Symbolic Execution of Hadamard-Toffoli Quantum Circuits

Jacques Carette
Department of Computing and
Software
McMaster University
Hamilton, Canada
carette@mcmaster.ca

Gerardo Ortiz
Department of Physics
Indiana University
Bloomington, USA
ortizg@iu.edu

Amr Sabry
Department of Computer Science
Indiana University
Bloomington, USA
sabry@iu.edu

Abstract

The simulation of quantum programs by classical computers is a critical endeavor for several reasons: it provides proof-of-concept validation of quantum algorithms; it provides opportunities to experiment with new programming abstractions suitable for the quantum domain; and most significantly it is a way to explore the elusive boundary at which a quantum advantage may materialize. Here, we show that traditional techniques of symbolic evaluation and partial evaluation yield surprisingly efficient classical simulations for some instances of textbook quantum algorithms that include the Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor's algorithms. The success of traditional partial evaluation techniques in this domain is due to one simple insight: the quantum bits used in these algorithms can be modeled by a symbolic boolean variable while still keeping track of the correlations due to superposition and entanglement. More precisely, the system of constraints generated over the symbolic variables contains all the necessary quantum correlations and hence the answer to the quantum algorithms. With a few programming tricks explained in the paper, quantum circuits with millions of gates can be symbolically executed in seconds. Paradoxically, other circuits with as few as a dozen gates take exponential time. We reflect on the significance of these results in the conclusion.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Operational semantics**; • **Computing methodologies → Symbolic and algebraic algorithms**; • **Applied computing → Physics**.

Keywords: quantum computation, partial evaluation, symbolic evaluation, retrodictive quantum computing, algebraic normal form, boolean circuits, quantum oracles

1 Introduction

Classical models of computation are widely believed to be less powerful than quantum ones. Nevertheless, it is of utmost importance to establish when, for a given problem, a classical algorithm is as resource efficient as its quantum counterpart. In this paper we address this question from a new angle, apparently not explored before. We consider traditional techniques of symbolic and partial evaluation to classically simulate quantum circuits assembled from Hadamard and Toffoli gates. The latter constitutes a set of quantum gates that is known to be computationally universal [2].

Using partial evaluation to optimize reversible or quantum circuits is not itself a novel idea (see, e.g., [5, 31]). What distinguishes our approach are the following two important observations. Firstly, since quantum algorithms are reversible, depending on the problem at hand, one can always take advantage of "backwards-in-time" execution (known as *retrodictive* execution [3, 7, 15, 41]). Secondly, since Hadamard is a purely quantum gate, with no classical counterpart, we need to eliminate the superposition generated by such a gate and replace it with a symbolic variable that preserves the relevant quantum correlations. This turns out to be straightforward for instances of Hadamard gates used in the first stage of quantum algorithms to introduce a uniform superposition of all relevant inputs. These two new ideas, while not expressive enough to model arbitrary quantum computations, are effective for instances of standard examples of quantum algorithms.

Our approach can be broadly related to other classical approaches to reason about subsets of quantum circuits, e.g., without entanglement [30], or for Clifford groups [4, 23], but with different tradeoffs: we allow arbitrary entanglement at the cost of sometimes generating exponentially large equations and we deal with an incomparable set of gates compared to the Clifford group.

Outline. We begin in Sec. 2 with background information on quantum computing with a focus on the idea of representing a class of qubit states using symbolic variables. Sec. 3 shows how standard quantum gates can be modeled using this symbolic representation by using the algebraic normal form (ANF) of boolean formulae. Sec. 4 reviews the family of quantum algorithms that is the focus of our approach and highlights the new perspective brought by symbolic execution. Our main technical contribution, the design and implementation of a symbolic evaluator for Hadamard-Toffoli quantum circuits, is explained in Sec. 5. The next section (Sec. 6) includes a complexity analysis and a performance evaluation of our symbolic evaluator on major textbook algorithms. Sec. 7 concludes with a summary and a discussion of the broader implications of our approach to the understanding of the classical / quantum performance characteristics. Our code is available online.[1]

2 Qubits as Symbolic Variables

The general state of a quantum bit (qubit) is mathematically modeled using an equation parameterized by two angles θ and ϕ as follows:

$$\cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle\,.$$

The description models the fact that the qubit is in a superposition of false $|0\rangle$ and true $|1\rangle$. The angle θ determines the relative amplitudes of false and true and the angle ϕ determines the relative phase between them. A particular case when $\theta = \pi/2$ and $\phi = 0$ is ubiquitous in quantum algorithms. In those cases, the general representation reduces to:

$$1/\sqrt{2}\,(|0\rangle + |1\rangle),$$

which represents a qubit in an equal superposition of false and true and with no relative phase between them.

The reason this particular case is distinguished is because a rather common template for quantum algorithms is to start with qubits initialized to $|0\rangle$ and immediately apply a Hadamard H transformation whose action is:

$$|0\rangle \mapsto 1/\sqrt{2}\,(|0\rangle + |1\rangle).$$

This superposition is then further manipulated depending on the algorithm in question.

Our observation is that a qubit in the special superposition $1/\sqrt{2}\,(|0\rangle + |1\rangle)$ is, computationally speaking, indistinguishable from a symbolic boolean variable with an unknown value in the same sense used in symbolic evaluation of classical programs [6, 10, 16, 25, 28]. First, the superposition is not observable. The only way to observe the qubit is via a measurement which collapses the state to be either false or true with equal probability. Second, and more significantly, this remarkably simple observation is quite robust even in the presence of multiple, possibly entangled, qubits.
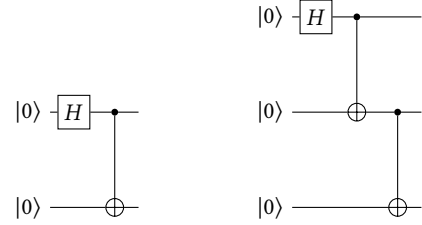
Figure 1. Circuits for constructing Bell and GHZ States

To see this, consider the conventional quantum circuits for creating the maximally entangled Bell and GHZ states in Fig. 1. On the left, the circuit generates the Bell state $(1/\sqrt{2})\,(|00\rangle + |11\rangle)$ as follows. First the state evolves from $|00\rangle$ to $(1/\sqrt{2})\,(|00\rangle + |10\rangle)$. Then we apply the cx-gate whose action is to negate the second qubit when the first one is true. By using the symbol x_1 for $H\,|0\rangle$, the input to the cx-gate is $|x_1 0\rangle$. A simple case analysis shows that the action of cx-gate on inputs $|x_1 y_1\rangle$ is $|x_1 (x_1 \oplus y_1)\rangle$ where $\oplus$ is the exclusive-or boolean operation. In other words, the cx-gate transforms $|x_1 0\rangle$ to $|x_1 x_1\rangle$. Since any measurement of the Bell state must produce either 00 or 11, a symbolic state that shares the same name in two positions accurately represents correlations of the entangled Bell state. Similarly, for the GHZ circuit on the right of Fig. 1, the state after the Hadamard gate is $|x_1 00\rangle$ which evolves to $|x_1 x_1 0\rangle$ and then to $|x_1 x_1 x_1\rangle$ again accurately capturing the entanglement correlations.

