Toward an architecture for quantum programming

S.Bettelli,* T.Calarco†L.Serafini‡

23rd October 2018

Abstract

It is becoming increasingly clear that, if a useful device for quantum computation will ever be built, it will be embodied by a classical computing machine with control over a truly quantum subsystem, this apparatus performing a mixture of classical and quantum computation.

This paper investigates a possible approach to the problem of programming such machines: a template high level quantum language is presented which complements a generic general purpose classical language with a set of quantum primitives. The underlying scheme involves a run-time environment which calculates the byte-code for the quantum operations and pipes it to a quantum device controller or to a simulator.

This language can compactly express existing quantum algorithms and reduce them to sequences of elementary operations; it also easily lends itself to automatic, hardware independent, circuit simplification. A publicly available preliminary implementation of the proposed ideas has been realized using the `C++` language.

1 Quantum programming

1.1 Introduction and previous results

In the last decade the field of quantum computing has raised large interest among physicists, mathematicians and computer scientists due to the possibility of solving at least some "hard" problems exponentially faster than with the familiar classical computers [1]. Relevant efforts have been concentrated in two directions: on one hand a (still not so large) set of quantum algorithms exploiting features inherent to the basic postulates of quantum mechanics has been developed [2]; on the other hand a number of experimental schemes have been proposed which could support the execution of these algorithms, moving quantum computation from the realm of speculation to reality (see a review of basic requirements in DiVincenzo [3]).

The link between these two areas is a framework for describing feasible quantum algorithms, namely a computational model, which appears to have settled down to the quantum circuit model[1], due to Deutsch [4], Bernstein and Vazirani [5] and Yao [6]. Though this is satisfactory from the point of view of computational complexity theory, it is not enough for a practical use (programming) of quantum computers, once they will become available.

---

*Laboratoire de Physique Quantique, Université Paul Sabatier, 118, Route de Narbonne, 31062 Cedex Toulouse (France), Tel.: +33 05 61 55 65 73 email: bettelli@irsamc.ups-tlse.fr.

†National Institute of Standards and Technology, 100 Bureau Drive, Stop 8423, Gaithersburg, MD 20899-8423 (USA), Tel.: (301) 975-5347, Fax: (301) 990-1350, email: Tommaso.Calarco@nist.gov. Also at: ECT*, European Centre for Theoretical Studies in Nuclear Physics and Related Areas, Villa Tambosi, Strada delle Tabarelle 286, 38050 Villazzano (Italy), Tel.: +39 0461 314738 (Room I/04), Fax: +39 0461 935007, email: calarco@ect.it.

‡Istituto Trentino di Cultura, Centro per la Ricerca Scientifica e Tecnologica (ITC-IRST), Via Sommarive 18 - Loc. Pantè, 38050 Povo (Italy), Tel.: +39 461 314319, Fax: +39 461 302040, email: serafini@itc.it.

[1] Different computational models, involving physical systems with continuous-variable quantum systems used as computational spaces, are far less developed and will not be considered in this paper. It should be noted however that any such quantum device will most likely require very different interface abstractions from quantum circuits.

1

A few papers can be found in literature concerning the problem of *scalable* quantum programming. An unpublished report by Knill [7], gathering common wisdom of the period about the $QRAM$ model (see section 2.1), moved the first steps towards a standardised notation for quantum pseudo-code pointing out some basic features of a quantum programming language (as an extension of a conventional classical language), though the interest was focused mainly on quantum registers (see section 3.1). This report however did not propose any scheme for implementing an *automatic* translation of the high level notation into circuit objects.

Sanders and Zuliani [8] extended the probabilistic version of an imperative language (`pGCL`) to include three high level quantum primitives (initialisation, evolution and finalisation). The resulting language (`qGCL`) is expressive enough to program a universal quantum computer, though its aim is more to be a tool for verification of the procedures against their specifications[2] (i.e. for verifying that a program really implements the desired algorithm) than to be the starting point for translation of quantum specifications to low level primitives. A related paper by Zuliani [9] showed an interesting technique for transforming a generic `pGCL` program into an equivalent but reversible one, which has a direct application to the problem of implementing quantum oracles for classical functions (see section 5.2).

B.Ömer [10, 11] developed a procedural formalism (`QCL`) which shares many common points with the approach presented in this article about the treatment of quantum registers[3]. `QCL` is however an interpreted environment and is not built on the top of a standard classical language. In this language, just like in `qGCL`, non trivial unitary operations are functions (`qufunct` or `operator`) instead of objects (see sections 3.2 and 5.1), so that their manipulation is subject to the function call syntax; hence automatic operator construction (e.g. controlled operators) and simplification are very difficult to implement, if not impossible. Last, no notion of parallelism for independent operators is present (see 3.3).

In the following section, building on top of these previous works, a list of desirable features for a quantum programming language is presented.

2   Desiderata for a quantum programming language

A common theme in the field of quantum computation is the attempt to think about algorithms in the new "quantum way", without being misled by classical intuition. It could seem that describing quantum computer algorithms in an almost entirely classical way would hide rather than emphasise the difference between quantum and classical computing. The point of this common objection is not, of course, about criticising the assumption that the control system which drives the evolution of the quantum device does not behave according to classical mechanics. Rather, it could be originated by the guess that regarding a part of the quantum resources as *program* and another one as *data*, a sort of quantum von Neumann machine, could lead more naturally to quantum algorithms. This guess has however been disproved by Chuang and Nielsen [12], who showed[4] that, if the program is to be executed deterministically, nothing can be gained by specifying it through a quantum state instead of through a classical one. These considerations can be summarised by saying that quantum algorithms are specified inherently by means of classical programming.

Subject of this article is the investigation and specification of the desirable features of a quantum programming language. The following list is a summary of the main points:

Completeness: the language must be powerful enough to express the quantum circuit model. This means that it must be possible to code every valid quantum algorithm and, conversely, every piece of code must correspond to a valid quantum algorithm.

---

[2]This work was extended in Zuliani's DPhil Thesis (Oxford University) submitted in July 2001, which however was not available at the time of writing.

[3]Quantum registers in `QCL` are however dealt with in a non uniform fashion: in addition to `qureg`, two other register types are present, the `quvoid` and the `quscratch`, which, for a proper type checking, require the knowledge of the quantum device state.

[4]The authors of [12] showed that a programmable quantum gate array, i.e. a quantum machine with a fixed evolution G which deterministically implements the transformation $|d\rangle \otimes |P_U\rangle \to U|d\rangle \otimes |P'_U\rangle$, is such that if U_1 and U_2 are distinct unitary evolutions up to global phase changes, then $|P_{U_1}\rangle$ and $|P_{U_2}\rangle$ are orthogonal. $|P_U\rangle$ here plays the role of the "program" and determines which operation U is to be executed on the "data" register prepared in the state $|d\rangle$. The orthogonality of program states means that the program specification is indeed a classical specification.

Classical extension: the language must include (i.e. be an extension of) a high level *classical computing paradigm* in order to integrate quantum computing and classical pre- and post-processing with the smallest effort. Ad hoc languages, with a limited implementation of classical primitives and facilities, would inevitably fall behind whenever "standard" programming technologies improve.

Separability: the language must keep classical programming and quantum programming separated, in order to be able to move to a classical machine all those computations which do not need, or which do not enjoy any speedup in being executed on, a quantum device.

Expressivity: the language must provide a set of high level constructs which make the process of coding quantum algorithms closer to the programmer's way of thinking and to the pseudo-code modular notation of current research articles. The language must allow an automated scalable procedure for translating and optionally optimising the high level code down to a sequence of low level control instructions for quantum machines.

Hardware independence: the language must be independent from the actual hardware implementation of the quantum device which is going to be exploited. This allows "re-compilation" of the code for different quantum architectures without the programmer's intervention.

The next sections are organised along the following lines. After a general introduction about the computational model (2.1), the guidelines for the envisioned language (2.2) are presented, together with a discussion on hardware requirements (2.3). Section 3 then describes the syntax for high level constructs (3.1, 3.2) as well as the low level, but still hardware independent, primitives to which these constructs get reduced (3.3). Section 4 shows some code samples to clarify the language layout. Section 5 discusses with more details some of the choices for the operator syntax (5.1) and the open problem of the implementation of operators for classical functions (5.2). The appendixes (A.1, A.2, A.3) present possible approaches for implementing the high level primitives previously described.

2.1 The *QRAM* model

Before describing the structure of the proposed quantum language, the quantum computer architecture which it is based on must be clarified. Quantum algorithms are currently described by (more or less implicitly) resorting to the *QRAM* model (see e.g. Knill [7], Knill and Nielsen [13]).

A *QRAM* machine is an extension of a classical random access machine which can exploit quantum resources and which is capable of all kinds of purely classical computations. This classical machine plays two roles: it both performs pre-processing and post-processing of data for the quantum algorithms (trying to keep the quantum processing part as limited in time as possible in order to help preventing decoherence), and controls the quantum subsystem by "setting" the Hamiltonian which generates the required unitary evolution, performing initialisations and collecting the results of measurements. In this scheme the quantum subsystem
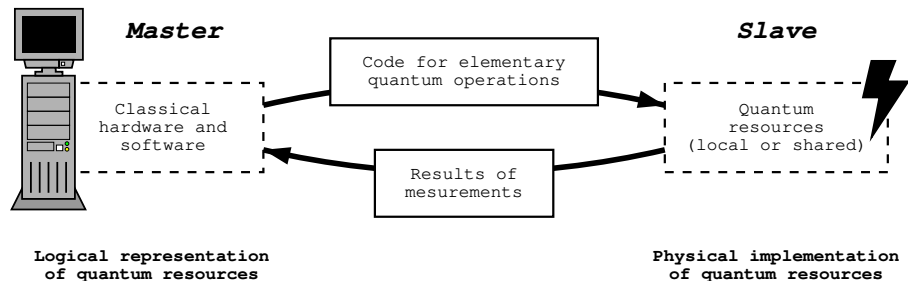

Figure 1: Simplified scheme of a *QRAM* machine. The classical hardware drives the quantum resources in a master-slave configuration; it also performs pre-processing and post-processing of quantum data. The only feedback from the quantum subsystem is the result of measurements.

plays a slave role, while the master classical machine uses it as a black-box co-processing unit. This is summarised in the diagram in fig.1.

