Compilation to Quantum Circuits for a Language with Quantum Data and Control

Yannis Rouselakis*† Nikolaos S. Papaspyrou* Yiannis Tsiouris* Eneia N. Todoran‡

*School of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens, Greece
Email: {nickie, gtsiour}.softlab.ntua.gr

†Department of Computer Science
University of Texas at Austin
2317 Speedway, Stop D9500, Austin, TX 78712, USA
Email: jrous@cs.utexas.edu

‡Computer Science Department
Technical University of Cluj-Napoca
Baritiu Street 28, 400027, Cluj-Napoca, Romania
Email: Eneia.Todoran@cs.utcluj.ro

***Abstract*—In this paper we further investigate nQML, a functional quantum programming language that follows the "quantum data and control" paradigm. We define a semantics for nQML, which translates programs to quantum circuits in the category FQC of finite quantum computations, following the approach of Altenkirch and Grattage's QML. This semantics, which coincides with the denotational semantics for nQML over density matrices and unitary transformations, serves as a compiler from nQML programs to quantum circuits. We also provide an implementation of this compiler, written in Haskell, as well as an interpreter for quantum circuits.**

I. Introduction

Quantum computing processes data that is stored in the form of quantum bits (qubits) and, for doing so, it employs quantum mechanical phenomena such as the superposition and entanglement of quantum states. Roughly speaking, a qubit may contain the digit "0", the digit "1", or any superposition of these two. Although research towards the manufacturing of quantum computers has not yet led to mature results, *quantum circuits* seem to be today a commonly accepted model for quantum hardware. Such circuits consist of appropriate formations of quantum gates, acting upon qubits in the same way that classical logic gates act upon bits in ordinary computers.

Most quantum programming languages that have been proposed so far are based on the principle "quantum data, classical control", that is, on the idea that the execution of a quantum program follows a specific control-flow, exactly as the execution of a program in a classical computer. Such languages allow programmers to use quantum data, in addition to classical, and through their manipulation to implement quantum algorithms. On a different track, we see languages following the "quantum data and control" paradigm. Such languages use quantum control flow; in other words, they allow the execution flow of a program to be in a superposition of various different states in exactly the same way as the quantum data that the program manipulates.

nQML [15], [14] is a high-level functional language based on the concept of "quantum data and control." It was defined by Lampis *et al.*, inspired by Altenkirch and Grattage's QML [1], [8], [9], and its main design goal was to give programmers sufficient expressive power to implement quantum algorithms easily, while preventing them from breaking the rules of quantum computation. nQML includes constructs which allow any unitary transformation to be expressed as a program in nQML quite naturally, more or less using the same notation that is used by the designers of quantum algorithms. It also permits quantum measurements to be carried out at any point during the execution of a program.

As explained in the paper defining nQML [14], the relative ease of use of the language comes at the cost of putting aside a number of important practical issues, such as the existence of imperfect quantum hardware, the need for quantum error correction and the fact that every quantum program will eventually have to be implemented as a quantum circuit using only a finite set of quantum gates and, therefore, some of the unitary transformations that nQML allows will have to be approximated. Similar problems were a source of concern for the founders of the classical programming model many decades ago. Fortunately they have been resolved and their solutions have been abstracted in such a way that people who use modern high-level programming languages need not know anything about them. The same can and must be done for quantum programming languages and, therefore, such issues should be tackled not by the designer and users of a quantum programming language, but by the architect of a quantum computer, the designer of its operating system and, to a lesser extent, the designer of the compiler.

In order to demonstrate the feasibility of using nQML as a quantum programming language and to draw attention to the assumptions that are necessary and to the problems that remain to be resolved, in this paper we define a compiler for nQML, targeting quantum circuits in the category FQC of finite quantum computations [1], [8]. We also provide an implementation of the compiler in Haskell, as well as an interpreter for quantum circuits in FQC. The combination of

these two can be used for the execution of quantum programs and for a direct comparison with the original definition of the language, using denotational semantics over density matrices and unitary transformations [14].

The rest of the paper is structured as follows. In section II we give the syntax of the language nQML and explain its constructs. Section III contains a description of the quantum circuits that we will use as the target language for our nQML compiler, which is in turn defined in section IV. Section V contains a number of examples in nQML, corresponding to well known quantum algorithms, and the quantum circuits in which they are compiled. We conclude with an exposition of related work, followed by some remarks and directions for future work.

Due to space limitations, in this paper we do not have the luxury to explain how quantum programming works. It is assumed that the reader is familiar with the basics of quantum computation and quantum circuits. There are several introductory books [3], [12], [16], as well as publicly available manuscripts and course material on this field.

