Retrodictive Execution of Quantum Circuits

Jacques Carette McMaster University
Gerardo Ortiz* Indiana University
Amr Sabry Indiana University

January 24, 2022

Abstract

1 Introduction

Retrodictive quantum theory [3], retrocausality [1], and the time-symmetry of physical laws [9] suggest that partial knowledge about the future can be exploited to understand the present. We demonstrate the even stronger proposition that, in concert with the computational concepts of *demand-driven lazy evaluation* [5] and *symbolic partial evaluation* [4], retrodictive reasoning can be used to de-quantize some quantum algorithms, i.e., to provide efficient classical algorithms inspired by their quantum counterparts.

We begin by introducing the principles of demand-driven lazy evaluation, symbolic partial evaluation, and then apply them to the quantum circuit model to de-quantize quantum algorithms.

Lazy Evaluation. Consider a program that searches for three different numbers x, y, and z each in the range $[1..n]$ and that sum to s. A well-established design principle for solving such problems is the *generate-and-test* computational paradigm. Following this principle, a simple program to solve this problem in the programming language Haskell is:

```
generate :: Int -> [(Int,Int,Int)]
generate n =  [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n]]

test :: Int -> [(Int,Int,Int)] -> [(Int,Int,Int)]
test s nums = [(x,y,z) | (x,y,z) <- nums, x /= y, x /= z, y /= z, x+y+z == s]

find :: Int -> Int -> (Int,Int,Int)
find s =  head . test s . generate
```

; The program consists of three functions: `generate` that produces all triples `(x,y,z)` from `(1,1,1)` to `(n,n,n)`; `test` that checks that the numbers are different and that their sum is equal to `s`; and `find` that composes the two functions: generating all triples, testing the ones that satisfy the condition, and returning the first solution. Running this program to find numbers in the range [1..6] that sum to 15 immediately produces $(4, 5, 6)$ as expected. But what if the range of interest was [1..10000000] ? A naïve execution of the generate-and-test method would be prohibitively expensive as it would spend all its time generating an enormous number of triples that are un-needed.

Lazy demand-driven evaluation as implemented in Haskell succeeds in a few seconds with the result $(1, 2, 12)$, however. The idea is simple: instead of eagerly generating all the triples, generate a process that, when queried, produces one triple at a time on demand. Conceptually the execution starts from the observer side which is asking for the first element of a list; this demand is propagated to the function `test` which itself propagates the demand to the function `generate`. As each triple is generated, it is tested until one triple passes the test. This triple is immediately returned without having to generate any additional values.

Partial Evaluation. Below is a Haskell program that computes a^n by repeated squaring:

```
power :: Int -> Int -> Int
power a n
  | n == 0     = 1
  | n == 1     = a
  | even n     = let r = power a (n `div` 2) in r * r
  | otherwise  = a * power a (n-1)
```

When both inputs are known, e.g., `a = 3` and `n = 5`, the program evaluates as follows:

```
   power 3 5
=  3 * power 3 4
=  3 * (let r1 = power 3 2 in r1 * r1)
=  3 * (let r1 = (let r2 = power 3 1 in r2 * r2) in r1 * r1)
=  3 * (let r1 = (let r2 = 3 in r2 * r2) in r1 * r1)
=  3 * (let r1 = 9 in r1 * r1)
=  243
```

Partial evaluation is used when we only have partial information about the inputs. Say we only know $n = 5$. A partial evaluator then attempts to evaluate `power` with symbolic input `a` and actual input `n=5`. This evaluation proceeds as follows:

```
   power a 5
=  a * power a 4
=  a * (let r1 = power a 2 in r1 * r1)
=  a * (let r1 = (let r2 = power a 1 in r2 * r2) in r1 * r1)
=  a * (let r1 = (let r2 = a in r2 * r2) in r1 * r1)
=  a * (let r1 = a * a in r1 * r1)
=  let r1 = a * a in a * r1 * r1
```

All of this evaluation, simplification, and specialization happens without knowledge of `a`. Just knowing `n` was enough to produce a residual program that is much simpler.

Quantum Circuit Model. Many quantum algorithms can be expressed using circuits consisting of three stages: preparation, unitary evolution, and measurement. For a large number of quantum algorithms, the unitary evolution is a quantum oracle U_f that encapsulates a classical function f to be analyzed as shown in Fig. 1.

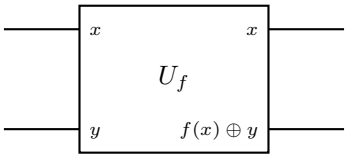


Figure 1: Quantum oracle

In the conventional execution model of quantum circuits, which is the conventional way to use quantum mechanics as a predictive theory, the U_f block receives both inputs and evolves in the forwards direction to produce the outputs. The preceding discussion about lazy evaluation and partial evaluation suggests, however, more creative ways to execute the U_f block as shown in Fig. 2. In this scenario, the second register has both a known initial and final condition. Using this knowledge, it is possible to perform a demand-driven partial evaluation with a symbolic placeholder for x. This execution produces information about the initial state which then determines x.