It must be noted that quantum resources are not necessarily local[5]; they can be shared among different $QRAM$ machines for quantum type communication or quantum distributed computing. This can be handled by the $QRAM$ model if the machine is given access to the hetero-controlled subsystem and to a classical synchronisation system (a quantum network interface), but this article will not delve into the details of these situations further.

The quantum resource, independently from its actual hardware implementation, is treated as a collection of identical elementary units termed *qubits*; a qubit is an abstract quantum system whose state space is the set of the normalised vectors of the two dimensional Hilbert space $\mathbb{C}^2$, and can encode as much information as a point on the surface of a unit sphere (the Bloch sphere). Due to the structure of the quantum state space, a qubit can encode a superposition of the two boolean digits and is thus more powerful than a classical bit, though the state can not be read out directly. A collection of identical qubits is not subject to the fermion or boson statistics, since the state space of the qubits is in general only a portion of the state space of the quantum system carrying the qubits, to which the statistics applies.

2.2 A scheme for a quantum programming language

As already said, in order to perform a quantum computation, the classical core of the $QRAM$ machine must modify the state of the elements of the quantum subsystem it controls. In the proposed language these elements are indexed by addresses. Though in the following these addresses are treated like unsigned integer numbers, thus abstracting the underlying quantum device to a linear structure, it is by no means assumed that quantum memory is physically organised as an array[6]. The goal of the addresses for quantum elements is simply to hide to the programmer the details of the memory handling.

It is well known that the *no-cloning* theorem excludes the possibility of replicating the state of a generic quantum system[7]. Since the call-by-value paradigm is based on the *copy* primitive, this means that quantum programming can not use call-by-value; therefore a mechanism for addressing parts of already allocated quantum data must be supplied by the language.

In view of these considerations, a new basic data type, the *quantum register* is introduced in the proposed quantum computing language. Quantum register objects are arbitrary collections of distinct qubit addresses. Arbitrary means both that the size is bounded only by the amount of available quantum resources and that the addresses need not be contiguous. Moreover, different quantum registers may overlap, so that the same address can be contained in more than one collection; an example of a set of overlapping registers is shown in fig.2. A qubit is "free" if its address is not referenced by any existing quantum register.

The second data type which is proposed for the quantum language is the *quantum operator*. A quantum operator object encodes the definition of a generic quantum circuit (an acyclic network of quantum gates) and, when fed with a quantum register, it is able to execute the circuit on the supplied register. The operators are the interface which is presented to the programmer for handling unitary transformations. The action of a quantum operator object onto a quantum register object produces, transparently to the programmer, a stream of *byte-code* (a sequence of quantum gate codes and the addresses upon which they must be executed) to be fed into the interface to the quantum device.

Since a quantum operator object embeds the definition of the corresponding circuit as a datum, it is possible to automatically manipulate this definition in a number of different ways, for instance for creating the composition of a pair of circuits, building derived operators (like controlled or adjoint ones) and running simplification routines.

A more detailed description of the proposed scheme is given in appendix A.1.

---

[5]Knill [7] notes that situations arising in quantum communication schemes "require operating on quantum registers in states prepared by another source (for example a quantum channel, or a quantum transmission overheard by an eavesdropper)". It is likely however that these communication schemes will require quite different hardware, so that one would end up with two quantum subsystems better than with one but more complicated.

[6]It was so in very early schemes, like the seminal linear ion trap by Cirac and Zoller [14], but many recent proposals with a concern to scalability are geared toward an at least two dimensional implementation.

[7]In other words, the transformation $|\phi\rangle|0\rangle \rightarrow |\phi\rangle|\phi\rangle$, where $|0\rangle$ is some fixed state and $|\phi\rangle$ is variable, is prohibited in quantum mechanics, because it is not linear.
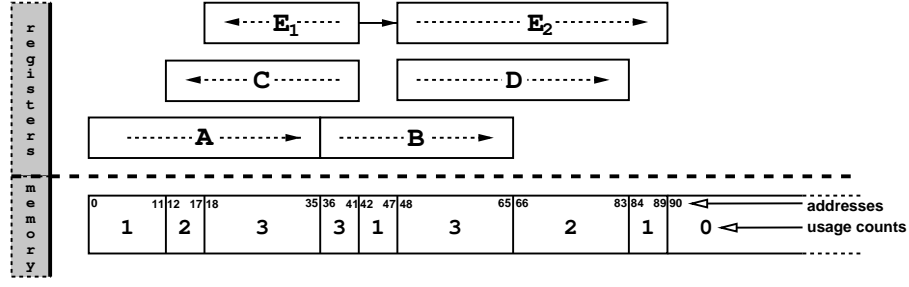
Figure 2: Examples of the organisation of quantum memory within registers. Each register is basically an ordered list of addresses with arbitrary size, like $A = [0, 35]$; registers may overlap, like B and D which both reference the $[48, 65]$ memory segment, or be "inverted", like $C = [41, 12]$. They can also be "disjoint", like $E = ([41, 18], [48, 89])$. The usage status of each qubit in the quantum memory array is maintained by the address manager, which is introduced in appendix A.1. The qubits with addresses from 90 on, in this example, are "free".

2.3   Assumptions on quantum hardware

The actual construction of a quantum computer is a tremendous challenge for both experimental and theoretical researchers in the field. At the moment, it is simply unknown what kind of quantum mechanical system is the most useful for this task. As a result, it is also unknown what kind of *low level* architecture is the most suitable for the implementation of a quantum computer. This is an additional reason, beyond general considerations on the design of high level programming languages, in favour of not referencing a specific physical system that is supposed to support the language design.

The real computational complexity of a quantum algorithm however depends on which capabilities the quantum hardware is endowed with. Though the programming language must be as general as possible, hence adaptable to a variety of quantum devices, some minimal assumptions on the $QRAM$ machine have been made in this paper.

The first assumption concerns the hardware implementation of primitive quantum gates. It seems very plausible that single-qubit gates as well as two-qubit gates between neighbouring quantum subsystems can be executed in constant time, independently of the physical location of the involved qubit(s); two-qubit gates between non-neighbouring locations on the other hand are more challenging. In most of the currently proposed schemes these gates can be implemented only locally, and physical qubit swapping is needed in order to fulfil this condition; this means that the execution time of a two-qubit gate scales linearly with the "distance" (the number of required swaps) between the qubit locations.

Since the physical layout of a quantum device remains unknown to a high level programming language, it is not possible to specify in the language any closeness relation, and the hardware independent language environment can not help but assume that two-qubit gates get executed in constant time, and optimise its behaviour with respect to this assumption. This means, in practice, that the real "complexity" of any specified circuit is in general implementation dependent, and worse than that presented by the language.

The second assumption concerns the implementation of parallelisable gates. A set of applied gates is parallelisable if every qubit in the quantum device is non trivially affected by at most one gate. A parallel quantum machine device is such if it is able to execute a parallelisable set of gates in a time bounded by the execution time of the most expensive gate. This capability is also required in order to perform fault tolerant quantum computation[8]. The language assumes that the underlying quantum hardware is parallel; indeed, only the parallelisation of homogeneous gates (see page 7) is actually exploited.

---

[8]See the discussion in [1], sec.10.6.4 at pag.493 about the threshold theorem for quantum computation.

3 Language primitives

In this section the set of primitives for the proposed language is introduced. The first part concerns the high level primitives (HLP) for the construction and manipulation of quantum registers (3.1) and quantum operators (3.2). Register and operators are classical data structures; their handling does not require any interaction with the quantum device, exception made for the application of an operator to a register, and the initialisation or measurement of a register.

The second part (3.3) introduces a set of low level primitives (LLP) to which the high level specification of a quantum program gets reduced, transparently to the programmer. The LLP are the actual "code" which is sent to the generic quantum device.

3.1 Register handling

As explained in the introductory section, a consequence of the basic rules of quantum mechanics is that an unknown generic state of a quantum register can not be inquired without being destroyed. Since such state encodes the intermediate steps of the computation, it must de facto be regarded as unknown; hence quantum registers can not be read while the computation is running on them. The main implication is that the programmer must think of a quantum register (`Qreg`) as an *interface* to a portion of the quantum device, not as an object carrying a value, unless, of course, he decides to measure it. The register is equivalent to a list of distinct[9] addresses in the quantum memory, and the language provides a set of compliant operations on such lists, which are specified in the following. While all addresses in the same register are distinct, the impossibility to make copies of the register content requires the ability to have more than one register reference the same addresses.

The initialisation and measurement primitives for registers are described in section 3.3, since they are not distinct from the corresponding LLP. All those primitives which are listed here do not involve any interaction with the quantum device.

1 *Register allocation*

Register allocation is the action of creating a quantum register which references only free qubit locations (i.e. not already referenced by an existing register). Quantum registers can be created with arbitrary size, limited only by the capacity of the quantum device, the smallest register interfacing to a single qubit. It is possible to inquire the size of a register. More details on how the allocation status of qubit locations can be handled are given in appendix A.1.

▶ `Qreg a_register(5);`   allocates a register with 5 qubits
▶ `int the_size = a_register.size();`   inquires the size of the register

2 *Register addressing and concatenation*

As already explained, the programmer must be able to operate on registers which overlap parts of already existing ones. Therefore the register objects support the addressing operation (the creation of a register from a subsection of an old register) and the concatenation operation (the creation of a register from the juxtaposition[10] of two old registers). A proper combination of these two operations allows for the creation of a register which is the most general reorganisation of the used portion of the quantum device.

▶ `Qreg a_qubit = a_register[3];`   selects the fourth qubit from `a_register`
▶ `Qreg a_subreg = a_register(2,5);`   selects 5 qubits starting at the third one
▶ `Qreg new_reg = a_subreg & a_qubit;`   concatenates the two registers

3 *Register resizing*

Once a register object has been created, it can be resized (extended or reduced) by adding new qubits or dropping some of them at its beginning. This ability is very useful for routines which need to spawn and reabsorb auxiliary qubits during their execution, provided they take care of a proper uncomputation. Extending a register works like

---

[9]Distinctness is required because multi-qubits operations need distinct physical locations; by assuming that all registers contain by construction only distinct addresses, checking this condition when a register is fed into a quantum operation object can be avoided.