II. The language nQML

The syntax of nQML is given by the following grammar. It is assumed that x is a variable identifier and λ is a complex constant. The grammar defines two syntactic classes. Quantum expressions are denoted by e; they represent quantum programs and their syntax is similar to that of QML. Classical expressions are denoted by c; they are only needed in the quantum transformation construct $|e\rangle \to x, x'.\, c$ and they can represent two types of information: a structure of classical bits or a complex number.

$$
\begin{array}{rcl}
e & ::= & x \mid \{\,(\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\,\} \\
& \mid & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
& \mid & (e_1, e_2) \mid \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \\
& \mid & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
& \mid & |e\rangle \to x, x'.\, c \\
c & ::= & x \mid \mathbf{false} \mid \mathbf{true} \mid \lambda \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \\
& \mid & (c_1, c_2) \mid \mathbf{let}\ (x_1, x_2) = c_1\ \mathbf{in}\ c_2 \\
& \mid & \mathbf{if}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid c_1 = c_2 \mid c_1 < c_2 \\
& \mid & \mathbf{int}\, c \mid c_1 + c_2 \mid c_1 - c_2 \mid c_1 * c_2 \mid c_1 / c_2 \mid c_1^{c_2}
\end{array}
$$

Variables in nQML are viewed as references to quantum information that is stored in a global quantum state. There are two types of quantum information: qubits and products. A new qubit is allocated in the quantum state when the superposition operator $\{\,(\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\,\}$ is used, in the same way that new objects are allocated on the heap when a data constructor is used in a functional programming language. Products are introduced and eliminated with the constructs (e_1, e_2) and $\mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2$. nQML also features three control constructs:

- $\mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$: It conducts a measurement on e, which must be of type qubit. Depending on the result, it executes one of its branches. It is similar to a classical random branching, based on a toss of a biased coin with probabilities depending on the state of the qubit being measured.
- $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$: It allows the programmer to perform quantum branching. If e, which must be of type qubit, is in a classical state, then the effect is what we would expect from **ifm**. But if e is in a quantum superposition, the program proceeds in a quantum superposition of both branches, most likely creating entanglement among the qubits of the quantum state.
- $|e\rangle \to x, x'.\, c$: A generic means of expressing any unitary transformation, which has to be relied upon when a transformation can not be easily broken down to a series of controlled operations, expressible with **if**. Its advantage is that, rather than forcing programmers to precompute and provide the whole unitary matrix of the transformation, whose size is exponential in the number of qubits that it affects, it allows them to express that matrix as a complex function of the input and output state of the transformed qubits. This leads to a succinct and clear expression of many useful quantum algorithms, such as the Deutsch-Jozsa or Grover's algorithm that are described in Section V.
  In quantum pseudocode notation, all unitary transformations can be expressed in the form:
  $$|i\rangle \to \sum_{j=0}^{2^n-1} f(i,j)\,|j\rangle$$
  where $f(i,j)$ is a function of the input state i of the quantum register and its output state j. The construct $|e\rangle \to x, x'.\, c$ allows the programmers to use precisely this natural notation: the classical variables x and x' denote the register's input and output state and the classical expression c denotes the function's body.
  From this notation, if the function f is known, the unitary matrix can be easily constructed by taking $S_{j,i} = f(i,j)$. Of course, not all functions f result in unitary matrices and the type system of nQML cannot efficiently decide whether the resulting transformation is indeed unitary. The type system of Altenkirch and Grattage's QML is able to do that, at the expense of making the size of the program exponential and complicating the typing with orthogonality constraints.

nQML admits a simple type system and denotational semantics [14]. By simple, we mean that both use structures and techniques that are typical in the study of classical programming languages of similar size and complexity.

The main novelty of nQML's type system is that the type of a quantum expression conveys information which reveals the exact qubits of the quantum state in which the expression's value resides. Qubit aliasing is allowed in such a way that the "no cloning" and "no dropping" principles are not violated. Programmers have the look-and-feel of a classical programming language, without linearity restrictions. The type system can be extended to support polymorphic higher-order functions, where polymorphism is over the exact qubits of the quantum state that are used for representing data [14].

Quantum types, in the type system of nQML, are defined by the grammar

$$\tau \ ::= \ \mathbf{qbit}[n] \ | \ \tau_1 \otimes \tau_2$$

where n is the exact qubit of the state that is used, e.g., an expression has type $\mathbf{qbit}[5]$ if its value is stored in the 5th qubit of the state. This information is used to make sure that the same qubit cannot be used twice in a transformation.