Because quantum circuits are reversible, i.e., executable forwards and backwards, the introduction of symbolic variables opens a host of new exciting possibilities beyond conventional (classical) symbolic evaluation: for a given quantum circuit *any mixture of inputs and outputs can be deemed symbolic.* For example, consider again the Bell circuit in Fig. 2 but with an arbitrary initial value for the second qubit. The right subfigure (Fig. 2b) removes the explicit use of $H\,|0\rangle$ and replaces the top qubit with another symbolic variable. Because quantum circuits are reversible, we can, at this point, "partially evaluate" the circuit under various regimes. For example, we can set $y_1 = 0$ and $y_2 = 1$ and ask about values of x_1 and x_2 that would be consistent with this setting. We can calculate backwards from $|x_2 1\rangle$ as follows. The state evolves to $|x_2(1 \oplus x_2)\rangle$ which can be reconciled with the initial conditions yielding the constraints $x_1 = x_2$ and $1 \oplus x_2 = 0$ whose solutions are $x_1 = x_2 = 1$.

Technically, the problem of symbolic evaluation of our quantum circuits then reduces to a mixture of partial evaluation, slicing, and symbolic evaluation. Like with partial evaluation, we have some inputs dynamic, some static, and a static program. Similarly for slicing, but with outputs. Our situation is significantly simpler than in both cases. First, our language is reversible, which makes backwards evaluation deterministic, unlike for most languages. Second, the values

(a) Bell circuit   **(b)** Symbolic variant

Figure 2. A conventional quantum circuit for generating a Bell state (a); its classical symbolic variant (b)

at each step of circuit execution are boolean functions manipulated with conditional exclusive-or operations with a well-understood normal form, ANF, explained next.

3 Algebraic Normal Form (ANF)

The circuits we are interested in can all be expressed in terms of *generalized Toffoli gates* with n control qubits: $a_0, \cdots, a_{n-1}$ and one target qubit c, where the effect is to leave all the control qubits unchanged and send c to $c \oplus \bigwedge_i a_i$, the exclusive-or of the target c with the conjunction of all the control qubits. In fact, we generalize this further, so that we can control either on a qubit or its negation, by using pairs of a control qubit and a boolean. In other words, our gates are specified by a collection $(a_0, b_0), \cdots, (a_{n-1}, b_{n-1})$ together with one target qubit c; their action is to send the target qubit to $c \oplus \bigwedge_i (a_i == b_i)$, the exclusive-or of the target c with the conjunction of the result of testing each qubit against its corresponding target boolean. Note that $(a_i == 1)$ can be expressed as just a_i, and $(a_i == 0)$ can be expressed as $1 \oplus a_i$. Such generalized Toffoli gates with n control qubits are called $c^n x$ gates. It is worth noting the following special cases:

- for $n = 0$, we get the *not* gate x,
- for $n = 1$, we get the *controlled not* gate cx, and
- for $n = 2$, we get the *controlled controlled not* or the conventional *Toffoli* gate ccx.

The *algebraic normal form* [37, 42] (ANF also called ring sum normal form, Zhegalkin normal form or Reed-Muller expansion) of boolean functions is the exclusive-or of $\wedge$-clauses where each clause is the conjunction of 0 or more inputs x_i. Note that the conjunction of 0 inputs is 1 and that $1 \oplus F$ is the negation of F which means that negation is not needed as a separate primitive. It is then easy to see that generalized Toffoli gates are essentially in ANF and that symbolic circuit evaluation can proceed by maintaining the ANF representation of the circuit. Furthermore, circuits that only use x and cx-gates never generate any conjunctions and hence lead to formulae that are efficiently solvable classically [37, 42].

The ANF of a boolean function is unique. This means that two different circuits implementing the same function have the same symbolic ANF representation. As a small example, Fig. 3 shows an equivalence that is often useful when only nearest-qubit interactions are available. Fig. 4 shows two fragments of a circuit used in Shor's algorithm that each use one side of the equivalence. We demonstrate, manually, how

the ANF symbolic representation of the two circuits is unique, which explains why a non-optimized circuit with millions of gates can be efficiently processed if the underlying function has an efficient circuit representation.

Let's start symbolically evaluating the circuit on the left of Fig. 4. Replacing $H \ket{0}$ by z for the top wire, the initial state for the symbolic execution is $\ket{z000}$. The white dot in the graphical representation of the first gate indicates that the control is active when it is 0. The state proceeds to $\ket{z00(1 \oplus z)}$. The second gate then produces $\ket{zz0(1 \oplus z)}$ as the final state. For the circuit on the right, symbolic evaluation proceeds as follows:

$$
\begin{aligned}
\ket{z000} &\mapsto \ket{z00(1 \oplus z)} \\
&\mapsto \ket{z0z(1 \oplus z)} \\
&\mapsto \ket{zzz(1 \oplus z)} \\
&\mapsto \ket{zz0(1 \oplus z)} \\
&\mapsto \ket{zz0(1 \oplus z)},
\end{aligned}
$$

which is the same ANF representation as the first circuit.

To summarize, we never need to residualize a circuit, we can always get a "closed form," in ANF, for evaluation, whether forward or backward. Evaluating a circuit in one fixed direction is then quite standard. A novel approach, enabled by the reversibility of quantum mechanics, is what we call the *retrodictive* mode of running circuits, as explained in our preprint [15] which further details the quantum and physics side of this work (but does not speak of the implementation beyond saying that it exists). In that mode, illustrated in Fig. 5b , we start execution in the forward direction with a fully static collection of inputs in order to partially determine a possible future; we then execute backwards from the partially specified possible future (with the unknown values represented symbolically). This combination of static and dynamic knowledge of the output produces as its result a *system of constraints* equating the resulting logical polynomials to the circuit's inputs. What we will actually see is that for many quantum circuits, we can "read off" the information we need from the system of constraint themselves, *without needing to actually solve them*.

4 Quantum Algorithms

Let $[2^n]$ denote the finite set $\{0, 1, \ldots, (2^n - 1)\}$ with elements generically denoted as x. The integer $n > 0$ determines the problem size for all the problems below. In the
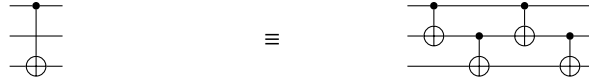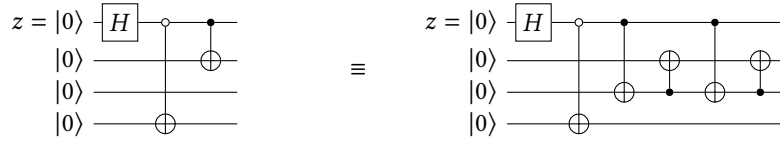
Figure 3. A gate equivalence

Figure 4. A circuit equivalence using the gate equivalence in Fig. 3

review below, we adapt the usual presentation of the algorithms [8, 19, 20, 24, 32, 34, 35] to one better suited to our context. In particular, we focus on the heart of the algorithm, the quantum oracle, which encapsulates the underlying boolean function of interest. Furthermore, instead of using the forward flow of execution using exact quantum superpositions, we express the problem as one asking for particular properties of the pre-image of the classical function embedded in the quantum oracle.