[10]Concatenation requires more bookkeeping than simple addressing, because it must be checked that all the addresses in the composed register are distinct.

spawning a new register with the required additional size, concatenating it with the old register and renaming the latter. Dropping qubits works like the deallocation of only a part of it, taking care that the reduced register contains at least one address.

▶ `my_register += 5;`   adds five qubits to `my_register`

▶ `my_register -= 3;`   drops three qubits from `my_register`

4 *Register deallocation*

Register deallocation is the act of destroying the classical object which represents the interface to a portion of the quantum device. Before being eliminated, this object must release the allocated resources. As a consequence, the "usage" of a part of the quantum device can drop to zero, which means that that part is free for a new allocation.

3.2 Quantum Operators and their manipulation

Quantum operator objects (`Qop`) are the counterpart in the proposed language of quantum circuits, that is unitary transformations on the finite dimensional Hilbert space of a register ($U(2^n)$ if n is the register size). The action of quantum operators is to modify the state of a part of the quantum device, interfaced by a register.

As it is well known[11], all such unitary transformations can be built by finite composition using only matrices acting non-trivially on a one- or two-level subsystem of the original Hilbert space; their number is in general exponential in n. Furthermore, it is possible to approximate[12] each of these matrices using a finite gate subset[13] containing some single-qubit operations and one two-qubit operation (see section 3.3).

The decomposition into these LLP has exponential complexity in general, but this is not the case, of course, for efficient quantum algorithms, which have both time (circuit depth) and space (register size) requirements which are polynomial in the input size; on the other hand, representing a transformation by its matrix elements without any compression scheme is always exponential in the input size. An efficient scheme for quantum operators should therefore be *(de)composition oriented*, i.e. an operator should be stored as the sequence of its factors.

The following list describes the HLP which can be used by the programmer in order to specify a quantum operator. The construction of a quantum operator is a purely classical computation, it does not need to reference quantum registers and must use a polynomial amount of classical resources (space for storage and time for calculations) in order to be useful.

1 *Identity operator*

A quantum operator object can be constructed without any parameter; in this case it corresponds to the identity operator over a quantum register with arbitrary size. It can then be extended using operator composition, see point 7.

▶ `Qop my_op;`   constructs the identity operator

2 *Fixed arity quantum operators*

Each primitive fixed arity quantum operator is associated to a matrix M acting on k qubit lines, i.e. a matrix in $U(2^k)$, and is specified by k index lists $\{\ell^{(h)}\}_{h \in [0,k[}$ (all the lists have the same size s, and all the indexes are distinct even among different lists). The action of such an operator onto a register is to apply M to the qubits in the register indexed by $\ell_j^{(0)}, \ldots, \ell_j^{(k-1)}$ for each $j \in [0, s[$. In simpler words, a single primitive fixed arity quantum operator represents a circuit with s copies[14] of the matrix M in parallel.

---

[11] See [1], sec.4.5.1 and 4.5.2 at pag.189.

[12] The approximation of a single-qubit gate is very efficient: the Solovay-Kitaev theorem proves that an arbitrary single-qubit gate may be approximated to accuracy ϵ using $O(\log^c(1/\epsilon))$ gates from a discrete set, where $c \sim 2$. The approximation of two-qubit gates can be efficiently reduced to previous case if a non trivial two-qubit gate is available as a primitive. The problem of approximating a generic transformation is however very hard: there are unitary transformations on m qubits which take $\Omega(2^m \log(1/\epsilon)/\log(m))$ operations from a discrete set to approximate. See [1], sec.4.5.3 and 4.5.4 at pag.194.

[13] See [1], sec.4.5.3 at pag.194.

[14] Such a way of representing quantum operations reduces the amount of classical resources needed to store the quantum circuit and becomes very useful when the quantum device is able to run a number of independent copies of a quantum gate in parallel.

The order of indexes inside a list is only relevant with respect to the order of indexes in the first list (hence the order of the single list of a single-qubit primitive is arbitrary). An example of how these lists are used is shown in the picture on the right: the circuit corresponds to the creation of a CNOT operator with control index list $\ell^{(0)} = (0, 4, 5)$ and target index list $\ell^{(1)} = (1, 2, 6)$. The symbol used for a CNOT gate is ⊥ .
A summary of primitive quantum operators can be found in table 3.
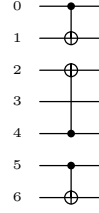
► `Qop my_op = QHadamard(7);`   Hadamard gates acting on first 7 qubits
► `Qop my_op = QCnot(ctrls, targets);`   see above ($\texttt{ctrls} = \ell^{(0)}$, $\texttt{targets} = \ell^{(1)}$)

3   *Macro quantum operators*

The primitive quantum operators previously described correspond to fixed arity quantum gates applied in a parallel fashion. It is useful to consider also a different type of high level primitive (a macro from now on) which is associated to a single transformation in $U(2^n)$ homogeneously parametrised with respect to the number n of qubit lines. A macro with a given dimension n can not in general be reduced to a tensor product of macros of the same type with a smaller dimension. A macro is defined by a single list of addresses, whose order is meaningful (differently from fixed arity quantum operators with arity equal to one).

Quantum macros could become very handy if a specific quantum hardware is built which is able to implement the macro more efficiently than by running the corresponding sequence of less specific low level primitives. In general however their constructor expands the macro into an equivalent sequence of fixed arity operators.

► `Qop my_op = QFourier(7);`   Fourier transform on the first 7 qubits

4   *Qubit line reordering*

Some quantum operations[15] require a permutation of the qubits inside the register they operate on, for instance in order to preserve a standard convention for the most or least significant location. This can be accomplished by properly exchanging the quantum states of the qubits referenced by the register. The language provides a fixed arity primitive (with arity equal to two) which performs such exchanges.

Running this swap operation on the quantum device is however a waste of computation time since it is a completely *classical* data manipulation. Appendix A.3 describes a possible approach for an implementation which, transparently to the programmer, reorganises (on the classical machine) the mapping between qubit addresses and qubit locations, achieving the same result.

► `Qop a_swap = QSwap(5);`   implements the swap of the first 5 qubits

5   *Controlled operators*

A controlled-U operator is a quantum operator C_U which implements the transformation $C_U |x\rangle |y\rangle = |x\rangle U^{\delta_{x,1\ldots1}} |y\rangle$, that is it applies U to the second register when the first is found in the state $|1\ldots1\rangle$. It is a very useful high level primitive for quantum algorithms. This operator is of course unitary and its adjoint is the $C_{U^\dagger}$ operator. Quantum operators have a constructor for such controlled objects, taking as input the operator to be controlled and the size of the control register. They need, in general, to use ancilla qubits during their execution, which are to be supplied by the language internals transparently to the user. Some techniques for the implementation of controlled operators are discussed in appendix A.2.

► `Qop a_controlled_op(U, 5);`   creates a U conditioned by 5 qubits

6   *Operators for classical functions*

Given an algorithm for a classical function $f : \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^m}$, it is often of interest in quantum computation the mapping $U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$ where $\oplus$ is the bitwise `XOR`, the two registers having respectively size n and m. These operators, which implement a classical function in a reversible fashion, are always self-adjoint[16], generally

---

[15]the best known example is the quantum Fourier transform, see [15], where the least significant qubits of the input are transformed into the most significant qubits of the output and vice versa.
[16]Since $U_f^2 |x\rangle |y\rangle = U_f |x\rangle |y \oplus f(x)\rangle = |x\rangle |y \oplus f(x) \oplus f(x)\rangle = |x\rangle |y\rangle$

create entanglement between the input and output registers and are necessary to insert non-injective classical functions in the quantum computing scheme[17].

A quantum language needs the ability to build the U_f operator automatically once the programmer has specified an algorithm for f using the formalism of the underlying classical language. If f is boolean (that is $m = 1$), an easy construction with an additional ancilla qubit can implement the "phase" mapping $P_f|x\rangle = (-1)^{f(x)}|x\rangle$. For a longer discussion about the problems this facility rises refer to section 5.2.

▶ `Qop an_oracle = Qop(f,3,5);`   oracle for f with $n = 3$ and $m = 5$
▶ `Qop a_phase_oracle = Qop(g,4);`   phase oracle for g with $n = 4$ (m is 1)

7 | *Operator composition*

Composing two quantum operator objects returns an operator which represents the concatenation of the underlying circuits in the specified order (i.e. the first operator gets executed first, similarly to how circuits are drawn and differently from the mathematical notation, where operators act on the right). A more elaborated analysis of the advantages of a composition oriented representation can be found in section 5.1.

The language provides three different versions of the composition of operators, in order to achieve increasing efficiency by reusing existing data structures: *concatenation* leaves the two argument operators untouched and returns a third object, *augmentation* modifies the first argument to hold the composed operator without modifying the second argument and *splicing* moves all the data from the second operator (which is left the identity) into the first.

▶ `Qop composed = part_1 & part_2;`   composes two `Qop`s into a third `Qop`
▶ `my_operator &= an_operator;`   extends `my_operator` with `an_operator`
▶ `my_operator << an_operator;`   moves `an_operator` into `my_operator`

8 | *Operator conjugation*

Given a quantum operator U, it is possible to specify its adjoint $U^\dagger$ with the *conjugation* transformation. Conjugation may act on the quantum operator object in place or create a new `Qop`. The proposed quantum language supplies both the mutating and the non mutating transformations.

▶ `Qop adj_operator = !an_op;`   creates the adjoint operator
▶ `an_op.adjoin();`   conjugates the operator

9 | *Operator permutations*

Quantum operators need a method for rearranging the order of the indexes of qubit lines for the underlying circuits, so that simpler operators can be adapted to fit as modules into more complex ones; an example of this is described in section 4.1 where two copies of a circuit for performing the addition of two input registers are rearranged to build a circuit for the addition of three input registers.

A generic permutation can be decomposed into a sequence of adjacent transpositions, but this approach would be highly inefficient in most situations; it is better to have access to "higher level" permutations which can modify the index lists in a single step. The proposed language supplies the *split* and *invert* permutations, both in their mutating and non mutating version. A split permutation leaves the first part of the circuit unaltered while "shifting down" the rest of it; an invert permutation instead "reverses" the central part of a circuit. Shifting the whole circuit (*offset*) is a sub-case of the split operation. A better understanding of these two manipulators can be gained visually with figure 3.

