A Procedural Formalism for Quantum Computing

Bernhard Ömer

23th July 1998

Department of Theoretical Physics

Technical University of Vienna

E-mail: `oemer@tph.tuwien.ac.at`
Homepage: `http://tph.tuwien.ac.at/~oemer`

Abstract

Despite many common concepts with classical computer science, quantum computing is still widely considered as a special discipline within the broad field of theoretical physics. One reason for the slow adoption of QC by the computer science community is the confusing variety of formalisms (Dirac notation, matrices, gates, operators, etc.), none of which has any similarity with classical programming languages, as well as the rather "physical" terminology in most of the available literature.

QCL (**Q**uantum **C**omputation **L**anguage) tries to fill this gap: QCL is a hight level, architecture independent programming language for quantum computers, with a syntax derived from classical procedural languages like C or Pascal. This allows for the complete implementation and simulation of quantum algorithms (including classical components) in one consistent formalism.

Chapter 1 is an introduction into the basic concepts of quantum programming, a complete language reference of QCL can be found in chapter 2 and chapter 3 gives some examples including the QCL implementation of Shor's factorisation algorithm. The sourcecode of the QCL interpreter is available at `http://tph.tuwien.ac.at/~oemer/qcl.html`.

Contents

Chapter 1

Introduction

1.1 Models of Quantum Computation

In classical information theory, the concept of the universal computer can be represented by several equivalent models, corresponding to different scientific approaches. From a mathematical point of view, a universal computer is a machine capable of calculating *partial recursive functions*, computer scientists often use the *Turing machine* as their favourite model, an electro-engineer would possibly speak of *logic circuits* while a programmer certainly will prefer a *universal programming language*.

As for quantum computation, each of these classical concepts has a quantum counterpart:

Model	**classical**	**quantum**
Mathematical	partial recursive funct.	unitary operators
Machine	Turing Machine	QTM
Circuit	logical circuit	quantum gates
Algorithmic	univ. programming language	QCL

Table 1.1: classical and quantum computational models

1.1.1 The Mathematical Model of QC

The moral equivalent in QC to partial recursive functions are *unitary operators*. As every classically computable problem can be reformulated as calculating the value of a partial recursive function, each quantum computation must have a corresponding unitary operator.

1.1.1.1 Unitary Operators

A unitary operator U over the *Hilbert space* $\mathcal{H}$ is a linear operator which matches the following condition:

$$\begin{aligned} U(\alpha\,|\psi\rangle + \beta\,|\phi\rangle) &= \alpha\,U\,|\psi\rangle + \beta\,U\,|\phi\rangle \\ \text{and} \quad U^\dagger = U^{-1} &\quad \text{with} \quad |\psi\rangle, \phi \in \mathcal{H} \end{aligned} \tag{1.1}$$

While unitary operators fully describe the quantum computation itself, this would be of no use, since quantum states cannot be directly observed.

According to the Copenhagen interpretation of quantum physics, the setup and outcome of any quantum mechanical experiment must be formulated in classical terms. We thus need 2 additional Operations for setting up a defined initial state $|\psi_0\rangle$ and for measuring the output.

1.1.1.2 Initialisation

The *reset operator* R is a constant operator over $\mathcal{H}$ which resets a general $|\psi\rangle$ to $|\psi_0\rangle$. Usually the base of $\mathcal{H}$ is chosen such that $|\psi_0\rangle = |0\rangle$.

1.1.1.3 Measurement

The *measurement operator* $M(\mathcal{O})$ of the observable $\mathcal{O} = \{\mathcal{E}_0, \mathcal{E}_1, \ldots \mathcal{E}_k\}$, where $\mathcal{E}_i$ is a mutually orthogonal decomposition of $\mathcal{H}$, randomly applies a *projection operator* $P(\mathcal{E}_m)$ to the measured state $|\psi\rangle$ biased by the probability $p_m = \langle P(\mathcal{E}_m)\rangle$ and renormalises the result.

The classical outcome of the measurement is $\mu(m)$ where μ is a mapping $\{0, 1, \ldots k\} \to \mathbf{R} \times \{\text{physical unit of}\, \mathcal{O}\}$. [1]

Since the mathematical model treats unitary operators as black boxes, no complexity measure is provided.

1.1.2 The Machine Model of QC

In analogy to the classic Turing Machine (TM) several propositions of Quantum Turing Machines (QTM), as a model of a universal quantum computer have been made [3, 1].

The complete machine-state $|\Psi\rangle$ is thereby given by a superposition of base-states $|l, j, s\rangle$, where l is the inner state of the head, j the head position and s the binary representation of the tape-content. To keep $\mathcal{H}$ separable,

[1] Since we are not interested in physical values, and $\mathcal{O}$ normally corresponds to a register of qubits, we can use the *standard observable*, so $\mu(i) = i$.

the (infinite) bit-string s has to meet the *zero tail state* condition i.e. only a finite number of bits with $s_m \neq 0$ are allowed.

The quantum analogon to the transition function of a classic probabilistic TM is the *step operator* T, which has to be unitary to allow for the existence of a corresponding Hamiltonian[2] and meet locality conditions for the effected tape-qubit, as well as for head movement.

QTMs provide a measure for execution times, but – as with the classical TM – finding an appropriate step operator can be very hard and runtime-complexity (i.e. the number of applications of T in relation to the problem size) remains an issue. Outside quantum complexity theory, QTMs are of minor importance.

1.1.3 The Gate Model of QC

Quantum circuits are the QC equivalent to classical boolean feed-forward networks, with one major difference: since all quantum computations have to be unitary, all quantum circuits can be evaluated in both directions (as with classical reversible logic). Quantum circuits are composed of elementary gates and operate on qubits, thus $\dim(\mathcal{H}) = 2^n$ where n is the (fixed) number of qubits.

The actual unitary transformation performed by a m-qubit gate depends on the 2^m-dimensional subspace $\mathcal{H}' \subseteq \mathcal{H}$ corresponding to the particular positions of the input qubits. Let G be the 2 dimensional operator ($m = 1$) of a gate operating on the k-th qubit of the state $|\psi\rangle \in \mathcal{H} = \mathbf{C}^{2^n}$ and $I(l)$ the l-qubit identity operator, then the resulting operator $G_k(n)$ is

$$G_k(n) = I(k-1) \times G \times I(n-k-1) \quad \text{with} \quad I(l)\,|\psi\rangle = |\psi\rangle,\ |\psi\rangle \in \mathbf{C}^{2^l}. \tag{1.2}$$

A quantum m-qubit gate in an n-qubit Hilbert space therefore represents a class of $\frac{n!}{(n-m)!}$ unitary operators.

To allow for implementation of all possible unitary transformations, a universal set of elementary gates must be available, out of which composed gates can be constructed. One possible set (as proposed by Deutsch) [4] is e.g. $\{\theta \in [0, 2\pi)\,|D(\theta)\}$ with

$$D(\theta) : |i,j,k\rangle \to \begin{cases} i\cos\theta\,|i,j,k\rangle + \sin\theta\,|i,j,1-k\rangle & \text{for } i = j = 1 \\ |i,j,k\rangle & \text{otherwise} \end{cases} \tag{1.3}$$

[2] In continuous-time quantum mechanics, the operator U of temporal propagation is given by $U(t) = e^{-iH(t-t_0)/\hbar}$. Since H has only real eigenvalues, U must be unitary.

However, even one $D(\theta)$ with irrational θ/π is sufficient, to approximate any unitary operator to arbitrary precision.[3]

As opposed to the operator formalism, the gate-notation is an inherently constructive method and – other than QTMs – the complexity of the problem is directly reflected in the number of gates necessary to implement it.

1.1.4 Programming Languages

When it comes to programming and the design of non-classic algorithms, we can look at operator-algebra as the specification and quantum gates as the assembly language of QC.

The lack of typical programming techniques as local quantum variables, scratch space management and dynamic register lengths as generic features in either formalism makes the the actual implementation[4] of non–trivial quantum algorithms very complex and difficult (see [13] for an example) since the classical "divide and conquer" approach is only of limited use.

Another problem is the classical control structure: Due to their probabilistic nature, evaluation of measurements and conditional retries are part of almost any quantum algorithm, however they cannot be described within the traditional formalism.

The purpose of QCL is, to fill this gap and serve as a high-level programming language for quantum computing.

1.2 Principles of Quantum Computing

1.2.1 Qubits

To implement a computational model as a physical device, the computer must be able to adept different internal states, provide means to perform the necessary transformations on them and to extract the output information. The correlation between the physical and the logical state of the machine is arbitrary (as long it is consistent with the desired transformations) and requires interpretation.

In an ordinary RAM module, the common quantum state of thousands of electrons is interpreted as only one bit. The logical state is determined by the expectation value of its register contents (e.g. tension of a capacitor)

[3]Note that $D(\pi/2)$ defines the *Toffoli gate*, which is universal for classical reversible logic.

[4]This means the constructive specification of the elementary gate sequence, not its experimental realisation.

The interpretation as (classical) bits is performed by comparing the measured value to a defined threshold, while the great number of particles guarantees that the uncertainty of the measurement is small enough ($O(1/\sqrt{n})$) to make errors practically impossible.

In a quantum computer, information is represented directly as the common quantum state of many subsystems. Each subsystem is described by a combination of two "pure" states interpreted as $|0\rangle$ and $|1\rangle$ (quantum bit, qubit). This can e.g. be realised by the spin of a particle, the polarisation of a photon or by the ground state and an excited state of an ion.

1.2.2 Entanglement of States

Due to the one-to-one relation between logical and physical state in a quantum computer, a quantum register containing more than one qubit can not be described by simply listing the states of each qubit. In fact, the "state of a qubit" becomes a meaningless term[5]

Given an isolated system of two qubits, its state can be described by four complex amplitudes $a|0,0\rangle + b|1,0\rangle + c|0,1\rangle + d|1,1\rangle$. You can define the expectation value for the first qubit, which is $\sqrt{bb^* + dd^*}$ but there is no isolated state for the first qubit anymore like e.g. $(a+c)|0\rangle + (b+d)|1\rangle$ since $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$ does not implicate that $|a+c|^2 + |b+d|^2 = 1$.

Therefore, manipulations on a single qubit effect the complex amplitudes of the overall state and have a global character. To describe the combined state $|\psi\rangle$ of n entangled qubits, 2^n complex numbers are necessary.

$$|\psi\rangle = \sum_{i=0}^{2^n-1} c_i\,|i\rangle \quad \text{with} \quad \sum_{i=0}^{2^n-1} c_i^* c_i = 1 \quad \text{and} \quad c_i \in \mathbf{C} \tag{1.4}$$

1.2.3 Reversibility

To keep the computation coherent, quantum registers must be kept isolated, to avoid entanglement with the environment. The entropy of such a system has to remain constant since no heat dissipation is possible, therefore state changes have to be adiabatic, which requires all computations to be reversible.

Every reversible operation can be described by a *unitary* operator U which matches the condition $U^{-1} = U^\dagger$. Compositions of unitary operators are also unitary since $(UV)^{-1} = V^\dagger U^\dagger$. The restriction to unitary operators can also

[5]The occasionally found notation $|\psi\rangle|\phi\rangle$ for multiple quantum registers instead of mere product states is somewhat misleading in this respect. In this paper, $|\psi\rangle|\phi\rangle$ always denotes a vector product and registers are labelled with subscripts like $|\cdot\rangle_\mathbf{s}$ if necessary.

be directly derived for the operator of temporal propagation $U = e^{-iHt/\hbar}$. Since the Hamilton operator H is an observable it has only real eigenvalues and $({}^\dagger U) = U^{-1} = e^{iHt/\hbar}$.

A general unitary transformation in the two dimensional Hilbert space $\mathbf{C}^2$ can be defined as follows:

$$U(\theta,\delta,\sigma,\tau) = \begin{pmatrix} e^{i(\delta+\sigma+\tau)}\cos\frac{\theta}{2} & e^{-i(\delta+\sigma-\tau)}\sin\frac{\theta}{2} \\ -e^{i(\delta-\sigma+\tau)}\sin\frac{\theta}{2} & e^{i(\delta-\sigma-\tau)}\cos\frac{\theta}{2} \end{pmatrix} \quad \text{with} \quad \theta,\delta,\sigma,\tau \in \mathbf{R} \tag{1.5}$$

If this operator can be applied to arbitrary 2-dimensional subspaces of $\mathcal{H}$, then any unitary transformation can be constructed by composition. If only subspaces corresponding to a subset of qubits are allowed, which is the case for many proposed architectures, among them also the linear ion trap (*Cirac-Zoller gate* [2]), then an additional 4-dimensional 2-qubit operator is needed to obtain a mixing between separate qubits [6].

One possibility for this operator is the 2-qubit XOR which is defined as $XOR : |x,y\rangle \to |x, x \oplus y\rangle$ or in matrix notation:

$$XOR = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.6}$$

A quantum computer which is capable of performing general single qubit and *XOR* operations can therefore perform any possible operation and is in this sense universal.