There are two typing relations: $\Gamma; n \ \vdash^{\circ} \ e \ : \ \tau; m$ is for type checking pure quantum expressions (i.e. without measurements); on the other hand, $\Gamma; n \ \vdash \ e \ : \ \tau; m$ is for type checking arbitrary quantum expressions, We refer to both by $\Gamma; n \ \vdash^{\alpha} \ e \ : \ \tau; m$, allowing the superscript $^{\alpha}$ to be either $^{\circ}$ or empty. As the types of nQML convey information regarding the position of qubits in the quantum state, the typing relation is forced to process and propagate such information. In $\Gamma; n \vdash^{\alpha} e : \tau; m$, the natural number n appearing on the left side of the relation stands for the number of qubits of the original quantum state, before e starts evaluating. The natural number m appearing on the right side of the relation stands for the number of new qubits that are allocated during the evaluation of e. The typing rules are defined in [14].

The denotational semantics of nQML is based on the use of density matrices to describe quantum states. The meaning of a well-typed nQML program is a function from density matrices to density matrices and describes the program's effect on an arbitrary quantum input state. Pure well-typed programs, i.e., programs which conduct no measurements, are also assigned a meaning in the form of a unitary matrix which describes the transformation they perform on the quantum state. The execution of an nQML program can be seen as a sequence of steps which affect the quantum state by allocating new qubits, by applying unitary transformations to existing qubits or by measuring existing qubits.

III. Quantum circuits

Quantum circuits are one possible model of quantum computation. We extend the Haskell data type proposed by Altenkirch and Grattage [6], [8] by adding one more constructor for arbitrary unitary matrices (`Unit`), which will be the target of nQML's $|e\rangle \to x, x'.\, c$ construct.

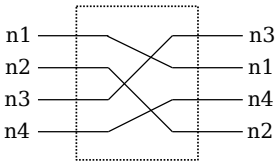
```
data Circ = Rot (C,C) (C,C)
          | Wire [Int]
          | Par Circ Circ
          | Seq Circ Circ
          | Cond Circ Circ
          | Unit (Matrix C)
```

The set of quantum circuits operating on a state of n qubits are defined inductively using these constructors:

- Rotation `Rot` $(\lambda_0, \lambda_1)\,(\kappa_0, \kappa_1)$: introduces a new unitary transformation on one qubit ($n = 1$), defined by the following matrix, where $\lambda_0^* \kappa_0 + \lambda_1 \kappa_1^* = 0$.

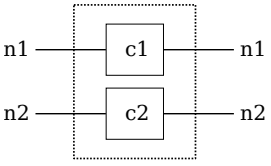
$$\begin{pmatrix} \lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1 \end{pmatrix}$$

- Wire reordering `Wire` p: reorders the qubits in the state. The parameter p must be a permutation of the sequence $[0\,..\,n-1]$. If the i-th element of this permutation is j, this means that the wire at the i-th position in the input state becomes the j-th wire of the output state. The identity permutation corresponds to the identity unitary matrix which leaves the state unchanged. When drawing quantum circuits, we will not use quantum gates to implement wire reordering; we will just draw crossing wires, e.g., as follows:

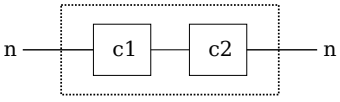


  In all circuits, the numbers next to the wires denote multiplicity, i.e., the number of qubits in the state. If $n_1 = n_2 = n_3 = n_4 = 1$, the reordering shown above would be encoded in Haskell by `Wire [1,3,0,2]`.

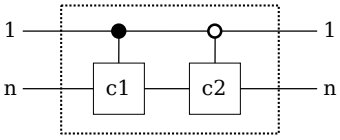
- Parallel composition `Par` $c_1\, c_2$: combines c_1 and c_2 in parallel, adding the number of qubits in their states.



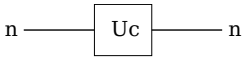
- Sequential composition `Seq` $c_1\, c_2$: combines c_1 and c_2 in sequence, The two circuits must have a state of n qubits, where n is also the number of qubits in their composition.



- Conditional `Cond` $c_1\, c_2$: creates a conditional circuit that is controlled by an extra qubit. The two circuits must have a state of n qubits, whereas the number of qubits in the conditional circuit is $n + 1$.