▶ `Qop split = an_op(2,3,SPLIT);`   creates a split operator
▶ `Qop inverted = an_op(2,3,INVERT);`   creates an inverted operator
▶ `Qop shifted = an_op >> 2;`   creates an offset operator
▶ `an_op.offset(2).invert(2,3).split(2,3);`   offsets, inverts and splits the operator

10 | *Application of an operator*

Quantum operators must have a method for running the circuit they embed onto a quantum register supplied by the programmer. Executing an operator means executing

---

[17]Implementation of classical functions is for instance needed in the Grover's algorithms [17], where they are used to evaluate the fitness of a candidate solution to an NP problem.

Figure 3: This figure illustrates the two kinds of index permutations for quantum operators. <u>Left</u>: a generic quantum circuit with various types of quantum gates. <u>Centre</u>: the same circuit after a *split(head, jump)* operation with *head* set to 2 and *jump* set to 3; the first *head* qubit lines are left untouched, while the others are shifted down by *jump* lines. <u>Right</u>: the same circuit after an *invert(head, size)* operation with *head* set to 2 and *size* set to 3; the first *head* qubit lines are left untouched, the following *size* lines are inverted and all the remaining circuit is unmodified.

all of its factors in sequence: the index lists in each primitive operator must be coupled with the address lists in the given quantum register in order to calculate the qubits to be addressed, and the appropriate byte-code must be sent to the quantum device[18]. This process may require ancilla qubits, which need to be spawned and reabsorbed transparently to the user. See appendix A.3 for more details on the steps taken when an operator is executed.

▶ an_operator(a_register);   runs the circuit onto the register

3.3 Low level primitives

Low level primitives are the basic building blocks for the communication between the language and the quantum device. They are divided in non unitary (initialisation and measurement) and unitary (quantum gates). Quantum gates are used to build up all quantum circuits and must of course form a complete set (redundancy is not a problem); choosing a universal set of gates together with a proper syntax for them (see the previous section and 5.1) ensures that all and only quantum circuits in the *QRAM* model can be expressed by the proposed quantum language.

Initialisations and measurements, which are not unitary, are operated directly onto quantum registers. Since registers can have arbitrary sizes, the assigned or returned values do not in general fit into a standard integer type of the classical language, therefore a new type for ordered sets of bits should be introduced (`Qbitset` in the following), with automatic conversion to/from unsigned integers when possible.

As remarked in section 3.2, an efficient scheme for quantum operators should store quantum circuits using one of their factorisations. The smallest factors are called in the following *time slices*. In the proposed language, in a way similar to primitive quantum operators, each time slice is not simply a quantum gate, but embeds a sort of parallelisation restricted to homogeneous gates, which can be acted in parallel over multiple independent qubits. The quantum programmer does not however deal directly with time slices, but he uses only the set of high level primitives described in the previous section.

Storing quantum primitives as a list of time slices fits nicely with the previous requirements for quantum operators, e.g conjugation is easily achieved by iterating through the list in the reverse order and conjugating all its elements[19]; splicing (the third version of operator composition) requires constant time.

---

[18] If a quantum operator must be repeated on the same register a number of times, a mechanism could be provided for caching the byte-code and resend it without recalculating all address pairings from the beginning each time.

[19] All the quantum gates corresponding to time slices should have their adjoint gate implemented as a primitive, so that each quantum operator and its adjoint have exactly the same circuit depth.

1 *Register initialisation and assignment*

The most obvious primitive for a quantum register is its *initialisation* to an element of the computational basis. On a realistic quantum device this involves setting all the qubits of the register to some reference state (e.g. the ground state) and subsequently performing the required unitary transformation to turn it to the representation of an arbitrary integer. It is evident that *assignment* of a `Qbitset` to a quantum register is the same operation as before, and involves a re-preparation of qubits of the register.

▶ `Qreg a_register(5,3);`   initialises a 5 qubits register to $|3\rangle$

▶ `a_register = 7;`   prepares the register again in $|7\rangle$

2 *Register measurement*

The programmer must be able to *measure* a register obtaining an element of the computational basis (that is an integer number or a sequence of boolean values) to be used in the following of the algorithm. This operation is the only blocking primitive with respect to the code flow in the classical core, because the classical environment must wait for the quantum device to execute all the generated byte-code, perform the measurement and return the result.

▶ `Qbitset val = a_register.measure();`   measures a register and saves the result

▶ `int val = a_register.measure();`   casts to integer if possible

3 *Low level unitary gates*

As already said, it is not important which low level unitary gates are chosen to implement a version of the proposed quantum language, as long as the set is complete: this ability to switch to another set must be retained, since it is far from obvious which primitives will most easily be implemented and standardise in future quantum computers.

In this paper (see table 3) the Welsh-Hadamard transform H, the enumerable set of phase shifts R_k (rotations around the z-axis) and their controlled counterparts C_{R_k} are used as a complete[20] set of unitary LLP. Let ϕ_k be $e^{2\pi i/2^k}$ for $k \in \mathbb{N}$ and its conjugate $e^{-2\pi i/2^{|k|}}$ for $k \in \mathbb{Z}/\mathbb{N}$; then the matrix representation of these LLP is as follows:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \boxed{H} \qquad R_k = \begin{pmatrix} 1 & 0 \\ 0 & \phi_k \end{pmatrix} \textcircled{k} \qquad C_{R_k} = \begin{pmatrix} \mathbb{I} & 0 \\ 0 & R_k \end{pmatrix} \textcircled{k}$$

Moreover, H is self-adjoint and R_k is the adjoint of R_{-k} (C_{R_k} is the adjoint of $C_{R_{-k}}$), hence this set is closed under conjugation. In appendix A.2 it is shown that C_U, where U is one of the previous gates, can be expanded into a circuit of gates from the same set with depth bounded by a constant. Primitive quantum operations are built using LLP, but they are logically distinct: there is no need for a one to one correspondence.

This decoupling allows more portable quantum code to be written, since the translation from HLPs into LLPs can be delegated to "more hardware-specific" libraries. Expanding on a previous example, in the picture on the right it is shown the circuit corresponding to the creation of a CNOT operator with control indexes $(0, 4, 5)$ and target indexes $(1, 2, 6)$, reduced to LLP. The relevant identity is $X = H R_1 H$, where X is the NOT port.
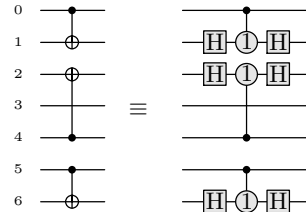

4 Code fragments

A preliminary implementation of the ideas presented in the previous sections has been developed in the form of a `C++` [18] library by the authors and is freely available on the Internet. This section introduces the flavour of the proposed high level language by showing some examples of source code. The code is of course not optimised in order to be more understandable. The `C++` like syntax is summarised in tables 1, 2 and 3, but they are not strictly necessary in order to follow the discussion.

---

[20]This set is redundantly universal; note that $(\mathbb{I} \otimes H) \circ C_{R_1} \circ (\mathbb{I} \otimes H)$ is the CNOT gate, R_2 is the "phase" gate and R_3 is the "$\pi/8$" gate. CNOT, "phase" and "$\pi/8$", together with H are the so called *standard set* of universal gates (see [1] at pag.195).
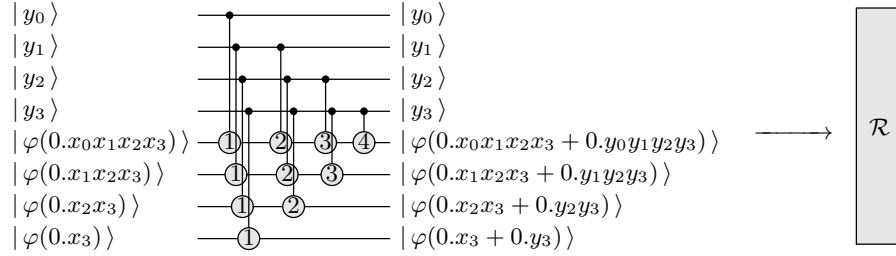
4.1　A three-input adder

This example illustrates how operator compositions, permutations and adjoining can be used in the classical preprocessing stage in order to build a complex parametric quantum operator by reusing smaller circuits.

The following circuit implements the core of the quantum Fourier transform [15] for a four-qubit register, where $|\varphi(\alpha)\rangle$ stands for $\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i\alpha}|1\rangle\right)$. The circuit is different from that usually reported on quantum computing textbooks since the final rearrangement of qubit lines is not performed. For this reason the corresponding unitary operator is named $\widetilde{\mathcal{F}}$. The effect of the transformation is to move information from the computational basis representation into the phase coefficients.
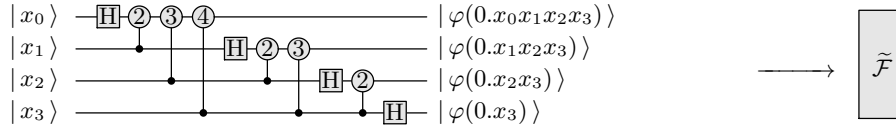
Following an idea of Draper [19], the accumulation of information into the phase coefficients can continue using conditional phase shifts from a second register into the Fourier transformed one. The state of the first register after this stage is the $\widetilde{\mathcal{F}}$ transformed state of $|x + y\rangle$ (modulus 2^4). It is easy to see that all the phase shifts of the same kind involve independent qubit lines, and can therefore be represented by a single time slice.