Deutsch. The conventional statement of the problem is to determine if a function $f : [2] \rightarrow [2]$ is constant or balanced. In this small case, there are just four possible functions; the function is balanced if it is the identity $f(x) = x$, or boolean negation $f(x) = 1 \oplus x$, and is constant otherwise. Equivalently, we can ask about the pre-image of an arbitrary boolean value (say false), i.e., the set of inputs that are mapped to false by the function, and check whether the pre-image has an even or odd number of elements. If the cardinality of the pre-image 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 $f : [2^n] \rightarrow [2]$ 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 $f : [2^n] \rightarrow [2]$ that hides a secret number $s \in [2^n]$. 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 computation is slightly more involved than in the previous two cases. To determine s, we compute the pre-image of a value in the range of the function, and then make n queries to this pre-image. 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 $f : [2^n] \rightarrow [2^n]$ with the property that there exists an a such $f(x) = f(x \oplus a)$ for all x where $\oplus$ in this context is bitwise exclusive-or; the goal is to determine $a \in [2^n]$. 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.

Grover. We are given a function $f : [2^n] \rightarrow [2]$ such that there is a unique $u \in [2^n]$ such that $f(u) = 1$. The problem is to find this u.

Shor. We are given a periodic function $f(x) = a^x \mod 2^n$, such that $a < 2^n$ with $\gcd(a, 2^n) = 1$, 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.

Template for Circuits. All the problems above have solutions using quantum circuits that all fit the template in Fig. 5. The U_f block, often called the "oracle," is uniformly defined as:

$$U_f(|x\rangle |y\rangle) = |x\rangle |f(x) \oplus y\rangle , \tag{1}$$

for all the problems. We also use the Quantum Fourier Transform (QFT) uniformly as the last step in all the circuits although for most circuits, with the notable exception of Shor's algorithm, the low precision approximation of QFT (which is the Hadamard gate) is sufficient [18].

After replacing $H |0\rangle$ by a symbolic variable, the U_f block ends up being completely classical, albeit performing mixed mode execution of the circuit. More precisely, it means that
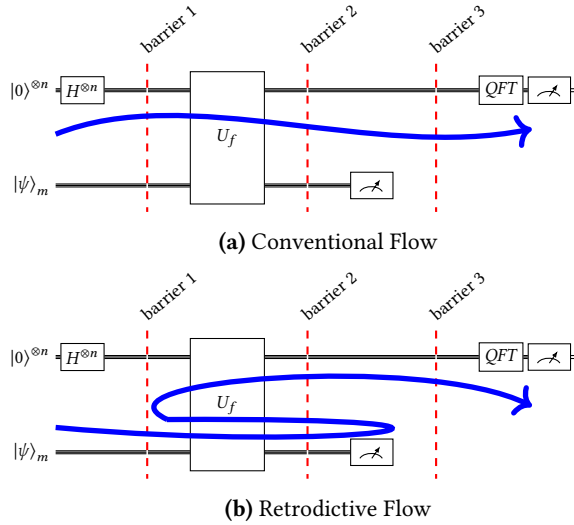
(a) Conventional Flow

(b) Retrodictive Flow
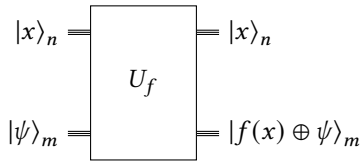
Figure 5. Template quantum circuit

Figure 6. Circuit Abstraction

in all these algorithms, the top collection of wires (which we will call the input register) is prepared in a uniform superposition which can be represented using symbolic variables. The measurement of the bottom collection of wires (which we call the output register) after barrier 2 provides partial information about the future which is, together with the initial conditions of the output register, sufficient to symbolically execute the circuit. In each case, instead of the conventional execution flow depicted in Fig. 5(a), we find a possible measurement outcome w at barrier 2 and perform a symbolic retrodictive execution with a state $|xw\rangle$ going backwards to collect the constraints on x that enable us to solve the problem in question.

In other words, from the perspective of our approach to symbolic execution, it is possible to disregard the initial Hadamard gates and the final QFT block and it suffices to look at circuits that match the template in Fig. 6.

5 Design and Implementation

Our exposition of the design and implementation of our system will follow the advice of Parnas and Clements [33] on *faking it*: a reconstruction of the requirements as we should have had them if we'd been all-knowing, and a design that fits those requirements. The version history in our GitHub repository can be inspected for anyone who wants to see our actual path.

As we experimented with the idea of partial evaluation and symbolic execution of quantum circuits, we ended up writing a lot of variants of essentially the same code, but with minor differences in representation. From these early experiments, we could see the major variation points:

- representation of *boolean values* and *boolean functions*,
- representation of ANF, and
- representation of circuits.

We also wanted to write out circuits only once, and have them be valid across these representation changes and be executable forwards, backwards, and in mixed retrodictive mode.

With the notable exception of Shor's algorithm, all the algorithms we study are expressed in the "black-box model." In that model, the circuit implementing the U_f is collapsed to just one function call. Of course, in any actual use of the algorithm, this circuit must be implemented and its execution time must be accounted for. Therefore, we want to—offline—synthesize a circuit from the boolean specification, i.e., for $f : [2^n] \to [2^m]$, we wish to generate the circuit for $g : [2^{n+m}] \to [2^{n+m}]$ such that for $x \in [2^n]$ and $y \in [2^m]$, we have $g(x, y) = (x, f(x) \oplus y)$.

This leads us to the following requirements that our code must fulfill.

5.1 Requirements

We need to be able to deal with the following variabilities:

1. multiple representations of *boolean values*,
2. multiple representations of *boolean formulae*,
3. different evaluation means (directly, symbolically, forwards, backwards, retrodictive).

It must also be possible to implement the following:

4. a reusable representation of circuits composed of generalized Toffoli gates,
5. a reusable representation of the inputs, outputs and ancillae associated to a circuit,
6. a *synthesis* algorithm for circuits implementing a certain boolean function,
7. a reusable library of circuits (such as Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor).

From those, we can make a set of design choices that drive the eventual solution.

We eventually want some non-functional characteristics to hold:

8. evaluation of reasonably-sized circuits should be relatively efficient.