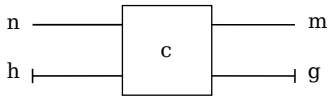
- Arbitrary unitary matrix `Unit` C: creates a circuit with a state of n qubits corresponding to the unitary matrix C. Such a circuit must in general be approximated by an appropriate composition of elementary quantum gates.



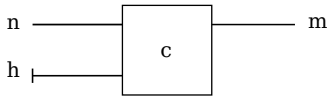
Reversible finite quantum circuits form the category $\mathsf{FQC}^{\approx}$, whose objects are states of n qubits and morphisms are unitary

transformations. Following Altenkirch and Grattage [6], [8], we define two larger categories, FQC and FQC$^\circ$.

Circuits in FQC are not necessarily reversible: they can also contain measurements and/or qubit initializations (which also amount to measurements). To model circuits in FQC, we separate a number of qubits of the input state, which we call *heap*, and a number of qubits of the output state, which we call *garbage*. Qubits in the heap are considered to be initialized to $|0\rangle$. Qubits in the garbage are measured and discarded. When drawing circuits, we denote the heap and garbage by terminating lines. It must be $n + h = m + g$.



Also, the category FQC$^\circ$ is a subset of FQC where circuits are allowed to have a heap, but not garbage. Such circuits are pure, in the sense that they do not contain measurements, and can be modelled by unitary transformations between pure quantum states. It must be $n + h = m$.

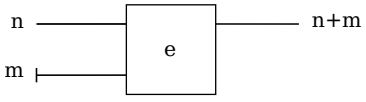


Obviously, FQC$^{\approx} \subset$ FQC$^\circ \subset$ FQC.

IV. A compiler for nQML

Following the compilation approach used by Altenkirch and Grattage [6], [8] we use the typing relation for compiling (pure and impure) quantum expressions. However, in contrast to the approach used for QML, the process is not guided by the linear type system, deciding how to split the wires of the input state. Instead, purity information and the numbers n and m from nQML's typing information are used.

If e is a pure quantum expression such that $\Gamma; n \vdash^\circ e : \tau; m$, then e is compiled to a circuit in FQC$^\circ$ which has an input state of n wires plus m wires of heap and an output state of $n + m$ wires (without garbage). We draw this as follows:



On the other hand, if e is an impure quantum expression such that $\Gamma; n \vdash e : \tau; m$, then e is compiled to a circuit in FQC which has an input state of n wires plus $h \geq m$ wires of heap and an output state of $n + m$ wires plus g wires of garbage. It must be $h = m + g$. We draw this as follows:

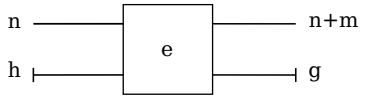


Fig. 1 shows how nQML constructs are compiled to circuits. The compilation process is based on the typing of expressions.

- **Superposition.** A new qubit is added to the state corresponding to $\{\,(\lambda)\,\mathbf{qfalse} + (\lambda')\,\mathbf{qtrue}\,\}$. The remaining n qubits of the state are unaltered, whereas the new qubit is initialized with the transformation matrix:

$$\begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix}$$

- **Let construct and products.** Although the typing rules for these three constructs (simple let, product formation and product elimination) are different when it comes to the types of the participating expressions, they are all the same w.r.t. the number of qubits in the state and they produce the same quantum circuit, which is essentially the sequential composition of two expressions.

- **Quantum conditional.** In the typing of $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$, the condition is the k-th qubit of the state and the pure expressions e_1 and e_2 are not allowed to refer to the k-th qubit (as the environment $\Gamma|_k$ suggests in the figure). The circuit corresponding to condition e is generated first and the k-th qubit of the output state is isolated. This qubit controls the conditional circuit of e_1 and e_2. Notice that it is not strictly true that this conditional circuit is composed of e_1 and e_2. First of all, we have to translate away the (unused) k-th qubit, by inductively transforming the circuits corresponding to e_1 and e_2. Then, we have to extend the input state of the smallest of the two circuits, so that both expect an input state of $n + m - 1 + \max(m_1, m_2)$ qubits.

- **Measurement.** The difference between the quantum conditional and the measurement is that (a) impure expressions are allowed in branches, (b) the branches can use the qubit of the condition, and (c) the qubit of the condition is measured at the end of the circuit. In order to be used by the two branches and (at the same time) be measured at the end, the qubit of the condition must be duplicated (creating a quantum entanglement). This is achieved by using one extra qubit and the controlled CNOT gate. The two expressions may, of course, use the measured value of this qubit. Notice that this is the only circuit which explicitly creates garbage, by measuring the qubit of the condition. Also, this is one of the two circuits that explicitly use qubits from the heap (the other one is generated by superposition).