1.2.4 Initialisation

To set a quantum computer to the desired input state $|\psi\rangle$, it suffices to provide means to initially "cool" all qubits to $|0\rangle$ and then apply a unitary transformation U which matches the condition $U|0\rangle = |\psi\rangle$. One might think of U as a base transformation which trivially exists for any desired $|\psi\rangle$.

1.2.5 Measuring States

Measuring n qubits reduces the dimensionality of $\mathcal{H}$ by a factor of 2^n. The outcome of the measurement is biased by the probability amplitude for a certain bit configuration.

Consider two quantum registers with n and m qubits in the state

$$|\psi\rangle = \sum_{i=0}^{2^n-1}\sum_{j=0}^{2^m-1} c_{i,j}|i,j\rangle \quad \text{with} \quad \sum_{i,j} c^*_{i,j}c_{i,j} = 1 \tag{1.7}$$

The base-vectors $|i,j\rangle$ are interpreted as a pair of binary numbers with $i < 2^n$ and $j < 2^m$. The probability $p(I)$ to measure the number I in the first register and the according post measurement state $|\psi'_I\rangle$ are given by

$$p(I) = \sum_{j=0}^{2^m-1} c^*_{I,j} c_{I,j}, \quad \text{and} \quad |\psi'_I\rangle = \frac{1}{\sqrt{p(I)}} \sum_{j=0}^{2^m-1} c_{I,j} |I,j\rangle \tag{1.8}$$

The measurement of qubits is the only non unitary operation, a quantum computer must be able to perform during calculation.

1.3 Quantum Programming

1.3.1 Quantum Parallelism

Since all unitary transformations are linear operators, any operation performed on a quantum state is simultaneously applied to all its base-vectors, thus

$$U \sum_i c_i |i\rangle = \sum_i c_i U |i\rangle \tag{1.9}$$

This unique feature of quantum computers is called *quantum parallelism*. Since the number of base-vectors exponentially increases with the number of qubits, it is possible to solve certain problems (e.g. Deutsch-Jozsa's problem [4]) in polynomial time (i.e. the number of elementary operations is a polynomial in the length of the input) where a classical computer would need an exponential number of steps.

1.3.2 Quantum Registers

1.3.2.1 2-register states

Since the qubits of a quantum computer constitute a possible entangled overall state, there is – strictly speaking – no such thing as a sub-state for particular qubits, since an $n + m$ qubit state of the form

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^m-1} a_{ij} \, |i,j\rangle \tag{1.10}$$

usually cannot be written as a product state $|\phi\rangle|\chi\rangle$ of an n and an m qubit state, because generally

$$|\psi\rangle \neq |\phi\rangle \times |\chi\rangle = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^m-1} b_i c_j \, |i,j\rangle \tag{1.11}$$

In analogy to the gate model (see section 1.1.3), we can, however, easily extent the notion of n qubit unitary operators (see section 1.3.3.1) to work on $n+m$ qubit states, by using the extended operator U_1 (the index indicates that the first register is affected):

$$U_1 = U \times I(m) = \sum_{i,j=0}^{2^n-1} \sum_{k=0}^{2^m-1} u_{ij} \, |i,k\rangle\langle j,k| \tag{1.12}$$

The first n qubits of the $n+m$ qubit state $|\psi\rangle$ are referred to as an n qubit *quantum register* relative to the operator U.

$$\begin{aligned} U_1 \, |\psi\rangle = \sum_{i,j=0}^{2^n-1} \sum_{k=0}^{2^m-1} \sum_{i'=0}^{2^n-1} \sum_{j'=0}^{2^m-1} u_{ij} a_{i'j'} \, |i,k\rangle\langle j,k|i',j'\rangle = \\ \sum_{i,j,i'=0}^{2^n-1} \sum_{k,j'=0}^{2^m-1} u_{ij} a_{i'j'} \delta_{ji'} \delta_{k,j'} \, |i,k\rangle = \sum_{i,j=0}^{2^n-1} \sum_{k=0}^{2^m-1} u_{ij} a_{jk} \, |i,k\rangle \end{aligned} \tag{1.13}$$

1.3.2.2 Register Reordering

The concept of quantum registers can be extended to arbitrary sequences of qubits.

Definition 1 (Quantum Register) *An n qubit quantum Register $\mathbf{s}$ is a sequence of mutually different zero-based qubit positions $\langle s_0, s_1 \ldots s_{n-1}\rangle$ of some state $|\psi\rangle \in \mathbf{C}^{2^N}$ with $N \geq n$.*

Let $\mathbf{s}$ be an n qubit register of the $n+m$ qubit state $|\psi\rangle$. Using an arbitrary permutation π over $n+m$ elements with $\pi_i = s_i$ for $i < n$, we can construct a *reordering operator* Π_s by permutating the qubits.

$$\Pi_s \, |b_0, b_1 \ldots b_l\rangle = |b_{\pi_0}, b_{\pi_1} \ldots b_{\pi_l}\rangle \quad \text{with} \quad l = n+m \tag{1.14}$$

Using any Π_s and U_1 we can define the application of U to the register $\mathbf{s}$ of $|\psi\rangle$ as

$$U(\mathbf{s}) \, |\psi\rangle = \Pi_s^\dagger \, U_1 \, \Pi_s \, |\psi\rangle \tag{1.15}$$

Note that despite the fact, that here are $m!$ possible implementations of Π_s, the definition of $U(\mathbf{s})$ is unique:

Definition 2 (Register Operators) *The register operator for an n qubit quantum register $\mathbf{s}$ of the state $|\psi\rangle \in \mathbf{C}^{2^N}$ and a unitary operator $U : \mathbf{C}^{2^n} \to \mathbf{C}^{2^n}$ is the N qubit unitary operator $U(\mathbf{s}) = \Pi_s^\dagger \, (U \times I(m)) \, \Pi_s$.*

1.3.3 Functions

1.3.3.1 Pseudo-classic Operators

The general form of a unitary operator U over n qubits is

$$U = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} |i\rangle \, u_{ij} \, \langle j| \quad \text{with} \quad \sum_{k=0}^{2^n-1} u_{ki}^* u_{kj} = \delta_{ij} \tag{1.16}$$

If the matrix elements u_{ij} are of the form $u_{ij} = \delta_{i\pi_j}$ with some permutation π, then their effect on pure states (base-vectors) can be described in terms of classical reversible boolean logic.

Definition 3 (Pseudo-classic Operator) *A pseudo-classic operator is a unitary operator of the form* $U : |i\rangle \to |\pi_i\rangle$*.*

Let $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^n}$ be a bijective function, then the corresponding pseudo-classic operator F is given as

$$F = \sum_{i=0}^{2^n-1} |f(i)\rangle\langle i| \quad \text{and} \quad F^{-1} = F^\dagger = \sum_{i=0}^{2^n-1} |i\rangle\langle f(i)| \tag{1.17}$$

1.3.3.2 Handling of Non-Reversible Functions

One obvious problem of quantum computing is its restriction to reversible computations. Consider a simple arithmetical operation like integer division by 2 ($DIV2'|i\rangle = |i/2\rangle$ for even i and $|(i-1)/2\rangle$ for odd i). Clearly, this operation is non-reversible since $DIV2'|0\rangle = DIV2'|1\rangle$, so no corresponding pseudo-classic operator exists.

However, if we use a second register with the initial value $|0\rangle$, then we can define an operator $DIV2$ witch matches the condition $DIV2|x, 0\rangle = |x, x/2\rangle$ or $|x, (x-1)/2\rangle$ respectively. The behaviour of $DIV2|x, y \neq 0\rangle$ is undefined and can be set arbitrarily under the condition that $DIV2$ is unitary[6].

Definition 4 (Quantum Functions) *For any function* $f : \mathbf{B}^n \to \mathbf{B}^m$ *(or equivalently* $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^m}$*) there exists a pseudo-classic operator* $F : \mathbf{C}^{2^{n+m}} \to \mathbf{C}^{2^{n+m}}$ *working on an* n*-qubits input and an* m*-qubits output register with* $F|x, 0\rangle = |x, f(x)\rangle$*. Operators of that kind shall be called quantum functions.*

[6] In this special case, just one additional qubit to hold the lowest bit of the argument would suffice to extend $DIV2'$ to a unitary operator.

1.3.4 Scratch Space Management

1.3.4.1 Register Reuse

While keeping a copy of the argument will allow us to compute non reversible functions, this also forces us to provide extra storage for intermediate results. In longer calculations this would leave us with a steadily increasing amount of "junk" bits which are of no concern for the final result.

A simple and elegant solution of this problem was proposed by Bennet [8, 9]: If a composition of two non-reversible functions $f(x) = h(g(x))$ is to be computed, the scratch space for the intermediate result can be "recycled" using the following procedure (G and H are the quantum functions for g and h, the indices indicate the registers operated on):

$$|x, 0, 0\rangle \xrightarrow{G_{12}} |x, g(x), 0\rangle \xrightarrow{H_{23}} |x, g(x), h(g(x))\rangle \xrightarrow{G_{12}^\dagger} |x, 0, f(x)\rangle \tag{1.18}$$

The last step is merely the inversion of the first step and uncomputes the intermediate result. The second register can then be reused for further computations.

Without scratch-management, the evaluation of a composition of depth d needs d operations and consumes $d - 1$ junk registers. Bennet's method of uncomputing can then be used to trade space against time: Totally uncomputing of all intermediate results needs $2d - 1$ operations and $d - 1$ scratch registers, which is useful, if the scratch can be reused in the further computation.

By a combined use of r registers as scratch and junk space, a composition of depth $d = (r + 2)(r + 1)/2$ can be evaluated with $2d - r - 1 = (r + 1)^2$ operations. An calculation of $f(x) = l(k(j(i(h(g(x))))))$ on a 4-register machine (1 input, 1 output and 2 scratch/junk registers) would run as follows (function values are in prefix notation):

$$|x, 0, 0, 0\rangle \xrightarrow{I_{34}H_{23}G_{12}} |x, gx, hgx, ihgx\rangle \xrightarrow{G_{12}^\dagger H_{23}^\dagger} |x, 0, 0, ihgx\rangle \xrightarrow{J_{42}^\dagger K_{23} J_{42}} \tag{1.19}$$

$$|x, 0, kjihgx, ihgx\rangle \xrightarrow{L_{32}} |x, lkjihgx, kjihgx, ihgx\rangle = |x, fx, kjihgx, ihgx\rangle$$

By using this method, we can reduce the needed space by $O(1/\sqrt{d})$ with a computation overhead of $O(2)$.

1.3.4.2 Junk Registers

If the computation of a function $f(x)$ fills a scratch register with the junk bits $j(x)$ (i.e. $|x, 0, 0\rangle \to |x, f(x), j(x)\rangle$), a similar procedure can free the register again:

$$|x,0,0,0\rangle \stackrel{F_{123}}{\longrightarrow} |x,f(x),j(x),0\rangle \stackrel{FANOUT_{24}}{\longrightarrow} |x,f(x),j(x),f(x)\rangle \stackrel{F_{123}^{\dagger}}{\longrightarrow} |x,0,0,f(x)\rangle \tag{1.20}$$

Again, the last step is the inversion of the first. The intermediate step is a *FANOUT* operation which copies the function result into an additional empty (i.e. in sub-state $|0\rangle$) register. Possible implementations are e.g.

$$FANOUT : |x,y\rangle \to |x, x \oplus y\rangle \quad \text{or} \quad |x,(x+y) \bmod 2^n\rangle \tag{1.21}$$

1.3.4.3 Overwriting Invertible Functions

As pointed out in section 1.3.3.1, every invertible function $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^n}$ has a corresponding pseudo classic operator $F : |i\rangle \to |f(i)\rangle$. While a direct implementation of F is possible with any complete set of pseudo-classic operators[7], the implementation as a quantum function can be substantially more efficient.

If we have efficient implementations of the quantum functions $U_f : |i,0\rangle \to |i,f(i)\rangle$ and $U_{f^{-1}} : |i,0\rangle \to |i,f^{-1}(i)\rangle$, then an overwriting operator F' can be constructed by using an n qubit scratch register.

$$F' : |i,0\rangle \stackrel{U_f}{\longrightarrow} |i,f(i)\rangle \stackrel{SWAP}{\longrightarrow} |f(i),i\rangle \stackrel{U_{f^{-1}}^{\dagger}}{\longrightarrow} |f(i),0\rangle \tag{1.22}$$

1.3.5 Conditional Operators

Classical programs allow the conditional execution of code in dependence on the content of a boolean variable (conditional branching).

A unitary operator, on the other hand, is static and has no internal flow-control.[8] Nevertheless, we can conditionally apply an n qubit operator U to a quantum register by using an *enable* qubit and define an $n+1$ qubit operator U'

$$U' = \begin{pmatrix} I(n) & 0 \\ 0 & U \end{pmatrix} \tag{1.23}$$

So U is only applied to base-vectors where the enable bit is set. This can be easily extended to enable-registers of arbitrary length.