The obvious step now is to apply $\widetilde{\mathcal{F}}^\dagger$ to $\widetilde{\mathcal{F}}|x + y\rangle$ in order to get the modular addition of x and y (see the left half in the following picture). Once the adder circuit $\mathcal{A}$ is constructed, the process can be iterated in order to build a three-input adder, by summing the content of a third register onto the register which holds the intermediate sum. The right half in the following figure shows the resulting circuit, where $\mathcal{A}_1$ and $\mathcal{A}_2$ represent the $\mathcal{A}$ operator acting on the first and third registers.
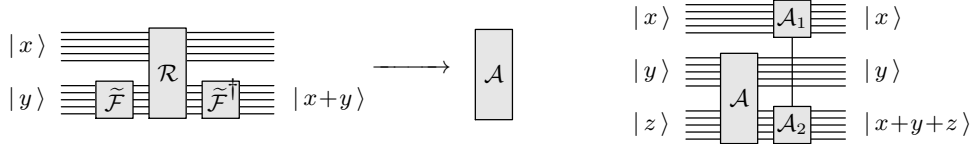

The operator syntax of the proposed quantum language allows to write source code for the implementation of the three-input adder which strictly follows the previous "high level" description. The desired circuit is built by the following function with `size`=4.

```
Qop build_three_adder(int size) {
   Qop phase_shifts;
   for (int i=0; i<size; ++i)
     phase_shifts << QCondPhase(size-i, i+1).offset(i);
   Qop transform = (QFourier(size) & QSwap(size)).offset(size);
   Qop adder_2   = transform & phase_shifts & (! transform);
   Qop adder_3   = (adder_2 >> size);
   adder_3 << adder_2.split(size, size);
   return adder_3;
}
```

The loop sets up the R circuit into the `phase_shifts` operator, which is initialised to the identity, by pushing the conditional phase shifts into it. The first argument to each `QCondPhase` is the number of gates to be stored and the second is the power k of C_{R_k} (see 3.3). Each `QCondPhase` is then offset to the correct position. The `transform` operator contains the Fourier transform over the lower register (`offset(size)`) once the final qubit swap has been reversed with a `QSwap` operator.

`phase_shifts` is then combined with `transform` and its adjoint to form the two-input adder `adder_2`. The three-input adder `adder_3` is built by concatenating two permutations of `adder_2`; the first is offset by `size` qubit lines (thus it acts on the second and third register) and the second is split with a hole in the middle (thus it acts on the first and third registers). Note that the `<<` operator at the end reuses the time slices in `adder_2`, which is therefore lost.

During the construction of `transform` any trivial simplification algorithm embedded in the language (see section 5.1) can simplify the double swapping of qubit lines at the end of the Fourier transform. The same algorithm can simplify the $\widetilde{\mathcal{F}}^\dagger$ at the end of the first two-input adder $\mathcal{A}$ with the $\widetilde{\mathcal{F}}$ at the beginning of the second adder (since they are performed on the same register). The data which is stored inside the quantum operator object `adder_3` is therefore the following, where quantum gates which belong to the same time slice have been grouped together, reducing the number of time slices to 28:
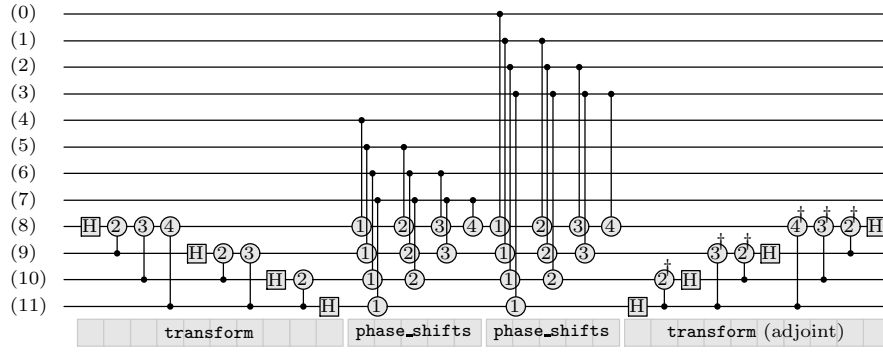
4.2 An example with phase estimation

This example is an implementation of the phase estimation algorithm as a subroutine of the randomised order finding algorithm, in order to illustrate the use of constructors for controlled operators. The order finding algorithm computes the order[21] r of x with respect to N, where x and N are two coprime integer variables with $x < N$. The phase estimation subroutine is used to return a mantissa which approximates s/r where s is a random number in $[0, \dots, r-1]$. The result is to be passed through the continued fraction algorithm (which is completely classical) in order to extract r. The interested reader can find further details in [2].

The phase estimation subroutine accepts two additional parameters, ϵ and n, where $1 - \epsilon$ is the probability bound on having n exact digits in the decimal expansion of s/r. The corresponding circuit is the following, where $M(q)$ is a matrix[22] which implements the multiplication by q modulo N, and M_j stands for $M(x^{2^j})$

---

[21]For positive integers x and N, with no common factors, the order of x modulo N is defined to be the least positive integer r such that $x^r \mod N = 1$.

[22]The definition of $M(q)$ is the following: $M(q)|i\rangle = |iq \mod N\rangle$ if $i < N$, the identity otherwise. If q and N are coprime the corresponding transformation is indeed invertible hence $M(q)$ is unitary. There are various strategies for the implementation of $M(q)$; the simplest one uses classical preprocessing, controlled summations and ancilla qubits. It would make the example too complicated to show the actual construction of $M(q)$, which is however detailed by many authors. See for instance Shor [16].

|0⟩ t qubits { phase register } ... estimate of $\frac{s}{r}$

|1⟩ m qubits { eigen. register } ... M_0 ... M_{t-1} ...
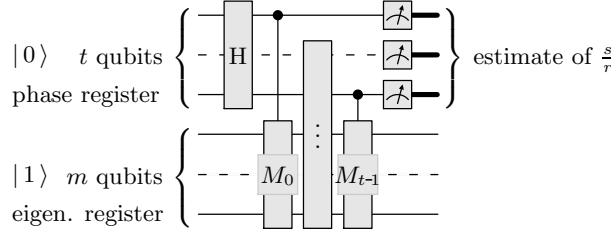
```
Qbitset run_order_finding(int x, int N, int n, float epsilon) {
    int t = n + ceil(log(1+1/(2*epsilon))/log(2));
    int m = ceil(log(N)/log(2));
    int q = x;
    Qop controlled_multiply[t];
    for (int i=0; i<t; ++i, q = ((q*q) % N))
        controlled_multiply[i] << Qop(generate_multiply(q, N), 1);
    Qop mixer = QHadamard(t);
    Qreg phase(t);
    Qreg eigen(m, 1);
    mixer(phase);
    for (int i=0; i<t; i++)
        controlled_multiply[i](phase[i] & eigen);
    return phase.measure();
}
```

The first lines simply calculate the number `t` of qubits in the phase register and the size `m` of the eigenvector register needed in order to host `N`. Then, for each power $q \in \{x^1, \cdots, x^{2^t}\}$, the helper function `generate_multiply` builds the $M(q)$ operator. The returned object (which is a `Qop`) is immediately used as first argument of the controlled operator constructor in order to build a one-qubit controlled $M(q)$ for later use (this last operator acts on `m+1` sized registers).

Everything up to now is classical preprocessing, the interaction with the quantum device starts with the creation and initialisation of the phase and eigenvector registers, followed by the application of the tensor product of `t` Hadamard gates (`mixer`) to the phase register, transforming its state into the uniform superposition of all computational basis states.

Inside the main loop the controlled multiplications are then executed by passing the control qubit and the target register together (using the register concatenation operator). The last line measures the phase register and returns the phase estimate as a bit set.

4.3 An example with Grover's algorithm

This example is the well known Grover's algorithm [17] which finds one or more elements in an unstructured input space with an exponential speed-up with respect to the classical case. It is meant to illustrate the usefulness of having a HLP for the automatic construction of oracle operators from functions specified with the underlying classical language. The premise is that an efficient algorithm is known for the computation of the classical oracle function f. The following example is the simplified version in which there is exactly one good input marked by the oracle, that is $f(x) = $ `true` if and only if x is the searched element $\tilde{x}$. The range of inputs is $[0, \ldots, N-1]$ where $N = 2^n$. The corresponding circuit is the following, where O is the phase oracle operator, M is the so called "inversion about mean" and $G = OM$ is the Grover iteration:

|0⟩ ═ H ═ G ═ 📐 ═ } $\tilde{x}$ with high probability

$\sqrt{N}$ times
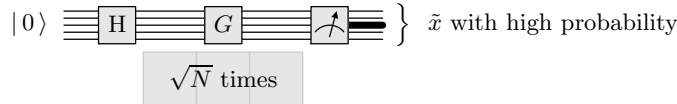
```
Qbitset run_Grover(bool(*f)(int), int n) {
    int repetitions = sqrt(pow(2.0,n));
    Qop phase_oracle(f,n);
    Qop invert_zero(f_0,n);
    Qop mixer = QHadamard(n);
```

14

```
        Qop invert_mean = mixer & invert_zero & mixer;
        Qop grover_step = phase_oracle & invert_mean;
        Qreg input(n);
        mixer(input);
        for (int i=0; i<repetitions; ++i) grover_step(input);
        return input.measure();
}
```

At the beginning the number of iterations to be performed is calculated. It is well known that this number scales as $O(\sqrt{N})$ [17]. Then the `phase_oracle` and `invert_zero` operators are built (`f_0` is a function which returns `true` only when the input is 0). This construction relies on automatic translation from the corresponding classical function provided by the language. This is a major difficulty in the language implementation and is discussed in section 5.2.

Once the previous two operators are ready, it is a matter of composition to build up the `invert_mean` (inversion about mean) and the `Grover_step`. Note that up to now everything is classical preprocessing. The quantum part of the routine starts when an `input` register is created with the appropriate size for holding the input range; then this register is subject to a Hadamard gate on each qubit line in order to generate the uniform superposition of all possible inputs. When the `input` is ready, the `Grover_step` is applied `repetitions` times, the iteration counter being classical. The algorithm is terminated by a register measurement which returns $\tilde{x}$ with high probability.

5 Language internals

5.1 Operator composition and simplification

This section analyses with more details the benefits gained by using quantum operator objects instead of functions for representing quantum circuits; the function-like approach, basically, is adopted both in the `QCL` [10, 11] and in the `qGCL` [8] languages.