The circuit in Fig. 3 explains the idea in a simple but realistic scenario. The circuit uses a hand-optimized U_f that implements the modular exponentiation $4^x \mod 15$ in order to factor 15 using Shor's algorithm. In a conventional execution, the execution proceeds as follows. At step (1), we have the initial state $(1/2\sqrt{2}) \sum_{i=0}^{7} |i\rangle |0\rangle$. The state evolves through the U_f block between (1) and (2) to become:

$$\frac{1}{2\sqrt{2}}((|0\rangle + |2\rangle + |4\rangle + |6\rangle)|1\rangle + (|1\rangle + |3\rangle + |5\rangle + |7\rangle)|4\rangle)$$

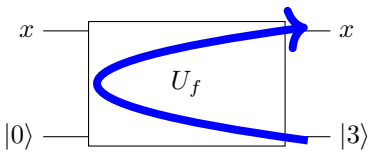


Figure 2: Retrodictive execution flow

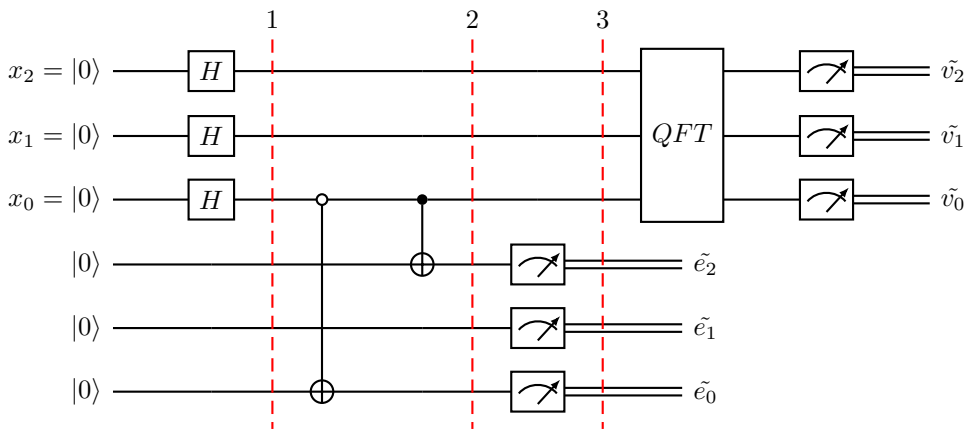


Figure 3: Finding the period of $4^x \mod 15$

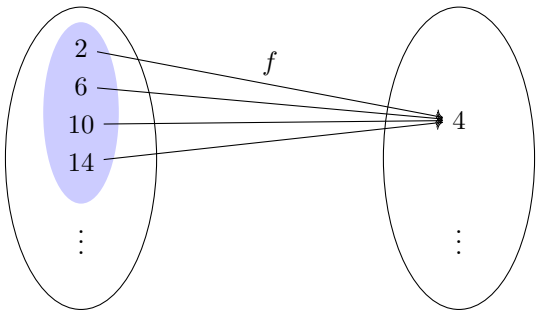
At this point, the second register is measured. The result of the measurement can be either $|1\rangle$ or $|4\rangle$. In either case, the top register snaps to a state of the form $\sum_{r=0}^{3} |a + 2r\rangle$ whose QFT has peaks at $|0\rangle$ or $|4\rangle$. If we measure $|0\rangle$ we repeat the experiment; otherwise we infer that the period is 2.

Instead of this forward execution, we can reason as follows. Since $x^0 = 1$, we know that $|1\rangle$ is a possible measurement of the second register. We can therefore proceeds in a retrodictive fashion with the state $|x_2x_1x_0\rangle\,|001\rangle$ at step 2 flowing backwards. In this symbolic execution, the first cnot-gate changes the state to $|x_2x_1x_0\rangle\,|x_001\rangle$ and the second cnot-gate produces $|x_2x_1x_0\rangle\,|x_0x_0x_0\rangle$. At that point, we reconcile the retrodictive result of the second register $|x_0x_0x_0\rangle$ with the initial condition $|000\rangle$ to conclude that $x_0 = 0$. In other the first register must be in a state of the form $|xx0\rangle$ where the least significant bit must be 0 and the other two bits are unconstrained. Expanding the possibilities, the first register can only be in a superposition of the states $|000\rangle, |010\rangle, |100\rangle$ or $|110\rangle$. The period is 2 and we have implemented this period-finding algorithm by purely classical reasoning about the modular exponentiation circuit!

In order to assess whether this idea works for a broader class of situations including different algorithms and different circuit sizes, we implemented the demand-driven symbolic partial evaluator and ran it on a variety of circuits.

2 Alternative Introduction

Given finite sets A and B, a function $f : A \to B$ and an element $y \in B$, we define $\{\cdot \xleftarrow{f} y\}$, the pre-image of y under f, as the set $\{x \in A \mid f(x) = y\}$. For example, let $A = B = \{0, 1, \ldots, 15\}$ and let $f(x) = 7^x \mod 15$, then the collection of values that f maps to 4, $\{\cdot \xleftarrow{f} 4\}$, is the set $\{2, 6, 10, 14\}$.