5.2 Design

To meet the first requirement, we use *finally tagless* [14] to encode a *language of values*:

```
class (Show v, Enum v) => Value v where
zero :: v
```

Module	Service
Value	representation of a *language of values* (as a typeclass) and some constructors
VarInFormula	abstract representation of variables in formulae
Variable	variables as locations holding values and their constructors
ModularArith	modular arithmetic utilities useful in implementing certain algorithms, like Shor's
BoolUtils	function to interpret a list of booleans as an Integer
GToffoli	representation of generalized Toffoli gates and some constructors
Circuits	representation of circuits (sequences of gates) and of the special "wires" of our circuits
Synthesis	synthesis algorithm for circuits with particular properties
ArithCirc	creation of arithmetic circuits
EvalZ	evaluation of circuits on concrete values
FormAsList	representation of formulae as xor-lists of and-lists of literals-as-strings
FormAsMaps	representation of formulae as xor-maps of and-maps of literals-as-Int
FormAsBitmaps	representation of formulae as xor-maps of bitmaps
SymbEval	Symbolic evaluation of circuits
SymbEvalSpecialized	Symbolic evaluation of circuits specialized to the representation from FormAsBitmaps
QAlgos	generating the circuits themselves
RunQAlgos	running the actual circuits
Trace	utilities for tracing and debugging

Figure 7. Modules and their services

```
one  :: v
snot :: v -> v
sand :: v -> v -> v
sxor :: v -> v -> v

-- has a default implementation
snand :: [v] -> v -- n-ary and
snand = foldr sand one
```

which is then implemented 4 times, once for Bool and then multiple times for different symbolic variations. As a side-effect, this gives us requirement 3 "for free" if we can write a sufficiently polymorphic evaluator (which we will present below).

Unlike value representations that can be computed from context, we want to explicitly choose how to represent variables in formulae ourselves (part of requirement 2). Thus we use an explicit record instead of an implicit dictionary:

```
data VarInFormula f v = FR
  { fromVar  :: v -> f
  , fromVars :: Int -> v -> [ f ]
  }
```

Each formula representation may have different *variable representation* v and how to insert them into the current *formula representation* f, singly or as *n* formulae.

A generalized Toffoli gate can be represented by a list of value accessors br (short for boolean representation) along with a list of *controls* that tell us whether to use the bit directly or negated, along with which value will potentially be flipped. The implementation of very common gates (negation and controlled not) are also shown.

```
data GToffoli br = GToffoli [Bool] [br] br

xop :: br -> GToffoli br
xop = GToffoli [] []

cx :: br -> br -> GToffoli br
cx a = GToffoli [True] [a]
```

The core of a circuit (requirement 4) is then implemented as a sequence of these (where Seq is from Data.Sequence).

```
type OP br = Seq (GToffoli br)
```

Mainly for efficiency reasons, we model circuits as manipulating *locations holding values* rather than directly acting on values. We use STRefs (aliased to Var) for that purpose. Putting this together with the circuit template of 4, we get

```
data Circuit s v = Circuit
  { op         :: OP (Var s v)
  , xs         :: [Var s v]
  , ancillaIns  :: [Var s v]
  , ancillaOuts :: [Var s v]
  , ancillaVals :: [v]
  }
```

which lets us achieve requirement 5.

For requirement 6, we implement a straightforward version of a well-established algorithm [36]. Our implementation is *language agnostic*, in other words it works via the Value interface, so that the resulting circuits are all of type OP br for a free representation br. As circuit synthesis is only done for generating examples, we are not worried about its efficiency.

The arithmetic circuit generators are also based on text-book algorithms, and are not optimized in any way, neither for running time nor for gate count. Neither are the code for the quantum algorithms. They are, however, representation polymorphic.

Above, we said we had 3 different symbolic evaluators. These were not driven by having different levels of *precision* but rather by requirement 8, efficiency. Our first evaluator (`FormAsList`) uses xor-lists of and-lists of literals (as strings, i.e., "x0", "x1", ... in lexicographical order of the wires). ANF is then easy: and-lists are sorted, and duplicates removed. Xor-lists are sorted, grouped, even length lists are removed, and then made unique. This is woefully inefficient, and was the clear bottleneck in our profiles.

A less naïve approach uses a set of bits for representing literals, an `IntSet` for and-lists, and a normalized multiset for xor maps (a normalized multiset is one with only 0 and 1 as multiplicities). We found it more efficient to use a multiset for intermediate computations with xor maps which is normalized at the end instead of trying to track even/odd number of occurrences. Only computing Cartesian products in this representation requires some thought for finding a reasonably efficient algorithm.

While significantly faster, this representation still did not make our programs sufficiently efficient. Our final representation uses natural numbers as and-maps where the encoding of literals is now positional, and xor maps are again multisets of these "bitmaps."

As a last optimization, our circuits have a very particular property: the control wires are not written to, so that they are all literals. We use this further optimize the evaluation of single gates, where we eagerly and directly compute the ANF rather than waiting for later demand.

5.3 Implementation

The final code consists of 18 modules that implement various services, see Fig. 7 for a full listing. It consists of only 1449 lines of Haskell text, of which 646 lines are blank, import or comments, module declaration, so that 809 are "code." Testing and printing utilities are not counted in the above.

The code that occupies the most volume is that for running the examples, as each circuit needs its own setup for the input and output wires. Next is the implementation of symbolic representations of formulae in ANF. This is largely because there are a lot of pieces that need to be defined, including many instances; the algorithmic aspect rarely span more than 15 lines in total. The code for generating arithmetic circuits is voluminous as well as largely computational, but is a re-implementation of known material, as is the synthesis code.

A few comments on further implementation details. Sharp readers might have noticed `snand` as defined in class `Value` instead of as a polymorphic function outside the class; we do this to enable its implementation to be overridden. Lastly,

`GToffoli`'s implementation relies on an unexpressed invariant: that its two lists are of equal length. We really ought to refactor the code to use a single list of tuples, but this is a pervasive change that would not bring much benefit as we use combinators to build circuits, and these already maintain that invariant. Similarly for `Circuit`: the lists `ancillaIns`, `ancillaOut` and `ancillaVals` should all be of the same length. That invariant is not checked in our code.

6 Evaluation

We want to evaluate the *effectiveness* of our evaluator by running it on standard algorithms, as well as its (relative) *efficiency*.

We first give interesting aspects of running the six quantum algorithms outlined in Sec. 4, before commenting on complexity and workflow.

6.1 Symbolic Execution of the Algorithms

Most of the algorithms end up generating differently shaped constraint systems, and thus each need to be examined on its own. It is worth noting that all these algorithms are not known to have fast classical versions except for a few special cases [1, 13]. We spend more time on the analysis of Shor's algorithm, as it is both more important and displays subtle behavior.

Deutsch and Deutsch-Jozsa. We perform a retrodictive execution of the U_f block with an output measurement 0, i.e., with the state $|x_{n-1} \cdots x_1 x_0 0\rangle$. The result of the execution is a symbolic formula r that determines the conditions under which $f(x_0, \cdots, x_{n-1}) = 0$. When the function is constant, the results are $0 = 0$ (always) or $1 = 0$ (never) *regardless of how large the circuit is*. When the function is balanced, we get a formula that mentions the relevant variables. As examples, we generated all 12872 functions $[2^6] \rightarrow [2]$ that are valid inputs to the algorithm; the 2 constant functions and the 12870 balanced functions. The result of the symbolic execution immediately provides the answer but with the understanding that the generation of the formulae takes time depending on the distribution of zeros and ones. For example, here are the results of three executions for balanced functions $[2^6] \rightarrow [2]$:

- $x_0 = 0$,
- $x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 = 0$, and
- $1 \oplus x_3 x_5 \oplus x_2 x_4 \oplus x_1 x_5 \oplus x_0 x_3 \oplus x_0 x_2 \oplus x_3 x_4 x_5 \oplus x_2 x_3 x_5 \oplus$ $x_1 x_3 x_5 \oplus x_0 x_3 x_5 \oplus x_0 x_1 x_4 \oplus x_0 x_1 x_2 \oplus x_2 x_3 x_4 x_5 \oplus x_1 x_3 x_4 x_5 \oplus$ $x_1 x_2 x_4 x_5 \oplus x_1 x_2 x_3 x_5 \oplus x_0 x_3 x_4 x_5 \oplus x_0 x_2 x_4 x_5 \oplus x_0 x_2 x_3 x_5 \oplus$ $x_0 x_1 x_4 x_5 \oplus x_0 x_1 x_3 x_5 \oplus x_0 x_1 x_3 x_4 \oplus x_0 x_1 x_2 x_4 \oplus x_0 x_1 x_2 x_4 x_5 \oplus$ $x_0 x_1 x_2 x_3 x_5 \oplus x_0 x_1 x_2 x_3 x_4 = 0$.