A simple example can stress the difference between the two solutions: a `Hadamard_2()` function is available, which accepts a quantum register and an index i inside the register as arguments. It applies two Hadamard gates, to the i-th and to the $i + 1$-th elements of the register, and is invoked with i assuming all the possible values for a valid index in the register:
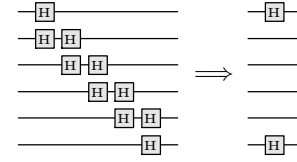

```
        Qreg myreg(size);
        for (int i=0; i<(size-1); i++) Hadamard_2(myreg, i);
```

The insert on the right shows the corresponding circuit (for `size`= 6) and its obvious simplification. It is clear that the quantum language should perform this optimisation, transparently to the user. With a function-like syntax however the quantum code is generated independently by each function call and the optimisation can be done only if the code is buffered and simplified before being sent to the quantum device. Moreover, if the whole loop is repeated with a different register, the buffering and the simplification have to be redone, even though the optimisation depends only on the circuit structure and not on the actual register.

What is really needed is a mechanism for generating the whole circuital description before the quantum device is even fired up, applying algebraic simplifications *once for all*. This is possible if quantum operators are implemented as data structures modifiable at run time, which can be manipulated, composed and simplified *before* allocating the quantum registers. The simplification routines which perform optimisations can be embedded[23] inside the composition primitives. In the proposed language the previous example is coded as:

```
        Qop circuit;
        for (int i=0; i<(size-1); i++) circuit << QHadamard(2).offset(i);
        Qreg myreg(size);
        circuit(myreg);
```

---

[23]If some optimisation routines are too expensive for being embedded, it is possible to leave to the programmer the freedom to force their call, e.g. `circuit.simplify(...)` where the arguments select the simplification strategy.

5.2 The implementation of classical functions

A requirement for a useful quantum language is the ability to implement pseudo-classical operators, that is transformations like $U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$ where $f : \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^m}$ is a classical function and n and m are the sizes of the registers. In section 3.2 it was suggested the introduction of an HLP for the automatic construction of quantum operators from the classical specification of an algorithm for f; this section discusses the problem more extensively.

Lecerf [20] and Bennett [21] have shown that any classical, potentially irreversible, algorithm can be efficiently converted into a reversible one. Since reversible classical algorithms can be converted into equivalent quantum operators efficiently, this means that each classical algorithm computing a function f can be efficiently converted into a quantum operator U_f. A constructive approach to this problem has been shown by Zuliani [9].

What is to be remarked immediately is that if f is "known" through a classical black-box it is useless as far as quantum computing is concerned. The reason for this is that getting the action of f through a black-box requires as many queries as the size of the input space; hence, if the classical preprocessing stage tries to understand f through a black-box, it experiences an exponential slowdown which nullifies any quantum gain. Therefore the constructor of a pseudo-classical operator does not need the ability to *call* the function but the ability to *inspect* its algorithmic definition.

A second remark is that "function" in the current context means a mathematical function, i.e. a deterministic mapping of any input to an output. This definition is more restrictive than the usual meaning in a programming language (a routine), because a routine may depend on the state of the classical machine[24] and may be not terminating on some inputs.

Summarising, the automatic generation of pseudo-classical quantum operators needs access to the specification of the algorithm which implements the function. Moreover, either the algorithm is written in a restricted language which allows only the coding of mathematical mappings, or a "filter" must be applied in order to check for the presence of operations which depend on the state of the classical machine or which may cause the algorithm not to terminate. The filtered algorithm must then be parsed and transformed into a finite size circuit. No scheme for this translation has yet been developed for the proposed quantum language.

Conclusions

It has been proposed a language scheme and a set of high level primitives for programming a $QRAM$ machine. The high level primitives have been studied in order to fit with current circuit model descriptions of quantum algorithms.

The scheme provides an automatic translation and optimisation of high level primitives into low level primitives which are sent to the quantum device. This generated code is still hardware independent, in order to make it easy to switch from real quantum devices to quantum simulators and between different models of quantum hardware and different schemes for low level hardware dependent primitive translation.

There is an ongoing effort[25] to provide a working implementation of the ideas indicated in this paper through a library using the `C++` programming language. A procedure for the automatic translation of classical mappings (see 5.2) is still to be studied.

Once this task is accomplished, it could be a valuable tool for a number of different purposes, like:

- testing the efficiency of different high level simplification and optimisation routines for quantum circuits, including the implementation of pseudo-classical operators;

- testing the efficiency of different schemes for high level to low level and hardware independent to dependent translation routines for quantum circuits;

---

[24]The dependence on the state of the classical machine can be through global or static variables, including random number generators, as well as through run time conditions, like user input

[25]A detailed description of a preliminary language implementation can be found in: S.Bettelli, PhD thesis, University of Trento, in preparation (February 2002). The code will be made available at: `http://sra.itc.it/people/serafini/quantum-computing/qlang.html`
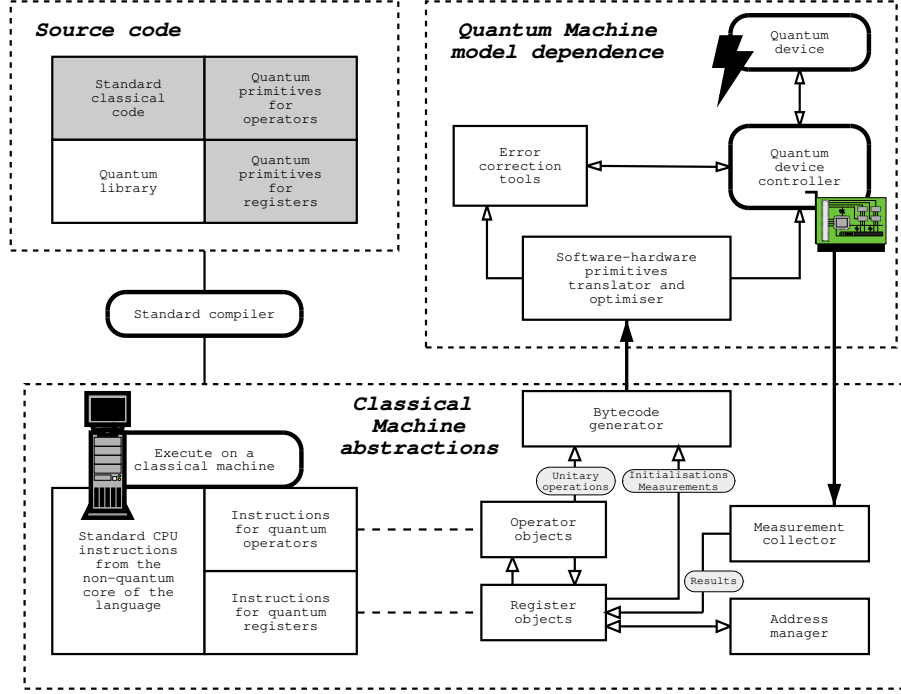
Figure 4: Overall scheme for a quantum device controlled by classical hardware, which details fig.1. The three shaded boxes are the resources available to the programmer for writing a "quantum program". The big dashed boxes contain the source code level, the classical machine control and the quantum machine architecture dependence. See appendix A.1 for more details.

- testing the efficiency of different hardware architectures for the execution of quantum code (with timing simulations);
- having an high level interface for the specification of algorithms which are to be fed into quantum simulators;
- testing the robustness of error correction codes and fault tolerant quantum computation with respect to generic error models, without modifying the simulation libraries;
- quantum programming (when quantum computers will be ready)

A   Implementation details

A.1   A detailed scheme for the language implementation

In section 2.2, the quantum registers and the quantum operators were introduced. Their syntax was examined in section 3.1 and 3.2. This appendix takes a closer look at these data types and at the language environment by describing the scheme presented in fig.4.

The specification of a "quantum program" starts with a source code text file, just like a plain program. The source code syntax uses a standard programming language as a base, and adds primitives for creating and managing quantum *register* objects and quantum *operator* objects. Additional routines (a "quantum library") for common circuits may be used. The code is compiled to an executable by a standard compiler for the base language.

At run-time this executable creates in the classical memory some data structures which correspond to the operator and register objects, and manages their "interaction". The data

structures for quantum registers are basically lists of distinct addresses. The implementation of non-unitary operations (initialisations and measurements) is achieved directly through these interfaces.

The "usage count" of each qubit (the number of registers which are referencing it) is kept by another data structure, the *address manager*, which can not be directly manipulated by the programmer. The address manager knows which qubits are "free" and provides lists of free addresses with the appropriate size when a new register is to be created. An example of a set of overlapping quantum registers with the corresponding status of the address manager was shown in fig.2.

Quantum operator objects, when applied onto registers, calculate which gates are to be executed on which qubit addresses and send this information to the *byte-code generator*, which provides an additional address translation in order to perform qubit swaps without resorting to the quantum device; an approach for these calculations is shown in appendix A.3.

The byte-code generator interfaces directly to a specific (hardware dependent) quantum device driver, exporting a stream of quantum gate codes and the locations where they must be executed. Quantum gate codes are still hardware independent: the translation to the real hardware primitives takes place at this stage. This allows for a very simple way to substitute an emulator to the real device. The device driver can implement additional specific optimisations and error correction tools.

What is appealing here is that all this machinery can be implemented by using a set of libraries and a standard compiler for an object oriented language. Our group has produced a prototype for these ideas using the `C++` [18] language.

A.2   Implementation of controlled circuits

This appendix introduces a possible approach for the construction of multi-controlled circuits. Though it is not part of the language definition and, to some extent, dependent on a particular choice for the elementary gate set, this approach shows that multi-controlled operators can be implemented with the same space and time complexity as the corresponding uncontrolled ones. Many ideas in the following are taken from the classic paper by Barenco et al. [22] and extended with the notion of parallelisation of homogeneous gates introduced in section 3.3.

First, one needs to recognise that the controlled version of each gate in the chosen set of elementary gates $\{H, R_k, C_{R_k}\}_{k\in\mathbb{N}}$ can be implemented by a circuit with depth bounded by a global constant. The controlled R_k is C_{R_k} itself, which is a primitive, so that only the construction of C_H and $C_{C_{R_k}}$ has to be shown.