[7]One example would be the Toffoli gate $T : |x,y,z\rangle \to |x \oplus (y \wedge z), y, z\rangle$ which can be used to implement any pseudo-classic operator for 3 or more qubits

[8]One might argue, that a QTM does in fact have internal flow control because head state and head position are quantum registers. Here, however, flow-control refers to the dynamic generation of a gate sequence, see section 1.1.3

Definition 5 (Conditional Operator) *A conditional operator $U_{[[\mathbf{e}]]}$ with the enable register* $\mathbf{e}$ *is a unitary operator of the form*

$$U_{[[\mathbf{e}]]} : |i, \epsilon\rangle = |i\rangle|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} (U\,|i\rangle)\,|\epsilon\rangle_{\mathbf{e}} & \textit{if}\ \epsilon = 111\ldots \\ |i\rangle|\epsilon\rangle_{\mathbf{e}} & \textit{otherwise} \end{cases} \tag{1.24}$$

Conditional operators a frequently used in arithmetic quantum functions and other pseudo-classic operators.

If the architecture allows the efficient implementation of the *controlled-not* gate $C : |x, y_1, y_2 \ldots\rangle \to |(x \oplus \bigwedge_i y_i), y_1, y_2 \ldots\rangle$, then conditional pseudo-classic operators can be realised by simply adding the enable string to the control register of all controlled-not operations.

Chapter 2

QCL – A Quantum Computation Language

2.1 Introducing QCL

2.1.1 Features

As pointed out in section 1.1.4, QCL is a high level language for quantum programming. Its main features are:

- a classical control language with functions, flow-control, interactive i/o and various classical data types (`int`, `real`, `complex`, `boolean`, `string`)
- 2 quantum operator types: general unitarian (`operator`) and reversible pseudo-classic gates (`qufunct`)
- inverse execution, allowing for on-the-fly determination of the inverse operator though caching of operator calls
- various quantum data types (qubit registers) for compile time information on access modes (qureg, quconst, quvoid, quscratch)
- convenient functions to manipulate quantum registers (`q[n]` - qubit, `q[n:m]` - substring, `q&p` - combined register
- Quantum memory management (quheap) allowing for local quantum variables
- Transparent integration of Bennet-style scratch space management

- Easy adaption to individual sets of elementary operators

The interpreter `qcl` additionally integrates a numeric simulator and a shell environment for interactive use.

2.1.2 Example: Discrete Fourier Transform in QCL

Table 2.1 shows the QCL implementation of Coppersmith's algorithm [7] of fast quantum discrete Fourier Transform for quantum registers of arbitrary length (see section 3.2.3 for details).

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

Table 2.1: **dft.qcl** Discrete Fourier Transform in QCL

Basically, `dft.qcl` contains of two loops: The outer loop performs a Hadamard transformations (see section 3.1.1.3) from the highest to the lowest qubit (`Mix`), while the inner loop performs conditional phase shifts (`CPhase`) between the qubits.

The `dft` operator takes a quantum register (`qureg`) `q` as argument. As pointed out in section 1.3.2, quantum register is not a quantum state by itself, but a pointer indicating the target qubits in the overall machine state, just like input and output lines in the gate model. To allow register size independent operator definitions, the number of qubits of a register can be determined at run time by the size operator `#`.

Inside the operator definition, sub-operators can be called just as sub-procedures in classic procedural languages. This means that the actual sequence of operators (either built in or user defined) can be fully determined at runtime, including the use of loops (in this case `for`-loops), conditional statements, etc.

Other then most classic languages, QCL has a strict mathematical semantics of functions and operators, meaning that two subsequent calls with the same parameters have to result in exactly the same operation. This requires the operators to be free from side-effects and forbids the use of global variables.

This allows for context dependent execution: `DFT(q)` called from toplevel works a expected, however if called as `!DFT(q)` (adjungation, thus inversion for unitary operators), all operators within `DFT` are also inverted and applied in reverse order. Inverse execution can also take place implicitly, when local quantum registers and Bennet-style scratch-space management is used.

To demonstrate the use of the new operator, let's start the QCL interpreter and prepare a test state:

```
$ qcl -b5 -i dft.qcl
[0/5] 1 |00000>
qcl> qureg q[5];          // allocate a 5 qubit register
qcl> Mix(q[3:4]);         // rotate qubits 3 and 4
[5/5] 0.5 |00000> + 0.5 |10000> + 0.5 |01000> + 0.5 |11000>
qcl> Not(q[0]);           // invert first qubit
[5/5] 0.5 |00001> + 0.5 |10001> + 0.5 |01001> + 0.5 |11001>
```

We now have a periodic state with period $2^3 = 8$ and an offset of 1 composed of 4 basevectors to which we can apply the DFT.

```
qcl> dft(q);
[5/5] 0.353553 |00000> + -0.353553 |10000> + 0.353553i |01000> +
-0.353553i |11000> + (0.25,0.25) |00100> + (-0.25,-0.25) |10100> +
(-0.25,0.25) |01100> + (0.25,-0.25) |11100>
```

The discrete Fourier transform "inverts" the period to $2^5/8 = 4$ and results in a periodic distribution with offset 0. The information about the original offset is in the phase factors, and has no influence on the probability distribution:

```
qcl> dump q;
: SPECTRUM q: |43210>
0.125 |00000> + 0.125 |00100> + 0.125 |01000> + 0.125 |01100> +
0.125 |10000> + 0.125 |10100> + 0.125 |11000> + 0.125 |11100>
```

"Uncomputing" the DFT brings back the initial configuration:

```
qcl> !dft(q);
[5/5] 0.5 |00001> + 0.5 |10001> + 0.5 |01001> + 0.5 |11001>
qcl> exit;
```

2.1.3 The QCL Interpreter

The interpreter `qcl` simulates a quantum computer with an arbitrary number of qubits (default is the system word length) and is called with the following syntax:

`qcl` [*options*] [*QCL-file*] . . .

2.1.3.1 Options

`qcl` takes the following command-line options (defaults are given in parantheses):

```
Startup Options:
-h, --help                      display this message
-V, --version                   display version information
-i, --interactive               force interactive mode
-n, --no-default-include        don't read default.qcl on startup
-o, --logfile                   specify a logfile
-b, --bits=n:                   set number of qubits (32)
Dynamic Options (can be changed with the set statement):
-s, --seed=<seed-value>         set random seed value (system time)
-I, --include-path=<path>       QCL include path (.)
-p, --dump-prefix=<file-prefix> set dump-file prefix
-f, --dump-format=b,a,x         list base vectors as hex, auto or binary (a)
-d, --debug=<y|n>               open debug-shell on error (n)
-a, --auto-dump=<y|n>           allways dump quantum state in shell mode (n)
-l, --log==<y|n>                log external operator calls (n)
-L, --log-state==<y|n>          log state after each transformation (n)
-T, --trace==<y|n>              trace mode (very verbose) (n)
-S, --syntax=<y|n>              check only the syntax, no interpretation (n)
-E, --echo=<y|n>                echo parsed input (n)
-t, --test=<y|n>                test programm, ignore quantum operations (n)
-e, --shell-escape=<y|n>        honor shell-escapes (y)
-r, --allow-redefines=<y|n>     ignore redefines instead of error (n)
```

Logical values can be given in any common format including `y[es]/n[o]`, `0/1` and `t[rue]/f[alse]`. Dynamic options can also be invoked from the shell or by QCL-Programs via the `set` command.

2.1.3.2 Default Include

Unless disabled with `--no-default-include`, `qcl` interprets the default include file `default.qcl` at startup. If no files are given at the command line, `qcl` starts in interactive mode, otherwise the files are executed in the given order and `qcl` exits.

2.1.3.3 Interactive Use

If started in interactive mode (no files given or option `-i`), `qcl` enters the top level shell and prompts for user input (`qcl>`). Subshells with private scope (see section 2.2.4.4) can be opened by the command `shell`; the prompt in this case is `qcl`n`>`, where n indicates the zero-based nesting depth. Subshells are also opened in case of errors when `--debug=yes` and prompt with `qcl`n`$`.

```
qcl> set debug true;     // turn on debuging
qcl> real inv(real x) { return 1/x; }
qcl> print inv(0.0);     // trigger an error
! in function inv: math error: division by zero
qcl1$ list x;            // this is the argument to fac
: local symbol x = 0.000000:
real x
qcl1$ exit;              // close the debug shell
qcl> list x;             // no global x defined
: symbol x is undefined.
```

A shell is closed by `EOF` (usually `Ctrl-d`) or by the `exit` command. Closing the top level shell terminates the program.

2.1.4 Structure of a QCL Program

2.1.4.1 Notation

The syntactic structure of a QCL program is described by a context free LALR(1) grammar (see appendix B.1). For the formal definition of syntactic expressions, the following notation is used:

$$\begin{array}{rcl} \textit{expression-name} & \leftarrow & \textit{expression-def}_1 \\ & \leftarrow & \textit{expression-def}_2 \\ & \dots & \dots \end{array}$$

Within expression definitions, the following conventions apply

Keywords and other litteral text is set in `courier`

Subexpressions are set in *italic*

Optional expressions are put in [square brakets] Optional expressions can be repeated 0 or 1 times.

Multiple expressions are put in { braces }. Multiple expression can be repeated 0, 1 or n times.

Alternatives are written as $alt_1 | \, alt_2 | \dots$ Exactly one alternative has to be chosen.

Grouping of expressions can be forced by (round brackets).

To simplify the notation of literals, the following character classes are defined:

- *digit* $\leftarrow$ decimal digit from `0` to `9`.
- *letter* $\leftarrow$ alphabetic letter form `a` to `z` or `A` to `Z`. Case is significant.
- *char* $\leftarrow$ printable character except '`"`'.

A QCL Program is basically a sequence of *statements* and *definitions* either read from a file or directly from the shell prompt. (In the latter case, input is restricted to one line which is implicitly terminated by '`;`'.)

$$\textit{qcl-input} \leftarrow \{\, \textit{stmt} \,|\, \textit{def} \}$$

2.1.4.2 Statements

Statements range from simple commands, over procedure-calls to complex control-structures and are executed when they are encountered.

```
qcl> if random()>=0.5 { print "red"; } else { print "black"; }
: red
```

2.1.4.3 Definitions

Definitions are not executed but bind a value (variable- or constant-definition) or a block of code (routine-definition) to a *symbol* (identifier).

```
qcl> int counter=5;
qcl> int fac(int n) { if n<=0 {return 1;} else {return n*fac(n-1);} }
```

Consequently, each symbol has an associated *type*, which can either be a *data type* or a *routine type* and defines whether the symbol is accessed by reference or call.

2.1.4.4 Expressions

Many statements and routines take arguments of certain data types. These *expressions* can be composed of *literals*, variable references and sub-expressions combined by operators and function calls.

```
qcl> print "5 out of 10:",fac(10)/fac(5)^2,"combinations."
: 5 out of 10: 252 combinations.
```

2.1.4.5 Comments

As C++, QCL supports two ways of commenting code. All comments are simply discarded by the scanner.

Line comments are started with `//` and last until the end of the current line

C-style comments are started with `/*` and terminated with `*/` and may continue over several lines. C-style comments may not be nested.

2.1.4.6 Include Files

The command `include "`*filename*`"` tells the interpreter, to process the file *filename*`.qcl`, before continuing with the current input file or command line. `qcl` looks for the file in the current directory and in the default include directory, which can be changed with the option `include-path`.

In interactive use `include "`*filename*`"` can be abbreviated by `<<`*filename*.

2.2 Classic Expressions and Variables

2.2.1 Constant Expressions

The classic data-types of QCL are the arithmetic types `int`, `real` and `complex` and the general types `boolean` and `string`.

Type	Description	Examples
`int`	integer	`1234, -1`
`real`	real number	`3.14, -0.001`
`complex`	complex number	`(0,-1), (0.5, 0.866)`
`boolean`	logic value	`true, false`
`string`	character string	`"hello world", ""`

Table 2.2: classic types and literals

2.2.1.1 Integer

The QCL type `int` is based on the C data type `singed long`. On a 32bit computer, this typically allows values form -2^{31} to $2^{31}-1$. While this is more than enough for any standard programming task, the implementation of certain combinatoric functions can require some attention.

However, the word-length limitation has no effect on the number of qubits which can be simulated or on the maximum size of quantum registers. [1]

2.2.1.2 Real

The QCL type `real` is based on C data type `double`, which is typically 64bit. To improve readability, output of real numbers is truncated to a reasonable number of digits.

Real numbers are written in decimal notation and must include the decimal point.

2.2.1.3 Complex

QCL complex numbers are internally represented as two `double` floats. This holds for variables of type `complex` as well as the internal representation of the machine state. Complex numbers are given as $(real, imag)$-pairs:

$$\begin{array}{rcl} \textit{const} & \leftarrow & \texttt{(}\ \textit{complex-coord}\ \texttt{,}\ \textit{complex-coord}\ \texttt{)} \\ \textit{complex-coord} & \leftarrow & [\,\texttt{+}\,|\,\texttt{-}\,]\ \textit{digit}\ \{\,\textit{digit}\,\}\ [\ \texttt{.}\ \{\,\textit{digit}\,\}] \end{array}$$

2.2.1.4 Boolean

Unlike C, logical values have their own data type in QCL. Literals for `boolean` are `true` and `false`:

2.2.1.5 Strings

String literals are quoted with `"`*character string*`"` and may contain any printable character except '`"`'.