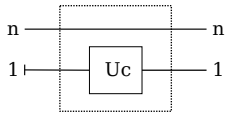
- **Unitary transformation.** In $|e\rangle \to x, x'.\, c$, it is assumed that $c(x, x')$ defines an arbitrary unitary transformation on states of $n + m$ qubits, and this transformation is applied to the result of expression e.

The implementation of our compiler applies several simple optimizations to the generated quantum circuits.[1] In general,

[1]The implementation of nQML can be found at http://www.softlab.ntua.gr/~nickie/Research/nqml/. It consists of approximately 3,200 lines of Haskell code. Parts of it have been written by Michael Lampis.

Superposition:

$\Gamma; n \vdash^{\circ} \{\, (\lambda)\, \mathbf{qfalse} + (\lambda')\, \mathbf{qtrue} \,\} : \mathbf{qbit}[n]; 1$



Let and products:

$\Gamma; n \vdash^{\alpha} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2$

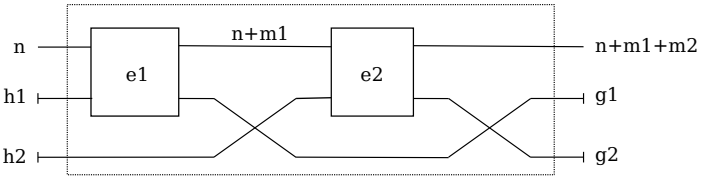
$\Gamma; n \vdash^{\alpha} (e_1, e_2) : \tau; m_1 + m_2$

$\Gamma; n \vdash^{\alpha} \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau; m_1 + m_2$

where:

$\Gamma_1; n \vdash^{\alpha} e_1 : \tau_1; m_1$

$\Gamma_2; n + m_1 \vdash^{\alpha} e_2 : \tau_2; m_2$



Quantum conditional:

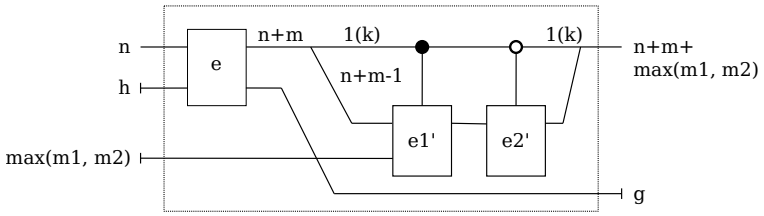
$\Gamma; n \vdash^{\alpha} \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)$

where:

$\Gamma; n \vdash^{\alpha} e : \mathbf{qbit}[k]; m$

$\Gamma|_k; n + m \vdash^{\circ} e_1 : \tau; m_1$

$\Gamma|_k; n + m \vdash^{\circ} e_2 : \tau; m_2$



Measurement:

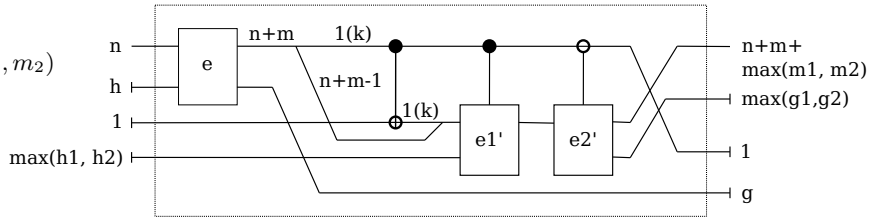
$\Gamma; n \vdash \mathbf{ifm}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau; m + \max(m_1, m_2)$

where:

$\Gamma; n \vdash e : \mathbf{qbit}[k]; m$

$\Gamma; n + m \vdash e_1 : \tau; m_1$

$\Gamma; n + m \vdash e_2 : \tau; m_2$



Unitary transformation:

$\Gamma; n \vdash^{\alpha} |e\rangle \to x, x'.\, c : \tau; m$

where:

$\Gamma; n \vdash^{\alpha} e : \tau; m$

$c(x, x')$ defines a unitary transformation on $n+m$ qubits

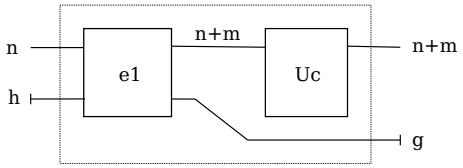


Fig. 1. Compiling nQML expressions to quantum circuits: A typing-directed approach.

the implementation is written in Haskell; it targets the polymorphic language described in [14] and consists of:

- a parser,
- a type checker,
- a first interpreter, based on the denotational semantics of nQML, using density matrices and unitary transformations,
- the compiler from nQML to quantum circuits, defined in this paper, and
- a second interpreter, based on the simulation of quantum circuits generated by our compiler.