In the first case, the function is balanced because it produces 0 exactly when $x_0 = 0$ which happens half of the time in all possible inputs; in the second case the output of the function is the exclusive-or of all the input variables which is another easy instance of a balanced function. The last case is

a cryptographically strong balanced function whose output pattern is balanced but, by design, difficult to discern [12].

Insight. In these algorithms, we actually do not care about the exact formula. Indeed, since we are *promised* that the function is either constant or balanced, then any formula that refers to at least one variable must indicate a balanced function: the outcome of the algorithm can be immediately decided if the formula is anything other than 0 or 1. Since the symbolic evaluation executes the U_f block "once," *one might conclude that it is a "de-quantization" of the algorithm in the black-box model, producing immediate answers even in cases when the quantum algorithm generates complicated entangled patterns during quantum evolution* [1]. However, it is important to remember that our circuits are "white-box" rather than "black-box" and that the time taken to execute the circuit is part of the overall complexity of the algorithm. We defer to Sec. 6.3 for a more detailed discussion of this point and refer to Komargodski et al. [29] for another perspective on black-box vs. white-box complexity in the context of graph algorithms.

Significance. That the details of the equations do not matter is crucial as the satisfiability of generally boolean equation is, in general, an *NP*-complete problem [17, 27, 38]. More directly, the answer to the algorithm does *not* require an exact calculation of the pre-image of the boolean function. Indeed, based on the conjectured existence of one-way functions which itself implies $P \neq NP$, pre-images calculations are believed to be computationally intractable in their most general setting. What is intriguing is that quantum algorithms appear to be able to answer certain general queries about pre-images without explicitly calculating the pre-image.

Bernstein-Vazirani. We show a complete small example. Let $n = 8$ and let the secret string be $s = 00111010$. In this case, the problem becomes: given a circuit for $f(x) = x_1 \oplus x_3 \oplus x_4 \oplus x_5$, determine the secret string. There are naturally many circuits that realize the function f; in our "white-box" model the running time of the symbolic execution will depend on which circuit is given. In the example below, we generate the simplest circuit: as all equivalent circuit have the same ANF representation, any other circuit would give the same symbolic output but potentially taking longer to execute:

```
retroBernsteinVazirani fr = print $ runST $ do
  xs <- newVars (fromVars fr 8 "x")
  y <- newVar zero
  let op = fromList [ cx (xs !! 1) y
                    , cx (xs !! 3) y
                    , cx (xs !! 4) y
                    , cx (xs !! 5) y
                    ]
  run Circuit { op = op
              , xs = xs
              , ancillaIns = [y]
```

```
              , ancillaOuts = [y]
              , ancillaVals = undefined
              }
  readSTRef y

runRetroBernsteinVazirani :: IO ()
runRetroBernsteinVazirani =
  retroBernsteinVazirani FL.formRepr

-- > runRetroBernsteinVazirani
-- x_1 \oplus x_3 \oplus x_4 \oplus x_5
```

As can be seen, `retroBernsteinVazirani` is parameterized by a representation of formulae: it allocates 8 symbolic variables, initializes the output to `zero`, generates the circuit, and runs it. The top level call `runRetroBernsteinVazirani` just needs to pick a particular representation of formulae. The result is the ANF representation of the circuit, from which the secret string can be read off: the indices $\{1, 3, 4, 5\}$ are the indices at which the secret string is 1.

Insight. Generally, the formulae are guaranteed to be of the form $\sum_i x_i$ for some indices i; the secret string is then the binary number that has a 1 at those indices.

Simon. The circuit below implements the black box for a function f such that $f(0) = f(3) = 0$ and $f(1) = f(2) = 3$:
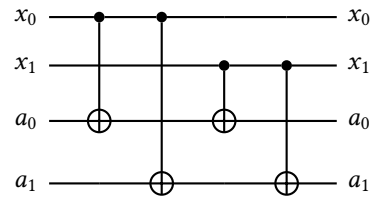
We have for all $x \in [2^2]$ that $f(x) = f(x \oplus 3)$ where $\oplus$ is applied bitwise, i.e., the secret value $a = 3$. To extract this a via symbolic execution we proceed as follows. We first pick a random x, say $x = 3$, fix the initial condition $a = 0$ and run the circuit forward. This execution produces, in the output register, the value of $f(x) = 0$. We now run a symbolic retrodictive execution with $a = 0$ at the output site. That execution produces information on all values of a that are consistent with the observed result. In this case, we get: $a_0 = x_0 \oplus x_1$ and $a_1 = x_0 \oplus x_1$. Reconciling these equations with the initial conditions $a_0 = a_1 = 0$, we conclude $x_0 = x_1$. In other words, any input $x_1 x_0$ such that $x_0 = x_1$ is consistent with the observed result. There are two such inputs $x = 0$ or $x = 3$ and the secret a is their difference.

Insight. Simon's problem does not seem to have a resolution that is easy to read from the resulting equations. Generally, we get equations that have exactly two solutions, one of which is known. In some cases, like above, it is straightforward to infer the second solution, but as the equations get more and more complex, the problem becomes harder.

$u = 0$ $1 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_2x_3$
$\oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 1$ $x_0 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 2$ $x_1 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 3$ $x_0x_1 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 4$ $x_2 \oplus x_2x_3 \oplus x_1x_2 \oplus x_0x_2 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 5$ $x_0x_2 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 6$ $x_1x_2 \oplus x_1x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 7$ $x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 8$ $x_3 \oplus x_2x_3 \oplus x_1x_3 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 9$ $x_0x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 10$ $x_1x_3 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 11$ $x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 12$ $x_2x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2x_3$

$u = 13$ $x_0x_2x_3 \oplus x_0x_1x_2x_3$

$u = 14$ $x_1x_2x_3 \oplus x_0x_1x_2x_3$

$u = 15$ $x_0x_1x_2x_3$

Figure 8. Result of retrodictive execution for the Grover oracle ($n = 4$, w in the range $\{0..15\}$). The highlighted red subformula is the binary representation of the hidden input u

Grover. A reversible oracle circuit for a function $f(x)$ that returns 1 for a unique input u and 0 otherwise is rather trivial: it consists of a single generalized Toffoli gate that flips the output when the input matches u:

```
synthesisGrover ::
  Int -> [Var s v] -> Integer -> OP s v
synthesisGrover n (viewL -> (xs,y)) u =
  S.singleton $ GToffoli (fromInt n u) xs y
```

The ANF representation of the circuit is sensitive to the value u as shown in Fig. 8 and the running time of symbolic execution varies accordingly. What is interesting is that the shortest subformula in the ANF representation is guaranteed to be the binary representation of u. As an example, if $u = 3$, then x_1x_0 must be a subformula in the ANF representation. By itself, this subformula would satisfy not just 3 but also 7, 11, and 15. To exclude this latter values, the ANF representation includes the clause $x_2x_1x_0$ to exclude 7, the clause $x_3x_1x_0$ to exclude 11, and the clause $x_3x_2x_1x_0$ to exclude 15.