2.2.2 Operators

2.2.2.1 Arithmetic Operators

Arithmetic operators generally work on all arithmetic data types and return the most general type (operator overloading), e.g.

```
qcl> print 2+2;          // evaluates to int
: 4
qcl> print 2+2.0;        // evaluates to real
: 4.000000
qcl> print 2+(2,0);      // evaluates to complex
: (4.000000,0.000000)
```

[1]As quantum measurements also return integers, measurements must be split for larger registers

To allow for clean integer arithmetic there are some exceptions to avoid typecasts:

- The division operator `/` does integer division if both arguments are integer.
- The modulus operator `mod` is only defined for integer arguments.
- The power operator `^` for integer bases is only defined for non-negative, integer exponents. For real exponents, the base must be non-negative.

table 2.3 shows all arithmetic operators ordered from high to low precedence. All binary operators are left associative, thus $a \circ b \circ c = (a \circ b) \circ c$. Explicit grouping can be achieved by using parentheses.

Op	**Description**	**Example**	**Type**	**Value**
`^`	power	`(0,1)^-1.5`	`complex`	$-\frac{1}{\sqrt{2}}(1+i)$
	integer power	`(-2)^11`	`int`	-2048
`-`	unary minus	`--1`	`int`	1
`*`	multiplication	`(0,1)*(0,1)`	`complex`	-1
`/`	division	`3./2`	`real`	$\frac{3}{2}$
	integer division	`3/2`	`int`	1
`mod`	integer modulus	`100 mod 16`	`int`	4
`+`	addition	`1.5+1.5`	`real`	3
`-`	subtraction	`(1,2)-(0,2)`	`complex`	1

Table 2.3: arithmetic operators

2.2.2.2 Comparison and Logic Operators

Table 2.4 shows all comparison and logic operators with their argument types. The return type of all operators is `boolean`.

2.2.2.3 Other Operators

QCL defines two more operators, which are mainly used with quantum expressions. They are described in section 2.3.3 and mentioned her only for completeness.

Concatenation The concatenation operator `&` combines two quantum registers or two strings. Its precedence is equal to the arithmetic operators `+` and `-`.

Op	Description	Argument type
`==`	equal	all arithmetic, `string`
`!=`	unequal	all arithmetic, `string`
`<`	less	`integer`, `real`
`<=`	less or equal	`integer`, `real`
`>`	greater	`integer`, `real`
`>=`	greater or equal	`integer`, `real`
`not`	logic not	`boolean`
`and`	logic and	`boolean`
`or`	logic inclusive or	`boolean`
`xor`	logic exclusive or	`boolean`

Table 2.4: comparison and logic operators

Size The size-of operator `#` gives the length (i.e. the number of qubits) of any quantum expression. This is the operator with the highest precedence.

2.2.3 Functions

Unlike user defined functions (see section 2.5.2), most of QCL's built-in functions are overloaded. Like arithmetic operators, they often accept more then one argument type and evaluate to different return types.

2.2.3.1 Trigonometric Functions

Trigonometric functions (table 2.5) are defined for `int`, `real` and `complex` arguments. There return type is `real` or `complex`, depending on the argument type.

```
qcl> print sin(0);           // integer or real arguments
: 0.000000                   // evaluate to real, even if the
qcl> print sin(pi);          // result is an integer number
: 0.000000
qcl> print tanh((0,1)*pi);   // complex arguments evaluate
: (0.000000,-0.000000)       // to complex
```

2.2.3.2 Exponents and Logarithms

`exp`, `log` and `sqrt` also work on all arithmetic types (table 2.6). Square roots of negative real numbers trigger an error, as do non positive real logarithms.

Funct.	Description	Funct.	Description
`sin(x)`	sine of x	`sinh(x)`	hyperbolic sine of x
`cos(x)`	cosine of x	`cosh(x)`	hyperbolic cosine of x
`tan(x)`	tangent of x	`tanh(x)`	hyperbolic tangent of x
`cot(x)`	cotangent of x	`coth(x)`	hyperbolic cotangent of x

Table 2.5: trigonometric and hyperbolic functions

```
qcl> print sqrt(-1);
! math error: real square root of negative number
qcl>  print log(-1.0);
! math error: real logarithm of non positive number
qcl> print sqrt((-1,0)),log((-1,0));
: (0.000000,1.000000) (0.000000,3.141593)
```

Funct.	Description
`exp(x)`	e raised to the power of x
`log(x)`	natural logarithm of x
`log(x,n)`	base-n logarithm of x
`sqrt(x)`	square root of x

Table 2.6: exponential and related functions

2.2.3.3 Complex Numbers

For handling and conversion of complex expressions, the functions `Re`, `Im`, `abs` and `conj` are defined (table 2.7). `abs` also works on `real` and `int` arguments.

Funct.	Description
`Re(z)`	real part of z
`Im(z)`	imaginary part of z
`abs(z)`	magnitude of z
`conj(z)`	complex conjugate of z

Table 2.7: functions for complex numbers

2.2.3.4 Rounding

`ceil(x)` and `floor(x)` round the real value x up- or downwards to the nearest integer. The rounded value is returned as `int`.

2.2.3.5 Maximum and Minimum

The functions `max` and `min` take an arbitrary number of `int` or `real` arguments and return the maximum or minimum value. The return type is `int`, if all arguments are integer, and `real` otherwise.

```
qcl> print max(3,pi,4);
: 4.000000
```

2.2.3.6 GCD and LCM

The greatest common divisor and the least common multiple of a list of integers can be determined by `gcd` and `lcm`.

```
qcl> print gcd(48,72,180),lcm(48,72,180);
: 12 720
```

2.2.3.7 Random Numbers

The pseudo function `random()` returns a random value from the interval $[0,1)$. The generation of random numbers can be determined by providing a seed value with the option `--seed`. Random numbers cannot be used in the definition of functions and quantum operators.

Funct.	Description
`ceil(x)`	nearest integer to x (rounded upwards)
`floor(x)`	nearest integer to x (rounded downward)
`max(x,...)`	maximum
`min(x,...)`	minimum
`gcd(n,...)`	greatest common divisor
`lcm(n,...)`	least common multiple
`random()`	random value from $[0,1)$

Table 2.8: other QCL functions

2.2.4 Symbols

2.2.4.1 Identifiers

Identifiers are restricted to alphanumeric characters (no underscores) and must begin with a letter. As in C, there is no limit in length and case is significant. Names must not collide with internal keywords.

2.2.4.2 Constants

Frequently used values can be defined as symbolic constants. The syntax of a constant declaration is

$$\textit{const-def} \leftarrow \texttt{const}\ \textit{identifier}\ \texttt{=}\ \textit{expr}\ \texttt{;}$$

The definition of `pi` in the standard include file is e.g.

```
const pi=3.141592653589793238462643383279502884197;
```

Constant definitions are not restricted to constant expressions, but keep their value, once defined:

```
qcl> const seed=random();
qcl> print seed;
: 0.378990
qcl> print seed;
: 0.378990
```

2.2.4.3 Variables

The definition of variables in QCL is analogous to C:

$$\textit{var-def} \leftarrow \textit{type}\ \textit{identifier}\ [\,\texttt{=}\ \textit{expr}\,]\ \texttt{;}$$

Classic data types are `int`, `real`, `complex`, `boolean` and `string` (see section 2.2.1). If no initial value is given, the new variable is initialised with zero, `false` or `""`, respectively.

The value of a variable can be changed by an *assignment*, as well as several other statements (see section 2.4):

```
qcl> complex z;          // declare complex variable z
qcl> print z;            // z was initialised with 0
: (0.000000,0.000000)
qcl> z=(0,1);            // setting z to i
qcl> print z;
: (0.000000,1.000000)
qcl> z=exp(z*pi);        // assignment to z may contain z
qcl> print z;
: (-1.000000,0.000000)
qcl> input z;            // ask for user input
? complex z [(Re,Im)] ? (0.8,0.6)
qcl> print z;
: (0.800000,0.600000)
```

Since the value of variables is by definition not unique, global variables cannot be accessed in routines with mathematical semantics, namely functions and (pseudo-classic) operators.

2.2.4.4 Scopes and Namespaces

All Symbols share the same namespace, therefore all global identifiers have to be unique, even if they designate different objects. To guarantee consistent behaviour of defined functions and operators, there is no way to undefine a once declared global symbol.

```
qcl> int f;
qcl> int f(int n) { return f^2; }
! illegal scope: Global symbol f already defined
```

The option `--allow-redefines` can be used to suppress errors when a symbol is declared a second time. Redefinitions are then silently ignored and the processing of the input file or line continues. This can be useful for safely including files or shadowing definitions by private versions:

```
qufunct flip(qureg q) { /* define my own version of flip */ }
set allow-redefines true;
include "dft";          // flip-version in dft.qcl is ignored
set allow-redefines false;
```

The definition of a symbol can be shown with the `list` command (see below for an example).

When using `qcl` interactively (see section 2.1.3.3), the `shell` command can be used to open a subshell with a temporal scope. This means, that new definitions are only valid within the current subshell.[2] All temporal symbols are destroyed again, when the subshell is left with `exit`.

```
qcl> list x;
: symbol x is undefined.
qcl> shell;
: shell escape
qcl1> int x;
qcl1> list x;
: global symbol x = 0:
int x;
qcl1> exit
qcl> list x;
: symbol x is undefined.
```

[2]Note that this doesn't compromise mathematical semantics of functions and operators, since the existence of used global symbols is checked already when the routine is defined and not just at execution-time.

2.3 Quantum Registers and Expressions

2.3.1 Registers and States

2.3.1.1 Machine State and Program State

The memory of a quantum computer is usually a combination of 2-state subsystems, referred to as quantum bits (qubits). As pointed out in section 1.2.2 the "memory content" is the combined state of all qubits. This state is referred to as the (quantum) *machine state* as opposed to the *program state* which is the current state of the controlling (classic) algorithm (e.g. contents of variable, execution stack, etc.) [3]

The machine state $|\Psi\rangle$ of an n qubit quantum computer is a vector in the Hilbert space $\mathcal{H} = \mathbf{C}^{2^n}$, however – due to the destructive nature of measurement (see section 1.2.5) – $|\Psi\rangle$ cannot be directly observed.

2.3.1.2 Quantum Registers

QCL uses the concept of quantum registers (see section 1.3.2) as an interface between the machine state and the controlling algorithm. A quantum register is a pointer to a sequence of (mutually different) qubits and all operations on the machine state (except for the `reset` command, see section 2.4.2.2) take quantum registers as arguments.

Since an n qubit quantum computer allows for $\frac{n!}{(n-m)!}$ different m qubit registers, any unitary or measurement operation on a m qubit register, can result in $\frac{n!}{(n-m)!}$ different operations on the machine state:

Let $\mathbf{s}$ be an m qubit register covering the qubits at the zero-based positions $\langle s_0, s_1 \ldots s_{m-1}\rangle$ (with $s_i \neq s_j$ for all $i, j < m$) of the n qubit machine state $|\Psi\rangle$ and Op be a m qubit unitary or measurement operator, then the register operation $Op(\mathbf{s})$ corresponds to the following machine state operation (see section 1.3.2.2):

$$Op(\mathbf{s}) : |\Psi\rangle \to R_s^\dagger \left(Op \times I(n-m)\right) R_s \,|\Psi\rangle \tag{2.1}$$

The reordering operator R_s and the k-qubit identity operator $I(k)$ are defined in (1.15) and (1.2), respectively.

2.3.1.3 Memory Management

In QCL, the relation between registers and qubits is handled transparently by allocation and deallocation from qubits of the *quantum heap*, which allows the

[3] Note that QTM has no program state in this sense, since it isn't subject to any classic meta-algorithm

use of local quantum variables. All free (i.e. unallocated) quantum memory has to be *empty*.

Definition 6 (Empty Registers) *A quantum register* `s` *is considered empty if*

$$P_0(\mathtt{s})\,|\Psi\rangle = |\Psi\rangle \quad \text{with} \quad P_0 = |0\rangle\langle 0| \tag{2.2}$$

At startup or after the `reset` command, the whole machine state is empty, thus $|\Psi\rangle = |0\rangle$.

Pseudo-classic operators (`qufunct`) allow the use of local quantum variables as scratch space (see section 1.3.4). When temporary scratch registers (`quscratch`) are allocated, memory management has to keep track of all applied operators until the local register is deallocated again. Then the result registers are saved using *FANOUT* and the computation is run in reverse.

The user can override default memory management by explicitly exclude local variables form uncomputing by declaring them as general registers `qureg` (see section 2.3.2.1). In this case, proper cleanup is in the responsibility of the programmer.

2.3.1.4 Simulation

While QCL – as a programming language – is designed to work with any qubit-based quantum computer architecture, the development of the necessary hardware will most probably take a few more decades. This is why QCL also supports the simulation of quantum computers and provides special commands for accessing the (simulated) machine state.