The construction of C_H is quite easy; first, H is decomposed into a sequence of three rotations around the z, x and z axis (Euler angles decomposition, though usually the chosen axes are z and y):

$$H = i\, R_z\left(\frac{\pi}{2}\right) R_x\left(\frac{\pi}{2}\right) R_z\left(\frac{\pi}{2}\right)$$

Since $HZH = X$, the R_x rotation can be turned into a R_z rotation with the same argument between two Hadamard matrices. The $R_z(\frac{\pi}{2})$ matrix is the same as $e^{-i\frac{\pi}{4}}R_2$, hence:

$$H = i\left(e^{-i\frac{\pi}{4}}\right)^3 R_2 H R_2 H R_2 = e^{-i\frac{\pi}{4}} R_2 H R_2 H R_2$$

The previous relation can be turned into a circuit for C_H by controlling the three phase shifts (since H^2 is the identity) and providing the phase factor with a phase shift $R_3^\dagger$ on the control line:
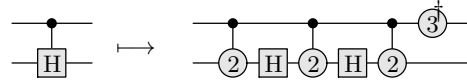
The doubly controlled phase shift $C_{C_{R_k}}$ can be built by adapting a circuit known in literature[26] for the Toffoli gate (the symbols k_1 and k_2 stand for $k+1$ and $k+2$ respectively):
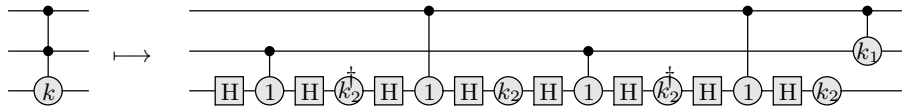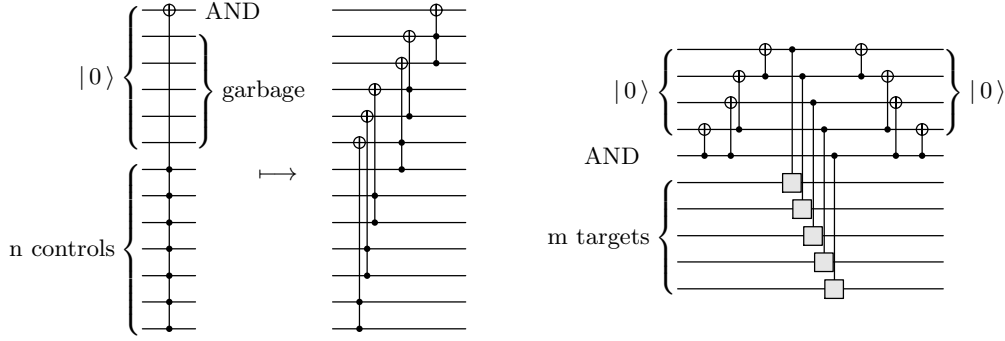

[26]See fig.4.9 on page 182 in [1]

Figure 5: (a) The circuit on the left shows the implementation of the coincidence circuit, that which calculates the AND of all the n control lines. $n-1$ ancilla qubits prepared in the $|0\rangle$ state are needed, including that which holds the result of the computation. If independent Toffoli gates are performed in parallel, the circuit depth grows like $\log n$. The adjoint of the circuit must be applied at the end of the controlled circuit for properly uncomputing the ancillae. (b) The circuit on the right shows how the qubit holding the AND of all the controls can be "copied" enough times to allow for a parallel implementation of m independent operations; indeed this is not a copy but the transformation $(\alpha|0\rangle + \beta|1\rangle) \otimes |0\ldots0\rangle \to \alpha|0\ldots0\rangle + \beta|1\ldots1\rangle$ which leads to a many-particle entangled state. The circuit of course requires $m-1$ additional qubits, to be uncomputed at the end. The depth of the copying and uncopying section grows like $\log m$.

This construction could be generalised to multi-controlled phase shifts, but the depth would scale exponentially with the number of controls. It is easy to see that this circuit performs $C_{C_{R_k}}$ correctly. Whenever one of the control qubits is found in the $|0\rangle$ state, all the gates on the target line cancel out and $C_{R_{k+1}}$ between the controls has no effect. When the control qubits are found in $|11\rangle$ the following relation holds:

$$XR_{k+2} = \begin{pmatrix} 0 & \phi_{k+2} \\ 1 & 0 \end{pmatrix} \implies A = XR_{k+2}^\dagger XR_{k+2} = \phi_{k+2}^* R_{k+1} \implies \phi_{k+1}A^2 = R_k$$

The depth of this circuit is a constant, independently from the parameter k. A Toffoli gate (doubly controlled NOT) can be obtained by enclosing a $C_{C_{R_1}}$ with two Hadamard matrices on the target line, since $HR_1H = HZH = X$. For the same reason, the CNOT gate can be built using C_{R_1}.

With $n-1$ Toffoli gates and $n-1$ ancilla qubits prepared in the $|0\rangle$ state it is possible to calculate the AND of a n-qubit register; if independent Toffoli gates can be applied in parallel the circuit depth grows like $\log n$. The construction is optimal when n is a power of two[27]. An example of this coincidence circuit for $n = 7$ is shown in figure 5.

Once the qubit holding the (quantum) AND of all the controls is ready, it can be used to perform the controlled operations. Since however it is unlikely that a single physical system could be used to control at the same time a number of different qubit lines, this would prevent parallelisation in the controlled operator, changing its complexity. A workaround consists in "copying" the single control into m qubits, m being the maximum number of parallel gates in a single time slice of the uncontrolled operation. Indeed this consists in the transformation $(\alpha|0\rangle + \beta|1\rangle) \otimes |0\ldots0\rangle \to \alpha|0\ldots0\rangle + \beta|1\ldots1\rangle$, which has a logarithmic complexity[28]. Each of the m independent controlled operations can then be performed using a different qubit as control (see figure 5).

The size of the register to be fed into the controlled operator is $m + n$ (n controls and m targets). The number of additional qubits to be used as ancillae is $m + n - 2$, therefore the ratio between the space requirements and the operator "size" is less than 2. The complexity of

---

[27]This is because the first bunch of Toffoli gates calculates $n/2$ ANDs, the second bunch half of that and so on, until the number of gates is one, hence $n/2^q \sim 1$ where q is the number of steps.

[28]During the first step the control is used to perform one copy, then the two qubits can be used during the second step for two copies and so on, hence $\sum_{j=0}^{q-1} 2^j \sim 2^q \sim m$ where q is the number of steps and scales as $\log m$.

the calculation and uncomputation of the AND of all the controls is $\log n$; that of the control copy is $\log m$. These values have to be compared with a (likely) polynomial complexity in m for the uncontrolled operator.

A.3   Techniques for managing qubit addresses

These appendix, and the scheme in figure 6, detail a possible approach for managing qubit addresses. The first part explains how the specifications of circuits and registers are matched to calculate which qubit locations the LLP must be executed on. The second part shows how to implement qubit line swaps as classical operations instead of as hardware ones, as suggested on page 8.

Quantum operators (sec.3.2 and 3.3) are stored as sequences of time slices, each of which is specified by one or more index lists. Each quantum register (sec.3.1) is specified by a list of addresses. Therefore, there is a common basic data structure, a "list", which can be implemented by ordered sets of integer numbers (not containing duplicates). The most important list operation is a transformation $\mathcal{T}$ which takes two lists as input and uses the elements of the former as indexes to select some elements from the latter. In other words, if a and b are lists, then $\mathcal{T}_a(b)$ is a list whose i-th element is b_{a_i}.

The following example will show how $\mathcal{T}$ is used for matching operators with registers. The time slice in the insert on the right is specified by the single index list $\ell = (0, 2, 3)$, and represents the circuit $H \otimes \mathbb{I} \otimes H \otimes H$. When it is executed on a quantum register with associated address list $r = (r_0, r_1, \ldots)$, the language run time environment must pair the elements of ℓ and r, forming a new list $\bar{r} = \mathcal{T}_\ell(r) = (r_{\ell_0}, r_{\ell_1}, \ldots)$ which in the current example gives $\bar{r} = (r_0, r_2, r_3)$.
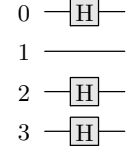
The $\mathcal{T}_\ell$ transformation corresponds to the "index to address translation" stage for the time slice of Hadamard gates in figure 6. The $\bar{r}$ list is not immediately sent to the quantum device, for reasons which will be apparent later, but undergoes a further mapping[29] which implements the "address permutation" $\mathcal{P}$ in figure 6: $\bar{r} \rightarrow \mathcal{P}(\bar{r}) = (\mathcal{P}_{\bar{r}_0}, \mathcal{P}_{\bar{r}_1}, \ldots)$. This translation, though different in nature from the previous one, can use the very same algorithm if a list $p = (\mathcal{P}_0, \mathcal{P}_1, \ldots)$ is provided; in this case $\mathcal{P}(\bar{r})$ is equal to $\mathcal{T}_{\bar{r}}(p)$. Summarising, if a time slice represents a real quantum operation, the addresses which are sent to the quantum device are $\mathcal{T}_{\mathcal{T}_\ell(r)}(p)$ for each list ℓ in the slice.

The behaviour of the language run time environment is however different if the time slice represents a classical permutation, like a "qubit line swap", which is the action of exchanging the quantum state of two qubits. This action can be implemented by three CNOT gates as shown in the following circuit decomposition:
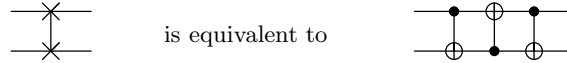
This circuit shows that the exchange is a legal quantum operation and that it can be implemented by sending the appropriate control commands to the quantum device. It is however also obvious that the additional mapping $\mathcal{P}$ between the qubit addresses in the quantum registers and the qubit locations in the quantum device can be used in order to achieve the same result; in this case it is sufficient to modify the list p appropriately.

This approach is preferable for two reasons. First, it concerns only the classical machine, hence leading to a smaller number of quantum operations to be actually performed. Second, it modifies the addresses which are sent to the quantum device transparently to the registers: this means that if two registers overlap and one of them undergoes a number of qubit line swaps, subsequent mappings of the addresses of the other one are influenced too. Therefore, the programmer can still think as if the qubit line swap was a real quantum operation. The additional mapping $\mathcal{P}$ is sufficient to implement the `QSwap` quantum operator, described at pag.8.