The implementation checks that the outputs of the two interpreters coincide, thus testing the correctness of our compiler.

V. Examples

In this section, we outline the use of nQML and its compiler with two relatively simple but historically important examples: Deutsch's algorithm for testing whether a function on one bit is balanced or constant [4], and Grover's algorithm for searching an unsorted database [11].

We begin by providing a couple of auxiliary functions, **not** and **had**, that will be useful in both examples. They correspond to the NOT gate and the Hadamard gate. Their definitions can be given by simple unitary transformations.

```
def not q = |q> -> x, x'.
  if x' = x then 0 else 1;

def had q = |q> -> x, x'.
  (if x then (if x' then -1 else 1) else 1)
    / sqrt(2);
```

The syntax of function definitions in nQML follows the proposed extension with polymorphic functions [14]. Such functions could be treated as macros by the compiler.

We will also abbreviate tuples of more than two elements by writing (x, y, z) instead of $(x, (y, z))$. Furthermore, we will

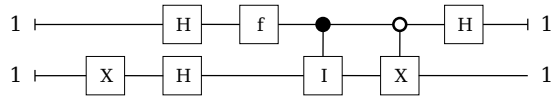


Fig. 2. The circuit produced for Deutsch's algorithm, where f is the parameter: the function that we want to determine if it's constant or balanced.

use **qtrue** as syntactic sugar for $\{\,(0)\,\mathbf{qfalse} + (1)\,\mathbf{qtrue}\,\}$ and **qfalse** as syntactic sugar for $\{\,(1)\,\mathbf{qfalse} + (0)\,\mathbf{qtrue}\,\}$.

A. Deutsch's Algorithm

Deutsch's algorithm (later generalized by Deutsch and Jozsa) was one of the first quantum algorithms to be studied. Supposing that we have a function $f(x) : \{0,1\} \to \{0,1\}$, we want to determine whether this function is *constant*, i.e., $f(0) = f(1)$, or *balanced*, i.e., $f(0) \neq f(1)$, by just computing it once.

There is obviously no classical solution to this problem. The quantum solution employs the trick of computing the function once, with a superposition of the two inputs, then appropriately measuring the result. (The interested reader is refered to the introductory literature in quantum computations for analyses of the algorithm and proofs of correctness.) In nQML, it can be written as follows. The measurement of **had** `i` gives 1 if function f is balanced and 0 if it is constant.

```
def Deutsch f =
  let (i, j) = (had qfalse, had qtrue) in
  let r = if f i then j else not j in
  ifm had i then qtrue else qfalse;
```

The circuit that our compiler produces for this program (just measuring the result and excluding the new qubits for the branches of **ifm**), is shown in Fig. 2.

B. Grover's Algorithm

As a second example, let us see an implementation of Grover's fast database search. Consider an unsorted database with $N = 2^n$ entries and the problem of finding the index of a particular database entry that satisfies some criterion. To simplify things, let us assume that c denotes the index that we are searching for. We first need to implement the query operator, which is a transformation corresponding to a matrix which has 0 everywhere, 1 along the primary diagonal and -1 at the element with coordinates (c, c).

```
def query q = |q> -> x, x'.
  if x = x' then
    if int x = c then -1 else 1
  else
    0;
```

We now define the diffusion operator, a transformation corresponding to the matrix $2P - I$, where P a matrix with 2^{-n} everywhere.

```
def diffusion q = |q> -> x, x'.
  if x = x' then 2 / 2^n - 1 else 2 / 2^n;
```

The algorithm proceeds by repeated iterations of queries and diffusions. Let us now consider the most simple application of Grover's algorithm: searching in a space of size $N = 4$ (with $n = 2$ qubits). In this special case, one iteration is enough to produce the correct result with certainty:

```
def grover2 =
  let qs = (had qfalse, had qfalse) in
  diffusion (query qs);
```

In the general case, $O(\sqrt{N})$ iterations of the two operators are required to obtain the result with a high probability. Consider $N = 16$ (with $n = 4$ qubits). Three iterations suffice:

```
def grover4 =
  let qs = (had qfalse, had qfalse,
            had qfalse, had qfalse) in
  let step1 = diffusion (query qs) in
  let step2 = diffusion (query qs) in
  let step3 = diffusion (query qs) in
  qs
```

The circuit that our compiler produces for `grover4` is shown in Fig. 3. The result is implicitly measured.