The interpreter `qcl` can simulate quantum computers with arbitrary numbers of qubits (option `--bits`). Only base-vectors with a nonzero amplitude are actually stored, so the use of scratch registers doesn't require additional memory.

All numerical simulations are handled by the QC library. Please refer to [17] for a more detailed description.

2.3.2 Quantum Variables

Quantum registers bound to a symbolic name are referred to as *quantum variables*.

2.3.2.1 General Registers

A general quantum Register with $n =$ *expr* qubits can be declared with

var-def ← `qureg` *identifier* `[` *expr* `]` `;`

Empty quantum memory is allocated from the heap and bound to the symbol *identifier*.

A declaration in global scope defines a permanent quantum register which is not to prone to scratch space management. This means that – as with classic global variables – there is no way to reclaim allocated qubits within the same shell.

If a global register is defined in a subshell (see section 2.2.4.4) and the subshell is closed, the symbol is destroyed and the allocated qubits are again marked as free. It is up to the programmer to guarantee that the deallocated qubits are in fact empty.[4]

```
qcl> qureg q[1];              // allocate a qubit
qcl> Rot(-pi/2,q);            // perform single bit rotation
[1/4] 0.707107 |0000> + 0.707107 |0001>
qcl> shell;                   // open subshell
: shell escape
qcl1> qureg p[1];             // allocate another qubit
qcl1> Rot(-pi/2,p);           // also rotate register p
[2/4] 0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
qcl1> exit;                   // leave subshell
qcl> dump;                    // former register p is not empty
: STATE: 1 / 4 qubits allocated, 3 / 4 qubits free
0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
qcl> list p;                  // however symbol p is undefined
: symbol p is undefined.
```

The reseting of the machine state with the `reset` command has no effect on register bindings.

```
[0/4] 1 |0000>
qcl> qureg q[1];              // allocate a qubit
qcl> reset;                   // reset: |Psi> -> |0>
[1/4] 1 |0000>
qcl> list q;                  // register q still exists
: global symbol q = |...0>:
qureg q[1];
```

The quantum types `quvoid` and `quscratch` are restricted to pseudo-classic operators (`qufunct`) and are equivalent to `qureg`, except that they are treated differently by memory management (see section 2.5.6.3 for details).

2.3.2.2 Quantum Constants

Registers can be declared constant, by using the register type `quconst`. A quantum constant has to be invariant to all applied operators.

[4] Note that proper uncomputation is only possible if no non-reversible operations as measurements have been performed since the allocation.

Definition 7 (Invariance of Registers) *A quantum register* **c** *is considered invariant to a register operator* $U(\mathbf{s},\mathbf{c})$ *if* U *meets the condition*

$$U:\;|i,j\rangle = |i\rangle_\mathbf{s}|j\rangle_\mathbf{c} \to (U_j\,|i\rangle_\mathbf{s})\,|j\rangle_\mathbf{c} \tag{2.3}$$

Quantum constants have a fixed probability spectrum: Let $|\Psi\rangle = \sum a_{ij}|i,j\rangle$ be the machine state and $|\Psi'\rangle = U(\mathbf{s},\mathbf{c})\,|\Psi\rangle$ and $p(J)$ and $p'(J)$ the probabilities to measure J in register **c** before and after the operator is applied, then

$$\begin{aligned} p(J) &= \langle\Psi|P_J|\Psi\rangle = \sum_i a^*_{iJ}a_{iJ} \quad \text{with} \quad P_J = \sum_k |k,J\rangle\langle k,J| & (2.4)\\ p'(J) &= \langle\Psi'|P_J|\Psi'\rangle = \langle\Psi|U^\dagger P_J U|\Psi\rangle = & (2.5)\\ &= \sum_{i',j',i,j} a^*_{i'j'}a_{ij}\;(\langle i'|_\mathbf{s}U^\dagger_{j'}\,\langle j'|_\mathbf{c})\;P_J\;(U_j\,|i\rangle_\mathbf{s}\,|j\rangle_\mathbf{c}) = \\ &= \sum_{i',i} a^*_{i'J}a_{iJ}\;\langle i|U^\dagger_J U_J|i\rangle = p(J) \end{aligned}$$

While global registers can be declared as quantum constants, this isn't particularly useful, since there is no way to change the register spectrum and the register will consequently always be empty.

```
qcl> quconst c[1];
qcl> Mix(c);
! parameter mismatch: quconst used as non-const argument to Mix
```

If an argument to an operator is declared as `quconst`, the register has to be invariant to all subsequent operator calls within the operator definition.

```
qcl> operator foo(quconst c) { Rot(pi,c); }
! in operator foo: parameter mismatch: quconst used as non-const
argument to Rot
```

When used as an argument type to a quantum function, constant registers aren't swapped out when local scratch registers are uncomputed (see section 2.5.5).

2.3.2.3 Empty Registers

If an argument **v** to an operator is declared `quvoid`, the quantum register is expected to be empty when the operator is called normally (see section 2.4.1.2), or to be uncomputed if the operator is called inverted. So, depending on the adjungation flag of the operator, the machine state $|\Psi\rangle$ has to conform to either

$$U(\mathbf{v},\ldots):|\Psi\rangle = |0\rangle_\mathbf{v}|\psi\rangle \to |\Psi'\rangle \quad \text{or} \quad U^\dagger(\mathbf{v},\ldots):|\Psi\rangle \to |0\rangle_\mathbf{v}|\psi'\rangle \tag{2.6}$$

This can be checked at runtime with the option `--check`.

```
qcl> qureg q[4];
qcl> qureg p[4];
qcl> set check 1;          // turn on consistency checking
qcl> Rot(pi/100,p[2]);     // slightly rotate one target qubit
[8/8] 0.999877 |00000000> + -0.0157073 |01000000>
qcl> Fanout(q,p);          // p is assumed void
! in qufunct Fanout: memory error: void or scratch register not empty
```

When used as an argument type to a quantum function, void registers are swapped out to a temporary register if local scratch registers are uncomputed.

2.3.2.4 Scratch Registers

As an argument `s` to an operator, registers of type `quscratch` are considered to be explicit scratch registers which have to be empty when the operator is called and have to get uncomputed before the operator terminates, so operator and machine state have to satisfy the condition

$$U(\mathbf{s},\ldots) : |\Psi\rangle = |0\rangle_{\mathbf{s}}|\psi\rangle \to |0\rangle_{\mathbf{s}}|\psi'\rangle = |\Psi'\rangle \tag{2.7}$$

As with `quvoid`, this is verified at runtime if the option `--check` is set.

If a scratch register is defined within the body of a quantum function, Bennet-style uncomputing as introduced in section 1.3.4.2 is used to empty the register again (see section 2.5.5 for a detailed explanation).

Quantum functions using local scratch registers may not take general (`qureg`) registers as arguments.

```
qcl> qufunct nop(qureg q) { quscratch s[1]; }
! invalid type: local scratch registers can't be used with
qureg arguments
```

2.3.2.5 Register References

To conveniently address subregisters or combined registers (see below), quantum expressions can be named by declaring a register reference.

$$\textit{def} \leftarrow \textit{type identifier}\ [\,\texttt{=}\ \textit{expr}\,]\ \texttt{;}$$

The quantum expression *expr* is bound the register *identifier* of the quantum type *type* which can be `qureg` or `quconst`.

```
qcl> qureg q[8];
qcl> qureg oddbits=q[1]&q[3]&q[5]&q[7];
qcl> qureg lowbits=q[0:3];
qcl> list q,oddbits,lowbits;
: global symbol q = |........76543210>:
qureg q[8];
: global symbol oddbits = |........3.2.1.0.>:
qureg oddbits;
: global symbol lowbits = |............3210>:
qureg lowbits;
```

References can also can be used to override typechecking by redeclaring a `quconst` as `qureg`, which can be useful if a constant argument should be temporarily used as scratch space but is restored later. See the implementation of modular addition (`addn`) in section 3.2.2.1 (page 62) for an example.

2.3.3 Quantum Expressions

A quantum expression is an anonymous register reference, which can be used as an operator argument or to declare named references (see above).

Expr.	Description	Register
`a`	reference	$\langle a_0, a_1 \dots a_n \rangle$
`a[i]`	qubit	$\langle a_i \rangle$
`a[i:j]`	substring	$\langle a_i, a_{i+1} \dots a_j \rangle$
`a[i\l]`	substring	$\langle a_i, a_{i+1} \dots a_{i+l-1} \rangle$
`a&b`	concatenation	$\langle a_0, a_1 \dots a_n, b_0, b_1 \dots b_m \rangle$

Table 2.9: quantum expressions

2.3.3.1 Subregisters

Subregisters can be addressed with the subscript operator `[...]`. Depending on the syntax (see table 2.9), Single qubits are specified by their zero-based offset and substrings are specified by the offset of the first qubit and either the offset of the last qubit (syntax `[...:...]`) or the total length of the subregister (syntax `[...\...]`).

```
qcl> qureg q[8];
qcl> print q[3],q[3:4],q[3\4];
: |....0...> |...10...> |.3210...>
```

Indices can be arbitrary expressions of type `int`. Invalid subscripts trigger an error.

```
qcl> int i=255;
qcl> print q[floor(log(i,2))];
: |0.......>
qcl> print q[floor(log(i,2))\2];
! range error: invalid quantum subregister
```

2.3.3.2 Combined Registers

Registers can be combined with the concatenation operator `&`. If the registers overlap, an error is triggered.

```
qcl> print q[4:7]&q[0:3];
: |32107654>
qcl> print q[2]&q[0:3];
! range error: quantum registers overlap
```

2.4 Statements

2.4.1 Elementary Commands

2.4.1.1 Assignment

The value of any classic variable can be set by the assignment operator `=`. The right-hand value must be of the same type as the variable. In contrast to arithmetic operators and built-in functions, no implicit typecasting is performed.

```
qcl> complex z;
qcl> z=pi;          // no typecast
! type mismatch: invalid assignment
qcl> z=conj(pi);    // implicit typecast
```

Since quantum variables are potentially subject to memory management, neither quantum registers nor register references can be reassigned.

```
qcl> qureg q[2];
qcl> quref p=q[0];
qcl> p=q[1];
! invalid type: assignment to quantum variable
```

2.4.1.2 Call

The routine types `procedure`, `operator` and `qufunct` can be called.

$$\textit{stmt} \leftarrow [\ \texttt{!}\ ]\ \textit{identifier}\ \texttt{(}\ [\ \textit{expr}\ \{\ \texttt{,}\ \textit{expr}\}]\ \texttt{)}\ \texttt{;}$$

As with assignments, no typecasting is performed for classical argument types. Quantum registers can be cast from `qureg` to `quconst` but not the other way around.

```
qcl> list CNot;          // The controlled-not operator takes
: global symbol CNot:    // a qureg and a quconst as arguments
extern qufunct CNot(qureg q,quconst c);
qcl> qureg q[2];
qcl> quconst p[2];
qcl> CNot(q[0],q[1]);    // cast from qureg to quconst
qcl> CNot(p[0],p[1]);    // no cast from quconst to qureg
! parameter mismatch: quconst used as non-const argument to CNot
```

The parameter type `quvoid` and the local register type `quscratch` merely give meta-information for memory management and are otherwise treated as `qureg`.

Calls to the operator types `operator` and `qufunct` can be inverted with the adjungation prefix '`!`'. The operator is then normally executed, except that all suboperators within the operator definition are not immediately invoked, but stored in an internal list together with their evaluated parameters.

When the execution is finished, the suboperators are called in reverse order with their adjungation flags inverted.

```
qcl> <<dft
qcl> qureg q[2];        // allocate 2 qubits
qcl> set log 1;         // turn on operator logging
qcl> dft(q);            // perform discrete Fourier transform
@ Rot(real theta=1.570796,qureg q=|..0.>)
@ CPhase(real phi=1.570796,qureg q=|..10>)
@ Rot(real theta=1.570796,qureg q=|...0>)
@ Swap(qureg a=|...0>,qureg b=|..0.>)
[2/4] 0.5 |0000> + -0.5 |0010> + -0.5 |0001> + 0.5 |0011>
qcl> !dft(q);           // inverse Fourier transform
@ !Swap(qureg a=|...0>,qureg b=|..0.>)
@ !Rot(real theta=1.570796,qureg q=|...0>)
@ !CPhase(real phi=1.570796,qureg q=|..10>)
@ !Rot(real theta=1.570796,qureg q=|..0.>)
[2/4] 1 |0000>
```

2.4.1.3 Input and Output

$$\mathit{stmt} \leftarrow \texttt{input}\ [\,\mathit{expr}\,]\ ,\ \mathit{identifier}\ ;$$

The `input` command prompts for user input and assigns the value to the variable *identifier*. Optionally a prompt string *expr* can be given instead of the standard prompt which indicates the type and the name of the variable. Each input line is prepended by a question mark.

```
qcl> real n;
qcl> input "Enter Number of iterations:",n;
? Enter Number of iterations: 1000
```

`input` repeats prompting until a valid input expression is entered.

```
qcl> boolean b;
qcl> input b;
? boolean b [t(rue)/f(alse)] ? yes
? boolean b [t(rue)/f(alse)] ? true
```

Like global variables, the use of `input` statements in the definition of functions and operators is forbidden.