---

[29]This mapping is indeed a permutation since each address must correspond to a physical location, and two distinct addresses must map to two distinct locations.
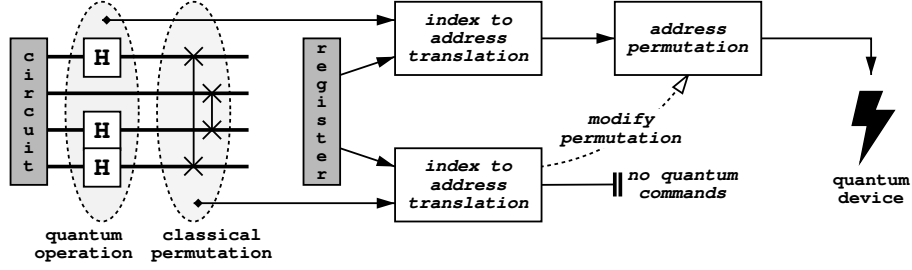
Figure 6: This figure visualises the difference between the treatment of a real quantum operation and a classical permutation. In the first case (a time slice of Hadamard matrices) the index list in the time slice is used to select some of the addresses contained in the quantum register object. These addresses are then subject to a further translation (a permutation) before being fed into the quantum device. In the second case (a swap time slice) the run time environment runs only the first translation, then uses the result in order to modify the permutation function for the following time slices. This change affects all the registers and is thus indistinguishable from a hardware swap for what concerns the programmer.

In the insert on the right a time slice for a qubit line swap operation can be seen; this slice needs two index lists, the corresponding elements of which are the line indexes to be exchanged. The two lists in the example are $\ell^{(0)} = (0, 1)$ and $\ell^{(1)} = (3, 2)$. The first stage of address translation is the same as before, the two lists are combined with the register to form $\bar{r}^{(0)} = \mathcal{T}_{\ell^{(0)}}(r) = (r_0, r_1)$ and $\bar{r}^{(1)} = \mathcal{T}_{\ell^{(1)}}(r) = (r_3, r_2)$.
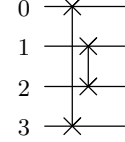
The swaps can then be easily implemented by transposing the $\bar{r}_i^{(0)}$-th and the $\bar{r}_i^{(1)}$-th elements of the list p for each valid i; in the example, this means transposing p_{r_0} with p_{r_3} and p_{r_1} with p_{r_2}. The transposition preserves the property of $\mathcal{P}$ of being a permutation. Summarising, if a time slice represents a qubit line swap no commands are sent to the quantum device; instead, for each address pair (a_i, b_i), where a_i is the i-th element of $\mathcal{T}_{\ell^{(0)}}(r)$ and b_i is the i-th element of $\mathcal{T}_{\ell^{(1)}}(r)$, the elements p_{a_i} and p_{b_i} are transposed in the list p.

Table 1: Quantum registers, the `Qreg` objects (see section 3.1)

	Prototype	Ref.
The register class	`class Qreg;`	sec.3.1
Type for a qubit address	`Qreg::address`	sec.2.2
Type for a register size	`Qreg::size_type`	sec.2.1
Type for a bit set	`Qbitset  or  unsigned integers`	p.10
Register constructors	`Qreg::Qreg(size_type s = 1, value v = 0);`	p.6, 11
	`Qreg::Qreg(const Qbitset &the_bits);`	p.6, 11
Register assignment	`void Qreg::operator=(value v) const;`	p.11
	`void Qreg::operator=(const Qbitset &the_bits) const;`	p.11
Measurement (blocking)	`Qbitset Qreg::measure(void) const;`	p.11
Register copy constructor	`Qreg::Qreg(const Qreg &a_register);`	
Register destructor	`Qreg::~Qreg();`	p.7
Qubit addressing	`Qreg Qreg::operator[](address a) const;`	p.6
	`Qreg Qreg::operator()(address a, size_type s) const;`	p.6
Register concatenation	`Qreg operator&(const Qreg &r_1, const Qreg &r_2);`	p.6
	`Qreg &Qreg::operator&=(const Qreg &second_register);`	p.6
Register resizing	`Qreg &Qreg::operator+=(size_type the_size);`	p.6
	`Qreg &Qreg::operator-=(size_type the_size);`	p.6
Register size	`Qreg::size_type Qreg::size(void) const;`	p.6

Table 2: Quantum operators, the `Qop` objects (see section 3.2)

	Prototype	Ref.
The operator class	`class Qop;`	sec.3.2
Default constructor	`Qop::Qop();`	p.7
Copy constructor	`Qop::Qop(const Qop &op);`	
Controlled operators	`Qop::Qop(const Qop &op, size_type ctrl);`	p.8
Oracle operators	`Qop::Qop(int(*f)(int), size_type in, size_type out);`	sec.5.2
Phase oracle operators	`Qop::Qop(bool(*f)(int), size_type in);`	sec.5.2
Operator composition	`Qop operator&(const Qop &op_1, const Qop &op_2);`	p.9
	`Qop &Qop::operator&=(const Qop &op);`	p.9
	`Qop &Qop::operator<<(Qop &op);`	p.9
Operator conjugation, mutable	`Qop &Qop::adjoin(void);`	p.9
Operator conjugation, const	`Qop Qop::operator!(void) const;`	p.9
Operator split, mutable	`Qop &Qop::split(size_type head, size_type jump);`	p.9
Operator invert, mutable	`Qop &Qop::invert(size_type head, size_type size);`	p.9
Operator split/invert, const	`Qop Qop::operator()(size_type,size_type,op_type) const;`	p.9
Operator offset, mutable	`Qop &Qop::offset(size_type jump);`	p.9
Operator offset, const	`Qop Qop::operator>>(size_type jump) const;`	p.9
Operator application	`void Qop::operator()(const Qreg &a_register) const;`	p.9

Table 3: Computational primitives (see section 3.2 and 3.3)

	Prototype	Ref.
Hadamard mixing	`class QHadamard;`	p.11,H
Phase shift(Z-rotation)	`class QPhase;`	p.11,R_k
Conditional phase shift	`class QCondPhase;`	p.11,C_{R_k}
Controlled NOT	`class QCnot;`	p.19
Toffoli gate	`class QToffoli;`	p.19
Swap gate (classical)	`class QSwap;`	app.A.3
Discrete Fourier transform	`class QFourier;`	[15]

References

[1] M.A.Nielsen, I.L.Chuang, "Quantum Computation and Quantum Information", Cambridge University Press, New York, NY, **2000**

[2] R.Cleve, A.Ekert, C.Macchiavello, M.Mosca, "Quantum Algorithms Revisited", *Proc. of the Royal Society of London*, **A454** (1998) pp.339-354, `quant-ph/9708016`

[3] David P.DiVincenzo, "The Physical Implementation of Quantum computation", *Fortschritte der Physik*, **48** (2000) pp.771-783, `quant-ph/0002077`

[4] David Deutsch, "Quantum computational networks", *Proc. of Royal Society of London*, **A425** (1989) pp.73-90

[5] E.Bernstein, U.Vazirani, "Quantum complexity theory", *Proc. of the 25th ACM Symposium on the Theory of Computation*, (1993) pp.11-20

[6] A.C.Yao, "Quantum Circuit Complexity", *Proc. of the 34th IEEE Symposium on Foundations of Computer Science*, (1993) pp.352-361, also available at
`http://feynman.stanford.edu/qcomp/yao/index.html`

[7] Emanuel H. Knill, "Conventions for Quantum Pseudocode", unpublished, `LANL report LAUR-96-2724`

[8] J.W.Sanders, P.Zuliani, "Quantum Programming", *Math. of Program Constr., Springer LNCS*, **1837** (2000) pp.80-99, also as `TR-5-99` (1999), Oxford University, available at
`http://web.comlab.ox.ac.uk/oucl/publications/tr/index.html`

[9] P.Zuliani, "Logical reversibility", *IBM Journal of Research and Development*, **45** (2001) pp.807-818, also as `TR-11-00`, Oxford University, available at
`http://web.comlab.ox.ac.uk/oucl/publications/tr/index.html`

[10] Bernhard Ömer (1998), "A Procedural Formalism for Quantum Computing",
Master Thesis (theoretical physics), `http://tph.tuwien.ac.at/~oemer/qcl.html`

[11] Bernhard Ömer (2000), "Quantum Programming in QCL",
Master Thesis (computer science), `http://tph.tuwien.ac.at/~oemer/qcl.html`

[12] M.A.Nielsen, I.L.Chuang, "Programmable quantum gate arrays", *Phys.Rev.Lett.*, **79** (1997) pp.321-324, `quant-ph/9703032`

[13] E.H.Knill, M.A.Nielsen, "Theory of quantum computation", Supplement III, Encyclopaedia of Mathematics (Summer 2001), `quant-ph/0010057`

[14] J.I.Cirac, P.Zoller, "Quantum Computation with Cold trapped ions", *Phys.Rev.Lett.*, **74** (1995) pp.4094-4097

[15] D.Coppersmith, "An Approximate Fourier Transform Useful in Quantum Factoring", unpublished, Technical report IBM, Research report 19642, IBM, 07/12/1994

[16] Peter W.Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM J. Computing*, **26** (1997) pp.1484-1509, also as `quant-ph/9508027`

[17] Lov K. Grover, "A fast quantum mechanical algorithm for database search", *Proc. of the 28th Annual ACM Symposium on the Theory of Computing (STOC)*, (1996) pp.212-219 `quant-ph/9605043`

[18] Bjarne Stroustrup, "The C++ Programming Language (3rd edition)", Addison Wesley Longman, Reading, MA, **1997**

[19] Thomas G. Draper, "Addition on a Quantum Computer", unpublished,
`quant-ph/0008033`

[20] Y.Lecerf, "Machines de Turing réversibles. Récursive insolubilité en $n \in \mathbb{N}$ de l'équation $u = \theta^n u$, où θ est un isomorphisme de codes", *Comptes rendus de l'Académie française des sciences*, **257** (1963) pp.2597-2600

[21] C.H.Bennett, "Logical reversibility of computation", *IBM J.Res.Dev.*, **17** (1973) pp.525-532

[22] A.Barenco, C.H.Bennett, R.Cleve, D.P.DiVincenzo, N.Margolus, P.Shor, T.Sleator, J.Smolin, H.Weinfurter, "Elementary gates for quantum computation", *Physical Review A*, **52** (1995) pp.3457-3467, `quant-ph/9503016`