Insight. For Grover as well, the result can be immediately read off the formula.

Shor. The circuit in Fig. 9 uses a hand-optimized implementation of quantum oracle U_f for the modular exponentiation function $f(x) = 4^x \mod 15$ to factor 15 using Shor's algorithm. In a conventional forward execution, the state before the QFT block is:

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

At this point, the output register is measured to be either $|1\rangle$ or $|4\rangle$. In either case, the input register snaps to a state of the
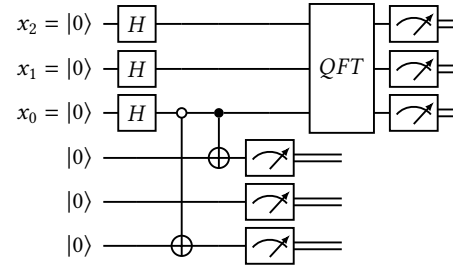

Figure 9. Hand optimized quantum circuit for finding the period of $4^x \mod 15$

form $\sum_{r=0}^{3} |a + 2r\rangle$ whose QFT has peaks at $|0\rangle$ or $|4\rangle$ making them the most likely outcomes of measurements of the input register. If we measure $|0\rangle$, we repeat the experiment; otherwise we infer that the period is 2.

In the backwards execution, we can start with the state $|x_2x_1x_0001\rangle$ since 1 is guaranteed to be a possible output measurement (corresponding to $f(0)$). The first cx-gate changes the state to $|x_2x_1x_0x_001\rangle$ and the second cx-gate produces $|x_2x_1x_0x_00x_0\rangle$. At that point, we reconcile the retrodictive result of the output register $|x_00x_0\rangle$ with the initial condition $|000\rangle$ to conclude that $x_0 = 0$. In other words, in order to observe the output at 001, the input register must be initialized to a superposition of the form $|??0\rangle$ where the least significant bit must be 0 and the other two bits are unconstrained. Expanding the possibilities, the input register needs to be in a superposition of the states $|000\rangle$, $|010\rangle$, $|100\rangle$ or $|110\rangle$ and we have just inferred using purely classical but retrodictive reasoning that the period is 2.

Base		Equations				Solution
$a = 11$	$x_0 = 0$					$x_0 = 0$
$a = 4, 14$	$1 \oplus x_0 = 1$	$x_0 = 0$				$x_0 = 0$
$a = 7, 13$	$1 \oplus x_1 \oplus x_0 x_1 = 1$	$x_0 x_1 = 0$	$x_0 \oplus x_1 \oplus x_0 x_1 = 0$	$x_0 \oplus x_0 x_1 = 0$		$x_0 = x_1 = 0$
$a = 2, 8$	$1 \oplus x_0 \oplus x_1 \oplus x_0 x_1 = 1$	$x_0 x_1 = 0$	$x_1 \oplus x_0 x_1 = 0$		$x_0 \oplus x_0 x_1 = 0$	$x_0 = x_1 = 0$

Figure 10. Equations generated by retrodictive execution of $a^x \mod 15$ for different values of a, starting from observed result 1 and unknown $x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$. The solution for the unknown variables is given in the last column
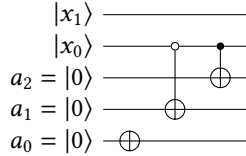

Figure 11. Quantum circuit for finding the period of 4^x mod 21 using qutrits. The three gates are from left to right are the X, SUM, and $C(X)$ gates for ternary arithmetic [9]. The X gate adds 1 modulo 3; the controlled version $C(X)$ only increments when the control is equal to 2, and the SUM gates maps $|a, b\rangle$ to $|a, a + b\rangle$

This result does not, in fact, require the small optimized circuit of Fig. 9. In our implementation, modular exponentiation circuits are constructed from first principles using adders and multipliers [40]. In the case of $f(x) = 4^x \mod 15$, although the unoptimized constructed circuit has 56,538 generalized Toffoli gates, the execution results in just two simple equations: $x_0 = 0$ and $1 \oplus x_0 = 1$. Furthermore, as shown in Fig. 10, the shape and size of the equations is largely insensitive to the choice of 4 as the base of the exponent, leading in all cases to the immediate conclusion that the period is either 2 or 4. When the solution is $x_0 = 0$, the period is 2, and when it is $x_0 = x_1 = 0$, the period is 4.

The remarkable effectiveness of retrodictive computation of the Shor instance for factoring 15 is due to a coincidence: a period that is a power of 2 is clearly trivial to represent in the binary number system which, after all is expressly designed for that purpose. That coincidence repeats itself when factoring products of the (known) Fermat primes: 3, 5, 17, 257, and 65537, and leads to small circuits [22]. This is confirmed with our implementation which smoothly deals with unoptimized circuits for factoring such products. Factoring 3*17=51 using the unoptimized circuit of 177,450 generalized Toffoli gates produces just the 4 equations: $1 \oplus x_1 = 1$, $x_0 = 0$, $x_0 \oplus x_0 x_1 = 0$, and $x_1 \oplus x_0 x_1 = 0$. Even for 3*65537=196611 whose circuit has 4,328,778 generalized Toffoli gates, the execution produces 16 small equations that refer to just the four variables x_0, x_1, x_2, and x_3 constraining them to be all 0, i.e., asserting that the period is 16.

Since periods that are powers of 2 are rare and special, we turn our attention to factoring problems with other periods.

The simplest such problem is that of factoring 21 with an underlying function $f(x) = 4^x \mod 21$ of period 3. The unoptimized circuit constructed from the first principles has 78,600 generalized Toffoli gates; its execution generates just three equations. But even in this rather trivial situation, the equations span 5 pages of text! A small optimization reducing the number of qubits results in a circuit of 15,624 generalized Toffoli gates whose execution produces still quite large, but more reasonable, equations. To understand the reason for these unwieldy equations, we examine a general ANF formula of the form $X_1 \oplus X_2 \oplus X_3 \oplus \ldots = 0$ where each X_i is a conjunction of some boolean variables, i.e., the variables in each X exhibit constructive interference as they must all be true to enable that $X = 1$. Since the entire formula must equal to 0, every $X_i = 1$ must be offset by another $X_j = 1$, thus exhibiting negative interference among X_i and X_j. Generally speaking, arbitrary interference patterns can be encoded in the formulae at the cost of making the size of the formulae exponential in the number of variables. This exponential blowup is actually a necessary condition for any quantum algorithm that can offer an exponential speed-up over classical computation [26].