The `print` command takes a comma separated list of expressions and prints them to the console. Each output is prepended by a colon and terminated with newline. Multiple expressions are delimited by space. In the case of quantum expressions, the position of the corresponding qubits in the machine-state is printed.

```
qcl> int i=3; real x=pi; complex z=(0,1); boolean b; qureg q[8];
qcl> print i,x,z,b,q;
: 3 3.141593 (0.000000,1.000000) false |........76543210>
```

In interactive use, the `print` command can be abbreviated with '?'.

2.4.1.4 Debugging

The commands `shell` and `exit` open and close subshells during interactive use. Please refer to section 2.1.3.3 and section 2.2.4.4 for a detailed description.

$$\textit{stmt} \leftarrow \texttt{list}\ [\,\textit{identifier}\ \{\ ,\ \textit{identifier}\}]\ ;$$

If called without arguments, the `list` command prints a list of all defined global and local symbols. When given a list of symbols, their scope, value (if appropriate) and their definition is printed.

```
qcl> <<dft
qcl> list flip,pi;
: global symbol flip:
qufunct flip(qureg q) {
  int i;
  for i = 0 to #q/2-1 {
    Swap(q[i],q[(#q-i)-1]);
  }
}
: global symbol pi = 3.141593:
const pi = 3.141593;
```

The `dump` command can be used to inspect the simulated machine state or the spectrum of quantum registers. Please refer to section 2.4.2.3 for a detailed description.

$$\textit{stmt} \leftarrow \texttt{set}\ \textit{option}\ [\ ,\ \textit{expr}]\ ;$$

The `set` command can be used to temporarily turn on command-line debug options. See section 2.1.3.1 for a complete list of options.

```
extern qufunct CNot(qureg q,quconst c);

qufunct Swap(qureg a,qureg b) {
  int i;
  if #a != #b { exit "Swap: unmatching register sizes"; }
  for i=0 to #a-1 {
    CNot(a[i],b[i]);    // |a,b>       -> |a xor b,b>
    CNot(b[i],a[i]);    // |a xor b,b> -> |a xor b,a>
    CNot(a[i],b[i]);    // |a xor b,a> -> |b,a>
  }
}
```

Table 2.10: **swap.qcl** Custom implementation of Swap

2.4.2 Quantum Statements

2.4.2.1 Unitary Operations

The operators `Fanout` and `Swap` play a major role in QC as the moral equivalent to the elementary `mov` operation in conventional microprocessors.

$$FANOUT : |i, 0\rangle \to |i, i\rangle \tag{2.8}$$

$$SWAP : |i, j\rangle \to |j, i\rangle \tag{2.9}$$

The default implementation of `Fanout` and `Swap` as defined in `default.qcl` declares them as external (i.e. elementary, see section section 3.1) operators:

```
extern qufunct Fanout(qureg a,qureg b);
extern qufunct Swap(qureg a,qureg b);
```

Since QCL doesn't enforce a specific set of elementary operators, the user is free to provide his own implementations. Table 2.10 gives an example of a custom `Swap` operator using controlled-not gates. Even the `Fanout` operation, which is used by internal scratch space management can be replaced if desired (see table 2.13 on page 53 for an example).

For convenience, QCL provides some syntactic sugar for calls to `Fanout` and `Swap`, which can be used instead of the standard syntax:

$$\textit{stmt} \leftarrow \textit{expr}_{\mathtt{a}} (\,\texttt{->} \mid \texttt{<-} \mid \texttt{<->}\,)\ \textit{expr}_{\mathtt{b}}\ ;$$

These are shortcuts for `Fanout(`**a**`,`**b**`)`, `!Fanout(`**a**`,`**b**`)` and `Swap(`**a**`,`**b**`)`.

2.4.2.2 Non-unitary Operations

As pointed out in section 1.1.1, any quantum computation must be composition of initialisations, unitary operators and measurements. A typical probabilistic quantum algorithm usually runs an evaluation loop like this:

```
{
  reset;             // R: |Psi> -> |0>
  myoperator(q);     // U: |0> -> |Psi'>
  measure q,m;       // M: |Psi'> -> |m>
} until ok(m);       // picked the right m ?
```

The `reset` command resets the machine-state $|\Psi\rangle$ to $|0\rangle$, which is also the initial state when `qcl` is started. The quantum heap and the binding of quantum variables are unaffected.

$$stmt \leftarrow \texttt{measure}\ expr\ [\ ,\ identifier]\ ;$$

The `measure` command measures the quantum register *expr* and assigns the measured bit-string to the `int` variable *identifier*. If no variable is given, the value is discarded.

The outcome of the measurement is determined by a random number generator, which – by default – is initialised with the current system time. For reproducable behaviour of the simulation, a seed value can be given with the option `--seed`.

Since `reset` and `measure` operations are irreversible, they must not occur within operator definitions.

2.4.2.3 Simulator Commands

QCL provides several commands to directly access the simulated machine state. Since this would be impossible when using a real quantum computer, they should be regarded as a non-standard extention to the QCL language.

$$\begin{array}{rcl} stmt & \leftarrow & \texttt{dump}\ [\ expr\ ]\ ; \\ & \leftarrow & \texttt{load}\ [\ expr\ ]\ ; \\ & \leftarrow & \texttt{save}\ [\ expr\ ]\ ; \end{array}$$

If called without arguments, the `dump` command prints the current machine state in bra-ket notation. When a quantum expression is given, it prints the probability spectrum instead.

```
qcl> qureg q[2];
qcl> Mix(q);
qcl> dump;
: STATE: 2 / 4 qubits allocated, 2 / 4 qubits free
0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
qcl> dump q;
: SPECTRUM q: |..10>
0.25 |00> + 0.25 |01> + 0.25 |10> + 0.25 |11>
```

The base-vectors are given in binary notation if the number of qubits is ≤ 32 and in hexadecimal otherwise. A particular output format can be forced by using the option `--dump-format`.

```
qcl> set dump-format "x";
qcl> dump;
: STATE: 2 / 4 qubits allocated, 2 / 4 qubits free
0.5 |0x0> + 0.5 |0x2> + 0.5 |0x1> + 0.5 |0x3>
```

If the `--auto-dump` option is set, the current state is logged at the shell-prompt in interactive mode. By default, `--auto-dump` is active if the number of qubits ≤ 8.

The current machine-state can be loaded and saved with the `load` and `save` command. State files have the extention `.qst`. If no filename is given, the default file `qclstate.qst` is used.

2.4.3 Flow Control

2.4.3.1 Blocks

All flow control statements operate on blocks of code. A block is a list of statements enclosed in braces:

$$\textit{block} \leftarrow \texttt{\{}\ \textit{stmt}\ \{\ \textit{stmt}\ \}\ \texttt{\}}$$

Blocks may only contain executable statements, no definitions. Unlike C, a block is not a compound statement and always part of a control structure. To avoid ambiguities with nesting, the braces are obligatory, even for single commands.

2.4.3.2 Conditional Branching

The `if` and `if-else` statements allow for the conditional execution of blocks, depending on the value of a boolean expression.

$$\textit{stmt} \leftarrow \texttt{if}\ \textit{expr}\ \textit{block}\ [\ \texttt{else}\ \textit{block}]$$

If *expr* evaluates to `true`, the `if`-block is executed. If *expr* evaluates to `false`, the `else`-block is executed if defined.

2.4.3.3 Counting Loops

`for`-loops take a counter *identifier* of type integer[5] which is incremented from $expr_{from}$ to $expr_{to}$. The loop body is executed for each value of *identifier*.

$$stmt \leftarrow \texttt{for}\ identifier\ \texttt{=}\ expr_{from}\ \texttt{to}\ expr_{to}\ [\,\texttt{step}\ expr_{step}]\ block$$

Inside the body, the counter is treated as a constant. The increment is $expr_{step}$ or 1 if unspecified. If $(expr_{to} - expr_{to})\,expr_{step} < 0$ the loop isn't executed at all.

```
qcl> int i;
qcl> for i=10 to 2 step -2 {  print i^2; }
: 100
: 64
: 36
: 16
: 4
qcl> for i=1 to 10 {  i=i^2; }          // i is constant in body
! unknown symbol: Unknown variable i
```

When the loop is finished, *identifier* is set to $expr_{to}$.

2.4.3.4 Conditional Loops

QCL supports two types of conditional loops:

$$\begin{aligned} stmt &\leftarrow \texttt{while}\ expr\ block \\ &\leftarrow block\ \texttt{until}\ expr\ ; \end{aligned}$$

A `while`-loop is iterated as long as a the condition *expr* is satisfied. When *expr* evaluates to `false`, the loop terminates.

An `until`-loop is executed at least once and iterated until the condition *expr* is satisfied.

2.4.3.5 Error Reporting

User defined routines often require their parameters to match certain conditions (e.g. sizes of quantum register arguments). Abnormal termination of subroutines can be forced with the `exit` statement.

$$stmt \leftarrow \texttt{exit}\ [\,expr]\ ;$$

Exit takes an error messages of type `string` as argument. Consider the custom `Swap` operator (Table 2.10) on page 39.

[5] This is to avoid subtle problems with floating point arithmetic

```
$ qcl -n -i swap.qcl
qcl> qureg q[2];
qcl> qureg p[1];
qcl> Swap(q,p);
! in qufunct Swap: user error: Swap: unmatching register sizes
```

If `exit` is called without arguments, the current subshell is closed as described in section 2.1.3.3.

2.5 Subroutines

2.5.1 Introduction

2.5.1.1 Syntax

QCL provides 4 kinds of subroutines: classical functions, pseudo-classical operators (`qufunct`), general unitary operators (`operator`) and procedures (`procedure`). The basic syntax for all subroutine declarations is

$$
\begin{array}{rcl}
\textit{def} & \leftarrow & (\,\textit{type}\,|\,\textit{routine-type}\,)\ \textit{identifier arg-list body} \\
\textit{routine-type} & \leftarrow & \texttt{operator}\,|\,\texttt{qufunct}\,|\,\texttt{procedure} \\
\textit{arg-list} & \leftarrow & \texttt{(}\ [\,\textit{arg-def}\ \{\ \texttt{,}\ \textit{arg-def}\}]\ \texttt{)} \\
\textit{arg-def} & \leftarrow & \textit{type identifier} \\
\textit{body} & \leftarrow & \texttt{\{}\,\{\,\textit{const-def}\,|\,\textit{var-def}\}\ \{\,\textit{stmt}\,\}\,\texttt{\}}
\end{array}
$$

2.5.1.2 Hierarchy of Subroutines

Since QCL allows for the inverse call of operators and can perform scratch-space management for quantum functions, the allowed side effects on the classical program state as well as on the quantum machine state have to be strictly specified.

routine type	**program state**	**machine state**
`procedure`	all	all
`operator`	none	unitary
`qufunct`	none	pseudo-classic
functions	none	none

Table 2.11: hierarchy of QCL Subroutines and allowed side-effects

The 4 QCL routine types form a call hierarchy, which means that a routine may invoke only subroutines of the same or a lower level (see table 2.11).

The mathematical semantic of QCL operators and functions requires that every call is reproducable. This means, that not only the program state must not be changed by these routines, but also that their execution may in no way depend on the global program state which includes global variables, options and the state of the internal random nuber generator.[6]

2.5.1.3 External Routines

While QCL incorporates a classical programming language, to provides all the necessary means to change the program state, there is no hardwired set of elementary operators to manipulate the quantum machine state, since this would require assumptions about the architecture of the simulated quantum computer.

An elementary `operator` or `qufunct` can be incorporated by declaring it as `extern`.

$$\begin{aligned} \textit{def} \;&\leftarrow\; \texttt{extern operator}\ \textit{identifier arg-list}\ ; \\ &\leftarrow\; \texttt{extern qufunct}\ \textit{identifier arg-list}\ ; \end{aligned}$$

External operators have no body since they are not executed within QCL, but merely serve as a hook for a binary function which implements the desired operation directly by using the numeric QC-library [17] and is linked to the interpreter.

The interpreter `qcl` includes binary versions of several common operators, including an implementation of the `Fanout` operator (see section 2.5.6.2) which is used by QCL scratch space management, and patterns of general unitary matrices to allow the implementation of new elementary operators.

To conveniently define a custom set of elementary operators, the external declarations can be included into the default include file `default.qcl`. Note that a definition of a `Fanout` has to be provided if local scratch variables are to be used.

For a complete list of available external operators, please refer to section 3.1.

2.5.2 Functions

Functions are the most restrictive routine type and don't allow any interactions with the global state.