VI. Related work

The design of quantum algorithms, such as Shor's algorithm for the factorization of integer numbers in polynomial time [21] and Grover's algorithm for searching an unordered list of n elements in $O(\sqrt{n})$ time [11], has shown that the quantum model of computation is strictly more powerful than the classical model; although both can compute the same set of functions, some functions can be computed in the quantum model strictly faster than in the classical one. Quantum algorithms are usually studied at a low level, either expressed directly in the form of quantum circuits or using appropriate mathematical models. The fact that reasoning about quantum circuits is no easier than reasoning about their classical counterparts has given rise to quantum programming languages, that is, languages that allow programmers to implement quantum algorithms and make use of the added power of the quantum computational model, while respecting its special restrictions.

Knill's conventions for quantum pseudocode [13] was the first proposed formal language for the description of quantum algorithms, tightly connected with the Quantum Random Access Machine. Since then, several quantum programming languages have been proposed; the reader is referred to an excellent (although slightly outdated) survey of the emerging field [5]. Ömer's QCL is an imperative language with quantum primitives and automatic quantum scratch space management [17]. Moreover, van Tonder has proposed a λ-calculus for higher-order quantum programs without measurements [22]. Both languages, however, do not compile to quantum circuits and, in the case of van Tonder's λ-calculus, it is not clear how this can be done. Sanders and Zuliani have defined qGCL, an extension of Dijkstra's guarded command language [18], and they have shown how to compile qGCL to a form of assembly language for a quantum computer [24].

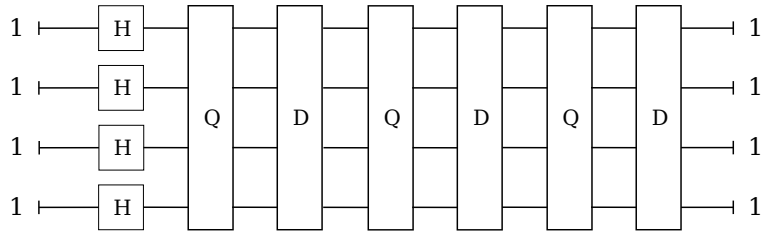


Fig. 3. The circuit produced for Grover's algorithm, where $N = 16$ ($n = 4$). The transformations Q and D correspond to the query and diffusion operators, which are applied iteratively.

Selinger's QPL is a language following the paradigm "quantum data, classical control" [20]. It is functional in nature, although from a programmer's point of view it looks more imperative than functional. QPL allows the programmer to access both classical and quantum memory and includes high-level features such as loops and recursion. Program control is strictly classical and quantum branching can only be implemented indirectly with appropriate unitary transformations. The denotational semantics of QPL is given in the form of superoperators on density matrices. A higher-order extension of QPL in the form of a quantum lambda calculus has also been proposed by Selinger and Valiron [19]. In the same paradigm, Green *et al.* have recently defined Quipper [10], a functional, higher-order quantum programming language designed to be used for implementing large-scale quantum algorithms. They have shown how programs can be compiled to quantum circuits consisting of a large number of gates.

On the other hand, Altenkirch and Grattage's QML is a functional language that follows the paradigm "quantum data and control" [1], [7], [8]. QML comes with a linear type system prohibiting implicit weakening, which would lead to implicit measurements and quantum collapse. The authors describe a way to compile QML programs to quantum circuits in the category FQC of finite quantum computations [6], [9]. Variables in QML correspond to wires in the produced quantum circuit and thus have to be shared implicitly when they are used in several places in a program so as not to break the "no cloning" rule. The sharing of wires is monitored by a linear type system. Altenkirch and Green have recently presented a monadic purely functional interface to quantum programming (the QIO monad) and they provide an implementation in the form of a quantum DSL in Haskell [2]. Again, there is an almost direct translation from QIO to quantum circuits. A similar embedding in Haskell, in the form of arrows, is proposed by Vizzoto *et al.* [23].

VII. Concluding remarks

We have defined a compiler for quantum programs written in the language nQML that follows the paradigm "quantum data and control". The compiler targets quantum circuits in the category FQC of finite quantum computations, defined by Altenkirch and Grattage. We have implemented our compiler as part of nQML's implementation, which is publicly available.

The real challenge in quantum programming, and a definite direction for future work, is the integration of features that are at a higher-level than quantum gates and unitary transformations, for example, reversible binary arithmetic, quantum data structures, etc. The proper integration of such features in quantum programming languages is a hard problem in terms of language design and usability, especially if one wants to keep compatibility with the way in which quantum algorithms are expressed (mostly by non-programmers) today.