It would however be incorrect to conclude that factoring 21 is inherently harder than factoring 15. The issue is simply that the binary number system is well-tuned to expressing patterns over powers of 2 but a very poor match for expressing patterns over powers of 3. Indeed, we show that by just using qutrits, the circuit and equations for factoring 21 become trivial while those for factoring 15 become unwieldy. The manually optimized circuit in Fig. 11 consists of just three gates; its retrodictive execution produces two equations: $x_0 = 0$ and $x_0 \neq 2$, setting $x_0 = 0$ and leaving x_1 unconstrained. The matching values in the qutrit system are 00, 10, 20 or in decimal 0, 3, 6 clearly identifying the period to be 3.

6.2 Time Measurements

We show a few representative set of timings for the Deutsch-Jozsa and Grover problems.

Fig. 12 shows what happens when we vary the size of the problem for Deutsch-Josza on three different balanced functions of n variables $x_0, \cdots, x_{n-1}$: one that returns x_0, one that returns x_{n-1}, and one that returns $x_0 \oplus \cdots \oplus x_{n-1}$. All
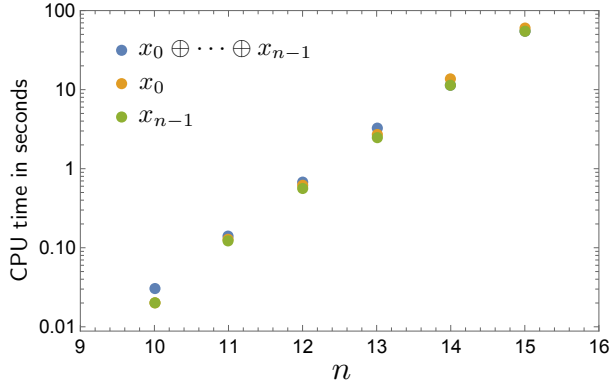
Figure 12. Execution times for the retrodictive execution of the Deutsch-Jozsa algorithm on 3 balanced functions at different sizes
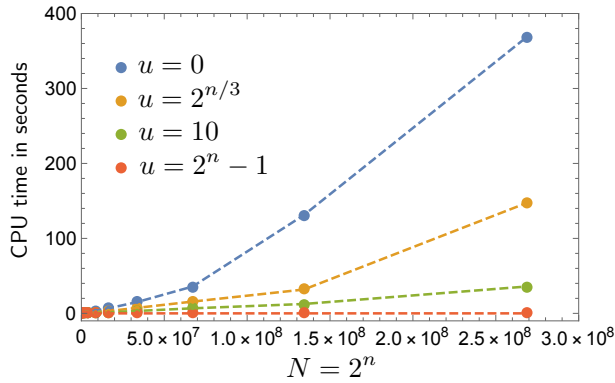

Figure 13. Execution times for the retrodictive execution of the Grover algorithm on different "secret" values u at different sizes
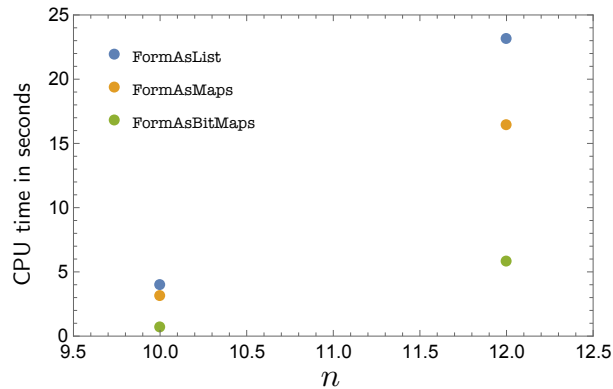

Figure 14. Execution times for the retrodictive execution of the Grover algorithm on secret value $u = 0$ at two sizes but using different ANF representations

three examples show clear exponential behavior that does not depend on the size of the final answer. We show the timings only for a single implementation of the representation of formulae as, somewhat mysteriously, this makes no difference in this case, we get essentially the same result in all cases.

Fig. 13 shows timings of different calls to Grover's algorithm, using the best representation for formulae (as bitmaps) but varying what "secret value u" we are looking for. Here the binary representation of u matters greatly, ranging from what looks like essentially constant time to exponential time.

Lastly, Fig. 14 shows Grover's algorithm again, using the worst case value from the previous figure ($u = 0$) but varying the representation at two sizes. Here we can clearly see the very strong effect that this has on the timings. This is not just a constant improvement, it is also a complexity improvement.

6.3 Complexity

In the general case, we have a circuit containing T generalized Toffoli gates over $n + m$ qubits split in two registers A (n qubits) and B (m qubits). The typical symbolic execution takes the following steps with the given worst-case complexity:

1. If the quantum algorithm is expressed in terms of calls to a black-box oracle (all the problems we consider except Shor), then the first step is to design the oracle *efficiently*. Perhaps surprisingly, it turns out we don't have to be particularly clever in designing that circuit: textbook designs with million of gates can work well.
2. Let $A = |00\dots0\rangle$ and $B = |00\dots0\rangle$ and run the circuit with classical inputs. This has complexity $O(T)$ as it takes T steps where each step takes constant time. The result of this evaluation will leave A intact and produce some value b for the B register.
3. We now run the circuit backwards with the symbolic values $A = |x_{n-1}\dots x_1 x_0\rangle$ and $B = |b\rangle$. This takes T steps. At each step, we have m ANF equations over the $\{x_0, x_1, \dots, x_{n-1}\}$ variables. The size of each equation might be $O(2^n)$ in the worst case. So the overall complexity of this step is $O(Tm2^n)$.
4. The answer to the algorithm is obtained by either inspecting or, in the worst case, solving the resulting m equations. In the Deutsch-Jozsa and Grover algorithms, the solution is immediate by inspection of the equations.

There are two potential bottlenecks: steps (3) above which has a worst-case complexity of $O(Tm2^n)$, and step (4) in the solve case, which is an *NP*-complete problem. The $O(T)$ factor is inevitable because we have a white box implementation of the oracle and we must touch every gate in that implementation. The $O(m)$ factor is also inevitable as it represents the number of variables. What varies from one function to

the other and, for a particular function, from one oracle implementation to the other is the $O(2^n)$ factor.

What our examples show is that while the worst-case for some algorithms is $O(2^n)$, it seems that the expected case actually depends on the length of the encoding (in binary) of the information contained in the answer, especially in the case where we do not need to solve the constraints.

6.4 Workflow

While we would like to be able to offer a uniform workflow, our case studies do not seem to reveal one: how to "read" the resulting system of equations to obtain an answer seems very algorithm-dependent. The fact that all the quantum algorithms uniformly use the QFT (or its Hadamard approximation) to extract the relevant properties of interest reconfirms the crucial but mysterious role of the QFT in quantum computing [11, 21, 39, 43].