Finding the pre-image of a function is a mathematical question that subsumes several practical computational problems such as pre-image attacks on hash functions [8], predicting environmental conditions that allow certain reactions to take place in computational biology [2, 6], and finding the pre-image of feature vectors in the space induced by a kernel in neural networks [7].

To appreciate the difficulty of computing pre-images in general, note that SAT is a boolean function over the input variables and that solving a SAT problem is asking for the pre-image of `true`. Indeed, based on the conjectured existence of one-way functions which itself implies $P \neq NP$, all these pre-images calculations are believed to be computationally intractable in their most general setting. What is however intriguing is that many computational problems that have efficient quantum algorithms are essentially queries over pre-images. We illustrate this connection briefly in the remainder of this section and analyze it further in the remainder of the paper.

Let $[\mathbf{n}]$ denote the finite set $\{0, 1, \ldots, (n-1)\}$. The parameter n determines the problem size in all the problems below (except Deutsch which is a fixed sized problem).

Deutsch. The conventional statement of the problem is to determine if a function $[\mathbf{2}] \rightarrow [\mathbf{2}]$ is constant or balanced. An equivalent statement is to answer a query about the cardinality of a pre-image. In this case, if the cardinality of the pre-image of any value in the range is even i.e. 0 or 2, the function must be constant and if it is odd, i.e., it contains just one element, the function must be balanced.

Deutsch-Jozsa. The problem is a generalization of the previous one: the question is to determine if a function $[\mathbf{2^n}] \rightarrow [\mathbf{2}]$ for some n is constant or balanced. When expressed as a pre-image computation, the problem reduces to a query distinguishing the following three situations about the pre-image of a value in the range of the function: is the cardinality of the pre-image equal to 0, 2^n, or 2^{n-1}? In the first two cases, the function is constant and in the last case, the pre-image contains half the values in the domain indicating that the function is balanced.

Bernstein-Vazirani. We are given a function $f : [\mathbf{2^n}] \rightarrow [\mathbf{2}]$ that hides a secret number $s \in [\mathbf{2^n}]$. We are promised the function is defined using the binary representations $\sum_i^{n-1} x_i$ and $\sum_i^{n-1} s_i$ of x and s respectively as follows:

$$f(x) = \sum_{i=0}^{n-1} s_i x_i \mod 2$$

The goal is to determine the secret number s.

Expressing the problem as a pre-image calculation is slightly more involved than in the previous two cases. To determine s, we make n queries to the pre-image of a value in the range of the function. Query i asks whether 2^i is a member of the pre-image and the answer determines bit i of the secret s. Indeed, by definition, $f(2^i) = s_i$ and hence s_i is 1 iff 2^i is a member of the pre-image of 1.

Simon. We are given a 2-1 function $f : [\mathbf{2^n}] \rightarrow [\mathbf{2^n}]$ with the property that there exists an a such $f(x) = f(x \oplus a)$ for all x; the goal is to determine a. When expressed as a computation of pre-images, the problem statement becomes the following. Pick an arbitrary x and compute the pre-image of $f(x)$. It must contain exactly two values one of which is x. The problem then reduces to finding the other value in the pre-image.

Shor. The quantum core of the algorithm is the following. We are given a periodic function $f(x) = a^x \mod 2^n$ and the goal is to determine the period. As a computation over pre-images, the problem can be recast as follows. For an arbitrary x, compute the pre-image of $f(x)$ and query it to determine the period.

References

[1] Yakir Aharonov and Lev Vaidman. "The Two-State Vector Formalism: An Updated Review". In: *Time in Quantum Mechanics*. Ed. by J.G. Muga, R. Sala Mayato, and Í.L. Egusquiza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 399–447.

[2] Tatsuya Akutsu, Morihiro Hayashida, Shu-Qin Zhang, Wai-Ki Ching, and Michael K Ng. "Analyses and algorithms for predecessor and control problems for Boolean networks of bounded indegree". In: *Information and Media Technologies* 4.2 (2009), pp. 338–349.

[3] Stephen M. Barnett, John Jeffers, and David T. Pegg. "Quantum Retrodiction: Foundations and Controversies". In: *Symmetry* 13.4 (2021).

[4] Yoshihiko Futamura. "Partial computation of programs". In: *RIMS Symposia on Software Science and Engineering*. Ed. by Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 1–35.

[5] Peter Henderson and James H. Morris. "A Lazy Evaluator". In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. POPL '76. Atlanta, Georgia: Association for Computing Machinery, 1976, pp. 95–103.

[6] Johannes Georg Klotz, Martin Bossert, and Steffen Schober. "Computing preimages of Boolean networks". In: *BMC Bioinformatics* 14.10 (Aug. 2013), S4.

[7] J.T.-Y. Kwok and I.W.-H. Tsang. "The pre-image problem in kernel methods". In: *IEEE Transactions on Neural Networks* 15.6 (2004), pp. 1517–1525.

[8] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption*. Ed. by Bimal Roy and Willi Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388.

[9] Satosi Watanabe. "Symmetry of Physical Laws. Part III. Prediction and Retrodiction". In: *Rev. Mod. Phys.* 27 (2 Apr. 1955), pp. 179–186.