Acknowledgment

This research is partially funded by the research project "SemNatComp: Semantic models and technologies for natural computations" (ΓΓΕΤ 11 ROM 11_1_ET30), funded by the Greek General Secretariat for Research and Technology and the European Regional Development Fund, through the operational program "Competitiveness Entrepreneurship & Regions in Transition", action "Billateral Co-operation Greece-Romania 2011-2012".

References

[1] T. Altenkirch and J. Grattage, "A functional quantum programming language," in *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2005, pp. 249–258.

[2] T. Altenkirch and A. S. Green, "The quantum IO monad," in *Semantic Techniques in Quantum Computation*, S. Gay and I. Mackie, Eds. Cambridge University Press, 2009, p. 173205.

[3] J. Brown, *Quest for the Quantum Computer*. Simon and Schuster, 2001.

[4] D. Deutsch and R. Jozsa, "Rapid solutions of problems by quantum computation," *Proceedings of the Royal Society of London*, vol. A 439, pp. 553–558, Dec. 1992.

[5] S. J. Gay, "Quantum programming languages: Survey and bibliography," *Mathematical Structures in Computer Science*, vol. 16, no. 4, pp. 581–600, Aug. 2006.

[6] J. Grattage and T. Altenkirch, "A compiler for a functional quantum programming language," Jan. 2005, manuscript, available from the authors' web page.

[7] ——, "QML: Quantum data and control," Feb. 2005, manuscript, available from the authors' web page.

[8] J. Grattage, "QML: A functional quantum programming language," Ph.D. dissertation, School of Computer Science and School of Mathematical Sciences, The University of Nottingham, Sep. 2006. [Online]. Available: http://etheses.nottingham.ac.uk/archive/00000250/

[9] ——, "An overview of QML with a concrete implementation in Haskell," *Electronic Notes in Theoretical Computer Science*, vol. 270, no. 1, pp. 165–174, 2011, proceedings of the *4th Workshop on Developments in Computational Models* (DCM '08), doi:10.1016/J.ENTCS.2011.01.015, arXiv:0806.2735. [Online]. Available: http://fop.cs.nott.ac.uk/qml

[10] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," in *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, Jun. 2013, to appear.

[11] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, Philadelphia, PA, May 22-24 1996, pp. 212–219.

[12] M. Hirvensalo, *Quantum Computing*, 2nd ed. Springer, 2004.

[13] E. Knill, "Conventions for quantum pseudocode," Los Alamos National Laboratory, Tech. Rep. LAUR-96-2724, 1996.

[14] M. Lampis, K. G. Ginis, M. A. Papakyriakou, and N. S. Papaspyrou, "Quantum data and control made easier," *Electronic Notes in Theoretical Computer Science*, vol. 210, pp. 85–105, Jul. 2008.

[15] M. Lampis, K. G. Ginis, and N. S. Papaspyrou, "Quantum data and control made easier," in *Preliminary Proceedings of the 4th International Workshop on Quantum Programming Languages*, P. Selinger, Ed., Oxford, UK, Jul. 2006, pp. 73–86. [Online]. Available: http://www.mscs.dal.ca/~selinger/qpl2006/

[16] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th ed. Cambridge University Press, 2010.

[17] B. Ömer, "Structured quantum programming," Ph.D. dissertation, Institute of Information Systems, Technical University of Vienna, May 2003.

[18] J. W. Sanders and P. Zuliani, "Quantum programming," in *Proceedings of the 5th International Conference on Mathematics of Program Construction*, ser. Lecture Notes in Computer Science, vol. 1837. London, UK: Springer-Verlag, 2000, pp. 80–99.

[19] P. Selinger and B. Valiron, "A lambda calculus for quantum computation with classical control," *Mathematical Structures in Computer Science*, vol. 16, no. 3, pp. 527–552, 2006.

[20] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, no. 4, pp. 527–586, 2004.

[21] P. W. Shor, "Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[22] A. van Tonder, "A lambda calculus for quantum computation," *SIAM Journal on Computing*, vol. 33, no. 5, pp. 1109–1135, 2004.

[23] J. K. Vizzotto, A. R. D. Bois, and A. Sabry, "The arrow calculus as a quantum programming language," in *Logic, Language, Information and Computation*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5514, pp. 379–393.

[24] P. Zuliani, "Compiling quantum programs," *Acta Informatica*, vol. 41, no. 7, pp. 435–474, Jun. 2005.