[6] These restrictions can be partially overridden for debugging purposes by using the `shell` command

User defined functions may be of any classic type, namely `int`, `real`, `complex` or `string`, and may take an arbitrary number of classical parameters. The value of the function is passed to the invoking routine by the `return` statement.

```
int digits(int n) {           // calculate the number of
  return 1+floor(log(n,2));   // binary digits of n
}
```

Lokal variables can be defined at the top of the function body.

```
int fibonachi(int n) {       // calculate the n-th
      int a=0;               // fibonachi number
      int b=1;               // by iteration
      int i;
      for i = 1 to n {
              b = a+b;
              a = b-a;
      }
      return a;
}
```

Functions can call other functions within their body. Recursive calls are also allowed.

```
int fac(int n) {              // calculate n!
  if n<2 {                    // by recursion
    return 1;
  } else {
    return n*fac(n-1);
  }
}
```

Other than most internal functions, no implicit typecasting is performed, so the function arguments have to exactly match the specified parameter type.

2.5.3 Procedures

Procedures are the most general routine type and used to implement the classical control structures of quantum algorithms which generally involve evaluation loops, the choise of applied operators, the interpretation of measurements and classical probabilistic elements.

With the exception of routine declarations, procedures allow the same operations as are available in global scope (e.g. at the shell prompt) allowing arbitrary changes to both the program and the machine state. Operations exclusive to procedures are

- Access to global variables

- (Pseudo) Random numbers by using the pseudo-function `random` (see section 2.2.3.7)
- Non-unitary operations on the machine state by using the `reset` and `measure` commands (see section 2.4.2.2)
- User input by using the `input` command (see section 2.4.1.3)

Procedures can take any number of classical or quantum arguments and may call all types of subroutines.

```
procedure prepare(qureg q) {
  const l = #q/2;         // use one half of the register
  int i;                  // for the offset
  reset;                  // initialize machine state
  Mix(q[l:#q-1]);         // generate periodic distribution
  for i = 0 to l-1 {      // randomize the offset
    if 0.5<random() {
      Not(q[i]);
    }
  }
}
```

The procedure `prepare` generates a periodic test state with random offset, as we have used in the DFT example in section 2.1.2.

```
qcl> qureg q[4];
qcl> prepare(q);
[4/4] 0.5 |0010> + 0.5 |1010> + 0.5 |0110> + 0.5 |1110>
qcl> prepare(q);
[4/4] 0.5 |0000> + 0.5 |1000> + 0.5 |0100> + 0.5 |1100>
qcl> prepare(q);
[4/4] 0.5 |0011> + 0.5 |1011> + 0.5 |0111> + 0.5 |1111>
```

Procedures may declare local variables of classical and quantum types. When local quantum registers are used, it is up to the programmer to properly empty them again, which can either be acheived by uncomputing or by a `reset` command. Table 2.12 shows a simple game where a local quantum register is used to generate "real" random numbers.

2.5.4 General Operators

The routine type `operator` is used for general unitary operators. Conforming to the mathematical notion of an operator, a call with the same parameters has to result in exactly the same transformation, so no global variable references, random elements or dependencies on input are allowed.

Since the type `operator` is restricted to reversible transformations of the machine state, `reset` and `measure` commands are also forbidden.

```
procedure quRoulette() {
  qureg q[5];
  int field;
  int number;
  input "Enter field number:",field;
  repeat {
    Mix(q);
    measure q,number;
    reset;
  } until number<=36;
  if field==number {
    print "Number",number,"You won!";
  } else {
    print "Number",number,"You lose.";
  }
}
```

Table 2.12: **roulette.qcl** quantum roulette

2.5.4.1 Operator Arguments

Operators work on one or more quantum registers (register operator, see section 1.3.2.2), so depending on the mapping of the registers, a call of an m qubit operator with a total quantum heap of n qubits can result in $\frac{n!}{(n-m)!}$ different unitary transformations.

In QCL, this polymorphism is even further extended by the fact, that quantum registers can be of different sizes, so for every quantum parameter $\mathbf{s}$, the register size $\#\mathbf{s} = |\mathbf{s}|$ is an implicit extra parameter of type `int`. An addition to that, operators can take an arbitrary number of explicit classical arguments.

If more than one argument register is given, their qubits may not overlap.

```
qcl> qureg q[4];
qcl> qureg p=q[2:3];
qcl> CNot(q[1\2],p);
! runtime error: quantum arguments overlapping
```

2.5.4.2 Inverse Operators

As allready mentioned in section 2.4.1.2, operator calls can be inverted by the adjungation prefix '`!`'. The adjoint operator to a composition of unitary operators is[7]

$$\left(\prod_{i=1}^{n} U_i\right)^\dagger = \prod_{i=n}^{1} U_i^\dagger \tag{2.10}$$

Since the sequence of applied suboperators is specified using a procedural classical language which cannot be executed in reverse, the inversion the composition, is is achieved by the delayed execution of operator calls.

When the adjungation flag is set, the operator body is executed and all calls of suboperators are pushed on a stack which is then processed in reverse order with inverted adjungation flags.

2.5.4.3 Local Registers

As opposed to pseudo-classic operators, it is in general impossible to uncompute an unitary operator in order to free a local register again without also destroying the intended result of the computation. This is a fundamental limitation of QC known as the *non cloning theorem* which results from the fact that a cloning operation i.e. a transformation with meets the condition

$$U : |\psi\rangle|0\rangle \to |\psi\rangle|\psi\rangle \tag{2.11}$$

for an arbitrary[8] $|\psi\rangle$ cannot be unitary if $|\psi\rangle$ is a composed state because

$$U\left(a|0,0\rangle + b|1,0\rangle\right) = a^2|0,0\rangle + ab\,|0,1\rangle + ba\,|1,0\rangle) + b^2|1,1\rangle \tag{2.12}$$

$$\neq a\,U\,|0,0\rangle + b\,U\,|1,0\rangle = a|0,0\rangle + b|1,1\rangle \tag{2.13}$$

U can only be unitary if $|\psi\rangle$ is in a pure state, i.e. $|\psi\rangle = |i\rangle$, in which case $U = \mathit{FANOUT}$.

Due to the lack of a unitary copy operation for quantum states, Bennet-style scratch space management is impossible for general operators since it is based on cloning the result register.

Despite this limitation, it is possible in QCL to allocate temporary quantum registers but it is up to the programmer to properly uncompute them again. If the option `--check` is set, proper cleanup is verified by the simulator.

---

[7] To avoid ambiguities with non-commutative matrix products, we use the convention $\prod_{i=1}^{n} f_i = f_n f_{n-1} \ldots f_1$

[8] For any particular $|\psi\rangle$ an infinite number of unitary "cloning" operators trivially exists, as e.g. $U_\psi = \sum_{i,j,k} |i, j \oplus k\rangle\langle k|\psi\rangle\langle i, j|$

```
qcl> set check 1
qcl> operator foo(qureg q) { qureg p[1]; CNot(p,q); }
qcl> qureg q[1];
qcl> Mix(q);
[1/4] 0.707107 |0000> + 0.707107 |0001>
qcl> foo(q);
! in operator foo: memory error: quantum heap is corrupted
[1/4] 0.707107 |0000> + 0.707107 |0011>
```

Local registers are useful if an operator contains some intermediary pseudo-classic operations which require scratch space. See the implementation of modular addition (`addn`) in section 3.2.2.1 (page 62) for an example.

2.5.5 Pseudo-classic Operators

The routine type `qufunct` is used for pseudo-classic operators and quantum functions, so all transformations have to be of the form

$$|\Psi\rangle = \sum_i c_i |i\rangle \to \sum_{i,j} c_i \delta_{j\pi_i} |j\rangle = |\Psi'\rangle \tag{2.14}$$

with some permutation π. All n qubit pseudo-classic operators F therefore have the common eigenstate

$$|\Psi\rangle = 2^{-\frac{1}{2}n} \sum_{i=0}^{2^n-1} |i\rangle \Rightarrow F\,|\Psi\rangle = |\Psi\rangle \tag{2.15}$$

2.5.5.1 Bijective Functions

The most straightforward application for pseudo-classic operators is the direct implementation of bijective functions (see section 1.3.3.1)

```
qufunct inc(qureg x) {
  int i;
  for i = #x-1 to 1 {
    CNot(x[i],x[0:i-1]);
  }
  Not(x[0]);
}
```

The operator `inc` shifts the base-vectors of it's argument. In analogy to boson states, where the increment of the eigenstate corresponds to the generation of a particle, `inc` is a *creation operator*.[9]

[9] In fact, this is not quite correct, since other than bosons, an n qubit register is limited to 2^n states, so `inc` $|2^n-1\rangle = |0\rangle$ whereas $a^\dagger\,|2^n-1\rangle = |2^n\rangle$

```
qcl> qureg q[4];
qcl> inc(q);
[4/4] 1 |0001>
qcl> inc(q);
[4/4] 1 |0010>
qcl> inc(q);
[4/4] 1 |0011>
qcl> inc(q);
[4/4] 1 |0100>
```

2.5.5.2 Conditional Operators

When it comes to more complicated arithmetic operations, it is often required to apply a transformation to a register **a** in dependence on the content of another register **e**.

If all qubits of **e** are required to be set, for the transformation to take place, the operator is a conditional operator with the invariant (`quconst`) enable register **e** (see section 1.3.5).

A simple example for a conditional operator is the Toffoli gate $T : |x, y, z\rangle \to |x \oplus (y \wedge z), y, z\rangle$ or it's generalisation, the controlled not gate. A conditional version of the above increment operator is also easy to implement:

```
qufunct cinc(qureg x,quconst e) {
  int i;
  for i = #x-1 to 1 step -1 {
    CNot(x[i],x[0:i-1] & e);
  }
  CNot(x[0],e);
}
```

Now, only base-vectors of the form $|i\rangle|11\ldots\rangle_{\mathbf{s}}$ are incremented:

```
qcl> qureg q[4]; qureg e[2]; Mix(e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110000>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110001>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110010>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110011>
```

2.5.6 Quantum Functions

As defined in section 1.3.3.2, a quantum function F is a pseudo-classic operator with the characteristic

$$F : |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} \to |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}} \quad \text{with} \quad f : \mathbf{B}^n \to \mathbf{B}^m \tag{2.16}$$

If we require the argument register **x** to be invariant to F by declaring **x** as `quconst`, this leaves us with $2^{(2^n-1)m}$ possible pseudo-classic implementations of F for any given f. To reflect the fact that $F\,|x, y \neq 0\rangle$ is undefined, the target register has to be of type `quvoid`. (see section 2.3.2.3).

Since, according to the above definition, quantum functions are merely ordinary pseudo-classic operators, whose specification is restricted to certain types of input states, they also use the same QCL routine type `qufunct`.

The following example calculates the parity of **x** and stores it to **y**:

```
qufunct parity(quconst x,quvoid y) {
  int i;
  for i = 0 to #x-1 {
    CNot(y,x[i]);
  }
}

qcl> qureg x[2]; qureg y[1]; Mix(x);
[3/3] 0.5 |000> + 0.5 |010> + 0.5 |001> + 0.5 |011>
qcl> parity(x,y);
[3/3] 0.5 |000> + 0.5 |110> + 0.5 |101> + 0.5 |011>
```

2.5.6.1 Scratch parameters

We can extend the notion of quantum functions, by also allowing an explicit scratch register **s** (see section 2.3.2.4) as an optional parameter to F, so we end up with an operator $F(\mathbf{x}, \mathbf{y}, \mathbf{s})$ with the characteristic

$$F : |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \to |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \tag{2.17}$$

Using the `parity` and the `cinc` operator form the above examples, we can implement an "add parity" function $f(x) = x + \text{parity}(x)$ by providing a scratch qubit:

```
qufunct addparity(quconst x,quvoid y,quscratch s) {
  parity(x,s);      // write parity to scratch
  x -> y;           // Fanout x to y
  cinc(y,s);        // increment y if parity is odd
  parity(x,s);      // clear scratch
}

qcl2> qureg x[2]; qureg y[2]; qureg s[1]; Mix(x);
[5/8] 0.5 |00000> + 0.5 |00010> + 0.5 |00001> + 0.5 |00011>
qcl2> addparity(x,y,s);
[5/8] 0.5 |00000> + 0.5 |01110> + 0.5 |01001> + 0.5 |01111>
```

Instead of providing a explicit scratch parameter, we can, of course, also use a local register of type `qureg`, which is functionally equivalent:

```
qufunct addparity2(quconst x,quvoid y) {
  qureg s[1];
  parity(x,s);
  x -> y;
  cinc(y,s);
  parity(x,s);
}

qcl2> qureg x[2]; qureg y[2]; Mix(x);
[4/8] 0.5 |00000> + 0.5 |00010> + 0.5 |00001> + 0.5 |00011>
qcl2> addparity2(x,y);
[4/8] 0.5 |00000> + 0.5 |01110> + 0.5 |01001> + 0.5 |01111>
```

Explicit scratch parameters are useful to save memory, if a quantum function F is to be used by another operator U, which still has empty scratch registers at the moment, the suboperator is called, which would e.g. be the case if U is of the form

$$U(\mathbf{x},\mathbf{y},\mathbf{s},\ldots) = \left(\prod_{i=2}^{l} U_i(\mathbf{x},\mathbf{y},\mathbf{s},\ldots)\right) F(\mathbf{x},\mathbf{y},\mathbf{s})\, U_1(\mathbf{x},\mathbf{y},\ldots) \tag{2.18}$$

Since both, explicit scratch parameters of type `quscratch` and local registers of type `qureg`, have to be uncomputed manually, they are especially useful for quantum functions $U : |x,0,0\rangle \to |x, f(s(x),x),0\rangle$ of the form

$$U(\mathbf{x},\mathbf{y},\mathbf{s}) = S(\mathbf{x},\mathbf{s})F(\mathbf{x},\mathbf{s},\mathbf{y})S^{\dagger}(\mathbf{x},\mathbf{s}) \tag{2.19}$$

if S is invariant to $\mathbf{x}$ and F is invariant to $\mathbf{x}$ and $\mathbf{s}$, because the uncomputation of $\mathbf{s}$ doesn't require an additional register to temporarily save $\mathbf{y}$ (see section 1.3.4.2) as would be the case, if a managed local scratch register of type `quscratch` would be used instead (see below).