7 Conclusion

Symbolic execution is a way of evaluating a given program abstractly, so that the abstraction represents multiple inputs sharing an evolution path through the program, with solutions encoded in equations or constraints. So far, this way of execution has been limited to the classical realm. In this work, we extended these ideas to the quantum realm by considering the computational quantum universality of Hadamard and Toffoli gates. The proposed replacement of $H|0\rangle$ by $|z\rangle$, where z is a symbol, provides the key to capturing some of the entanglement (non-local correlations) present in those programs; however, the execution is classical. Surprisingly, in many well-known quantum algorithms (such as Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover) these correlations are sufficient to obtain the solution efficiently for some inputs with a plain *classical symbolic execution* as opposed to a purely quantum execution (involving states that belong to a complex vector space endowed with an inner product). This raises many questions, in particular, foundational ones regarding the origin of the power of quantum computation.

References

[1] Alastair A. Abbott. 2012. The Deutsch-Jozsa problem: de-quantization and entanglement. *Natural Computing* 11 (2012).

[2] D. Aharonov. 2003. A simple proof that Toffoli and Hadamard are qauntum universal. *arXiv:quant-ph/0301040* (2003).

[3] Yakir Aharonov and Lev Vaidman. 2008. *The Two-State Vector Formalism: An Updated Review*. Springer Berlin Heidelberg, Berlin, Heidelberg, 399–447. https://doi.org/10.1007/978-3-540-73473-4_13

[4] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). 1–21. https://doi.org/10.4204/EPTCS.287.1

[5] Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *Computer Aided*

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[7] Stephen M. Barnett, John Jeffers, and David T. Pegg. 2021. Quantum Retrodiction: Foundations and Controversies. *Symmetry* 13, 4 (2021). https://doi.org/10.3390/sym13040586

[8] Ethan Bernstein and Umesh Vazirani. 1997. Quantum Complexity Theory. *SIAM J. Comput.* 26, 5 (1997), 1411–1473. https://doi.org/10.1137/S0097539796300921 arXiv:https://doi.org/10.1137/S0097539796300921

[9] Alex Bocharov, Shawn X. Cui, Martin Roetteler, and Krysta M. Svore. 2016. Improved Quantum Ternary Arithmetic. *Quantum Info. Comput.* 16, 9–10 (jul 2016), 862–884.

[10] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.* 10, 6 (apr 1975), 234–245. https://doi.org/10.1145/390016.808445

[11] Daniel E Browne. 2007. Efficient classical simulation of the quantum Fourier transform. *New Journal of Physics* 9, 5 (may 2007), 146. https://doi.org/10.1088/1367-2630/9/5/146

[12] Linda Burnett, William Millan, Edward Dawson, and Andrew Clark. 2004. Simpler Methods for Generating Better Boolean Functions with Good Cryptographic Properties. *Australasian Journal of Combinatorics* 29 (2004), 231–247. https://eprints.qut.edu.au/21763/

[13] Cristian S. Calude. 2007. De-quantizing the solution of Deutsch's problem. *International Journal of Quantum Information* 5, 3 (2007), 409–415.

[14] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[15] Jacques Carette, Gerardo Ortiz, and Amr Sabry. 2022. Retrodictive Quantum Computing. *arXiv:2205.06346* (2022).

[16] Lori A. Clarke. 1976. A Program Testing System. In *Proceedings of the 1976 Annual Conference* (Houston, Texas, USA) *(ACM '76)*. Association for Computing Machinery, New York, NY, USA, 488–491. https://doi.org/10.1145/800191.805647

[17] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (Shaker Heights, Ohio, USA) *(STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/800157.805047

[18] D. Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring. (2002). arXiv:quant-ph/0201067.

[19] David Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A 400* (1985).

[20] David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A 439* (1992).

[21] Johann Makowsky Dorit Aharonov, Zeph Landau. 2006. The quantum FFT can be classically simulated. (2006). arXiv:quant-ph/0611156.

[22] Michael R. Geller and Zhongyuan Zhou. 2013. Factoring 51 and 85 with 8 qubits. *Scientific Reports (3023)* 3, 1 (2013).

[23] D Gottesman. 1998. The Heisenberg representation of quantum computers. (6 1998). https://www.osti.gov/biblio/319738

[24] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[25] William E. Howden. 1976. Experiments with a symbolic evaluation system. In *Proceedings of the National Computer Conference*.

Verification, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–21.

[26] Richard Jozsa and Noah Linden. 2003. On the Role of Entanglement in Quantum-Computational Speed-Up. *Proceedings: Mathematical, Physical and Engineering Sciences* 459, 2036 (2003), 2011–2032. http://www.jstor.org/stable/3560059

[27] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[28] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252

[29] Ilan Komargodski, Moni Naor, and Eylon Yogev. 2019. White-Box vs. Black-Box Complexity of Search Problems: Ramsey and Graph Property Testing. *J. ACM* 66, 5, Article 34 (jul 2019), 28 pages. https://doi.org/10.1145/3341106

[30] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified Compilation of Quantum Oracles. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 146 (oct 2022), 27 pages. https://doi.org/10.1145/3563309

[31] Torben Ægidius Mogensen. 2011. Partial Evaluation of the Reversible Language Janus. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA) *(PEPM '11)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/1929501.1929506

[32] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

[33] David Lorge Parnas and Paul C Clements. 1986. A rational design process: How and why to fake it. *IEEE transactions on software engineering* 2 (1986), 251–257.

[34] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509. https://doi.org/10.1137/S0097539795293172 arXiv:https://doi.org/10.1137/S0097539795293172

[35] D.R. Simon. 1994. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 116–123. https://doi.org/10.1109/SFCS.1994.365701

[36] Mathias Soeken, Gerhard W Dueck, and D Michael Miller. 2016. A fast symbolic transformation based algorithm for reversible logic synthesis. In *International Conference on Reversible Computation*. Springer, 307–321.

[37] Natalia Tokareva. 2015. Chapter 1 - Boolean Functions. In *Bent Functions*, Natalia Tokareva (Ed.). Academic Press, Boston, 1–15. https://doi.org/10.1016/B978-0-12-802318-1.00001-7

[38] B.A. Trakhtenbrot. 1984. A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms. *Annals of the History of Computing* 6, 4 (1984), 384–400. https://doi.org/10.1109/MAHC.1984.10036

[39] Maarten Van Den Nest. 2013. Efficient Classical Simulations of Quantum Fourier Transforms and Normalizer Circuits over Abelian Groups. *Quantum Info. Comput.* 13, 11–12 (nov 2013), 1007–1037.

[40] Vlatko Vedral, Adriano Barenco, and Artur Ekert. 1996. Quantum networks for elementary arithmetic operations. *Phys. Rev. A* 54 (Jul 1996), 147–153. Issue 1. https://doi.org/10.1103/PhysRevA.54.147

[41] Satosi Watanabe. 1955. Symmetry of Physical Laws. Part III. Prediction and Retrodiction. *Rev. Mod. Phys.* 27 (Apr 1955), 179–186. Issue 2. https://doi.org/10.1103/RevModPhys.27.179

[42] Ingo Wegener. 1987. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., USA.

[43] Nadav Yoran and Anthony J. Short. 2007. Efficient classical simulation of the approximate quantum Fourier transform. *Phys. Rev. A* 76 (Oct 2007), 042321. Issue 4. https://doi.org/10.1103/PhysRevA.76.042321