body ← λf. f Vs[1] ... Vs[n]
    LayerFn ← (λx_1 ... x_n. Context[Body])
    Term ← Term LayerFn
end for
return Term
```

Because each gate in a layer adds at most one let-expression, the construction of the corresponding term requires $O(n)$ time and space. It follows immediately that the above procedure has time and space cost $O(nk)$, where k is the depth of the circuit.

4 Conclusion

4.1 Results

In order to ensure that QLAM behaves in a similar way to that of other models of QLC, we have interpreted a majority of the examples in section 7 of [3]. The QFT example was skipped as the given definition relies on a classical procedure for constructing gates implementing a conditional phase of $2\pi i/2^n$. The completed examples include Deutsch's algorithm, Bell state preparation, and quantum teleportation. We also constructed equivalent quantum circuits for each of the examples above and verified that the circuit translation routine produces the same output state as the hand-constructed QLC terms.

To stress-test QLAM, we also used hyperfine to benchmark the evaluation of terms generating the n-dimensional maximally mixed state. As expected of our approach to storing states, there is an exponential blowup in time and space as the dimensionality increases.

4.2 Future Work

As it stands, there are many ways to extend the semantics of QLAM to model different computational behavior. One promising approach is the introduction of classical control: rather than returning measured basis states, measured qubits could return nonlinear classical bits that could then be eliminated via a classical if-statement. One could also explore adding types to a QLC; ideally, this type system should be linear and dependent as in [1]. Finally, constructing an explicit

Length	Mean Time
1	1.7 ms
2	1.7 ms
3	1.9 ms
4	2.0 ms
5	2.5 ms
6	3.3 ms
7	6.5 ms
8	18.1 ms
9	61.4 ms
10	233.6 ms
11	951.7 ms
12	4.071 s

Table 1: The mean runtime for QLAM to evaluate a term producing $H^{\otimes n} |0\rangle^{\otimes n}$.

translation in the other direction should be considered; turning an arbitrary QLC term into a circuit appears considerably more difficult.

4.3 Remarks

In this writeup, we have showcased QLAM, an interpreter for a model of QLC. We have also demonstrated a routine for translating quantum circuits of one and two-qubit gates into an equivalent QLC term with time and space complexity linear in the depth and width of the circuit. These tools were used to construct some elementary quantum algorithms in QLC.

QLC, while being useful theoretical models of quantum computation and possibly more amenable to proof, are still quite difficult to work with when building programs of reasonable size (in fact, the circuit compilation routine was born out of being tired of constructing terms by hand). However, we believe this area is still worth studying due to the great influence of classical lambda calculi on contemporary programming language theory and practice. In the future, we hope that work on QLC can similarly advance the state of programming languages for quantum computers.

References

[1] Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages. *Logical Methods in Computer Science*, Volume 18, Issue 3, September 2022.

[2] Peter Selinger and Benoît Valiron. *A Lambda Calculus for Quantum Computation with Classical Control*, page 354–368. Springer Berlin Heidelberg, 2005.

[3] André van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, January 2004.