2.5.6.2 The Fanout Operator

The restriction to base-vector permutations implies that the computational path of a pure state is also a sequence of pure states, so in the case of superpositions each base-vector can be treated separately.

As shown in section 2.5.4, a arbitrary pure state $|\psi\rangle = |i\rangle$ can be copied onto an empty register by a unitary *FANOUT* operation:

$$\mathit{FANOUT} : |\psi\rangle|0\rangle = |i,0\rangle \to |i,i\rangle = |\psi\rangle|\psi\rangle \tag{2.20}$$

For non-empty target registers, $\mathit{FANOUT}\,|i,j\neq 0\rangle$ is undefined, so for two n qubit registers there are $(2^{2n}-2^n)!$ possible pseudo-classic implementations of a fanout gate.[10]

[10] In fact, the definition of fanout used by QCL is somewhat more restrictive, since it requires the first argument to be invariant (`quconst`). This can be overridden be redeclaring the argument as `qureg` if necessary (see section 2.3.2.5)

```
extern qufunct CNot(qureg q,quconst c);

qufunct Fanout(quconst a,quvoid b) {
  int i;
  if #a != #b { exit "Fanout: arguments must be of equal size"; }
  for i=0 to #a-1 {
    CNot(b[i],a[i]);
  }
}
```

Table 2.13: **fanout.qcl** Custom implementation of Fanout

Table 2.13 shows a realisation using controlled-not gates which is mathematically equivalent to the default implementation of the external operator `Fanout`.

2.5.6.3 Scratch Space Management

The quantum type `quscratch` declares a local register as managed scratch space. Managed scratch space (or junk) registers are temporary registers which are empty when allocated and automatically get uncomputed after the body of the `qufunct` has been applied.

So, in contrast to local `qureg` registers or `quscratch` parameters, a local `quscratch` register $\mathbf{j}$ has *not* to be emptied within the the `qufunct` definition but can be left dirty. So, in order to compute some $f(x)$, it is sufficient, the the body of the quantum function merely implements some operator $F : |x, 0, 0\rangle \to |x, f(x), j(x)\rangle$ with an arbitrary junk string $j(x)$ in the scratch register.

When a quantum function with the local junk register $\mathbf{j}$ and the body $F(\mathbf{x}, \mathbf{y}, \mathbf{j})$ is called, an additional scratch register $\mathbf{y}'$ of the same size as $\mathbf{y}$ is allocated and instead of the 3 register operator F, the 4 register operator F' is applied, which is defined as

$$F'(\mathbf{x}, \mathbf{y}, \mathbf{j}, \mathbf{y}') = F^\dagger(\mathbf{x}, \mathbf{y}', \mathbf{j})\, \mathit{FANOUT}(\mathbf{y}', \mathbf{y})\, F(\mathbf{x}, \mathbf{y}', \mathbf{j}) \tag{2.21}$$

F' initially calls F, but instead of $\mathbf{y}$, a temporary target register $\mathbf{y}'$ is used. The desired result $f(x)$ is then copied onto the original target register $\mathbf{y}$, while the undesired junk result $j(x)$ is left in the junk register. By undoing the initial computation by applying the adjoint operator $F^\dagger$, both, the junk register $\mathbf{j}$ and the scratch register $\mathbf{y}'$, get uncomputed again, so the whole procedure is:

$$|x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|0\rangle_{\mathbf{j}}|0\rangle_{\mathbf{y'}} \xrightarrow{F(\mathbf{x},\mathbf{y'},\mathbf{j})} |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|j(x)\rangle_{\mathbf{j}}|f(x)\rangle_{\mathbf{y'}} \xrightarrow{FANOUT(\mathbf{y'},\mathbf{y})} \tag{2.22}$$
$$|x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}}|j(x)\rangle_{\mathbf{j}}|f(x)\rangle_{\mathbf{y'}} \xrightarrow{F^{\dagger}(\mathbf{x},\mathbf{y'},\mathbf{j})} |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}}|0\rangle_{\mathbf{j}}|0\rangle_{\mathbf{y'}}$$

By using the conditional increment operator from page 50 we can construct a quantum function `bitcmp` which implements a "bit comparison" function $f(x_1, x_2)$ which returns 1 if the bitstrings x_1 and x_2 contain the same number of set bits, and zero otherwise.

```
qufunct bitcmp(quconst x1,quconst x2,quvoid y) {
  const n=ceil(log(max(#x1,#x2)+1,2));
  int i;
  quscratch j[n];        // allocate a managed scratch register
  for i=0 to #x1-1 {     // j = number of bits in x1
    cinc(j,x1[i]);       // increment j if bit i of x1 is set
  }
  Not(j);                // j = 2^n-j-1 = -1-j mod 2^n
  for i=0 to #x2-1 {     // j = j+number of bits in x2
    cinc(j,x2[i]);       // increment j if bit i of x1 is set
  }
  CNot(y,j);             // set y=1 if j==2^n-1
}

qcl> qureg x1[2]; qureg x2[2]; qureg y[1];
qcl> Mix(x1[1]); Mix(x2[0]); Not(x2[1]);
[5/8] 0.5 |00001000> + 0.5 |00001100> + 0.5 |00001010> + 0.5 |00001110>
qcl> bitcmp(x1,x2,y);
[5/8] 0.5 |00001000> + 0.5 |00001100> + 0.5 |00011010> + 0.5 |00001110>
```

By using the option `--log` we can trace the call of each elementary operator:

```
qcl> set log 1
qcl> bitcmp(x1,x2,y);
@ CNot(qureg q=|0.......>,quconst c=|.0.....1>)
@ CNot(qureg q=|.0......>,quconst c=|.......0>)
@ CNot(qureg q=|0.......>,quconst c=|.0....1.>)
@ CNot(qureg q=|.0......>,quconst c=|......0.>)
@ Not(qureg q=|10......>)
@ CNot(qureg q=|0.......>,quconst c=|.0...1..>)
@ CNot(qureg q=|.0......>,quconst c=|.....0..>)
@ CNot(qureg q=|0.......>,quconst c=|.0..1...>)
@ CNot(qureg q=|.0......>,quconst c=|....0...>)
@ CNot(qureg q=|..0.....>,quconst c=|10......>)
@ Fanout(quconst a=|..0.....>,quvoid b=|...0....>)
@ !CNot(qureg q=|..0.....>,quconst c=|10......>)
@ !CNot(qureg q=|.0......>,quconst c=|....0...>)
@ !CNot(qureg q=|0.......>,quconst c=|.0..1...>)
@ !CNot(qureg q=|.0......>,quconst c=|.....0..>)
@ !CNot(qureg q=|0.......>,quconst c=|.0...1..>)
@ !Not(qureg q=|10......>)
@ !CNot(qureg q=|.0......>,quconst c=|......0.>)
@ !CNot(qureg q=|0.......>,quconst c=|.0....1.>)
@ !CNot(qureg q=|.0......>,quconst c=|.......0>)
@ !CNot(qureg q=|0.......>,quconst c=|.0.....1>)
```

The first 10 operations belong to the body operator F; after the $FANOUT$, they are repeated in reverse order with inverted adjungation flags ($F^\dagger$).

By additionally using the option `--log-state`, we can also trace the evolution of the machine state.

```
qcl> bitcmp(x1,x2,y);
@ CNot(qureg q=|0.......>,quconst c=|.0.....1>)
% 0.5 |00001000> + 0.5 |00001100> + 0.5 |00001010> + 0.5 |00001110>
.....
% 0.5 |00001000> + 0.5 |01001100> + 0.5 |11101010> + 0.5 |00001110>
@ Fanout(quconst a=|..0.....>,quvoid b=|...0....>)
% 0.5 |00001000> + 0.5 |01001100> + 0.5 |11111010> + 0.5 |00001110>
.....
@ !CNot(qureg q=|0.......>,quconst c=|.0.....1>)
% 0.5 |00001000> + 0.5 |00001100> + 0.5 |00011010> + 0.5 |00001110>
```

Chapter 3

Operators and Algorithms

3.1 Elementary Operators

This section introduces the possible elementary operators, that can be used with the current implementation of QCL. Since QCL as a language doesn't enforce a specific set of elementary operators, they have to be declared as external (see section 2.5.1.3), to make them available to programs.

This is usually done within the default include file `default.qcl` (see appendix A.1), which is loaded at startup.

3.1.1 General Unitary Operators

3.1.1.1 Unitary Matrices

The most general form for specifying a unitary operator (or any other linear transformation) is by defining it's matrix elements: An n qubit unitary operator U describes a transformation $U : \mathbf{C}^{2^n} \to \mathbf{C}^{2^n}$ and therefore corresponds to a $2^n \times 2^n$ matrix in $\mathbf{C}$

$$U = \sum_{i,j=0}^{2^n} |i\rangle u_{ij} \langle j| = \begin{pmatrix} u_{0,0} & \cdots & u_{0,2^n-1} \\ \vdots & \ddots & \vdots \\ u_{2^n-1,0} & \cdots & u_{2^n-1,2^n-1} \end{pmatrix} \tag{3.1}$$

Since for a unitary transformation $U^\dagger U = (U^*)^{\mathrm{T}} U = I(n)$, the Matrix U unitary if and only if

$$\bigwedge_{i,j=0}^{2^n-1} \sum_{k=0}^{2^n-1} u_{ik}^* u_{kj} = 1 \tag{3.2}$$

QCL provides external operators for general unitary 2×2, 4×4 and 8×8 matrices, which the programmer can use to directly implement a custom set of 1, 2 and 3 qubit gates.

```
extern operator Matrix2x2(
  complex u00,complex u01,
  complex u10,complex u11,
qureg q);
extern operator Matrix4x4(...,qureg q);
extern operator Matrix8x8(...,qureg q);
```

Matrix operators are checked for unitarity before they are applied:

```
qcl> const i=(0,1);
qcl> qureg q[1];
qcl> Matrix2x2(i*cos(pi/6),i*sin(pi/6),(0,0),(1,0),q);
! external error: matrix operator is not unitary
```

3.1.1.2 Qubit Rotation

The rotation of a single qubit is defined by the transformation matrix $U(\theta)$

$$U(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & \sin\frac{\theta}{2} \\ -\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \tag{3.3}$$

The factor $-\frac{1}{2}$ to θ is set in analogy to spin rotations, which can be shown to be of the form $\mathcal{D} = e^{-\frac{i}{2}\delta_j \sigma_j}$ and thus have a period of 4π.

```
extern operator Rot(real theta,qureg q);
```

In contrast to most other external Operators, `Rot` is not generalised to work with arbitrary register sizes.

```
qcl> Rot(pi/2,q);
! external error: Only single qubits can be rotated
```

3.1.1.3 Hadamard Gate

The *Hadamard Gate* is a special case of a generalised qubit Rotation and defined by the transformation matrix H

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.4}$$

For the case of n qubit registers, H can be generalised to

$$H : |i\rangle \to 2^{-\frac{n}{2}} \sum_{j \in \mathbf{B}^n} (-1)^{(i,j)} |j\rangle \tag{3.5}$$

The vectors $\mathcal{B}' = \{i \in \mathbf{B}^n \mid |i'\rangle = H|i\rangle\}$ form the *Hadamard base* or *dual base* or *parity base* to $\mathcal{B} = \{i \in \mathbf{B}^n \mid |i\rangle\}$.

The Hadamard Transformation is self adjoint (i.e. $H^\dagger = H$), which, for unitary operators, implies that $H^2 = I$.

Since $\mathcal{B}'$ only contains uniform superpositions that just differ by the signs of the base-vectors, the external implementation of H id called `Mix`.

```
extern operator Mix(qureg q);
```

3.1.1.4 Conditional Phase Gate

The *conditional phase gate* is a pathological case of a conditional operator (see section 1.3.5), for the zero-qubit phase operator $e^{i\phi}$.

$$V(\phi) : |\epsilon\rangle \to \begin{cases} e^{i\phi}\,|\epsilon\rangle & \text{if } \epsilon = 111\ldots \\ |\epsilon\rangle & \text{otherwise} \end{cases} \tag{3.6}$$

The conditional phase gate is used in the quantum Fourier transform (see section 3.2.3).

```
extern operator CPhase(real phi,qureg q);
```

3.1.2 Pseudo-classic Operators

3.1.2.1 Base Permutation

The most general form for specifying an n qubit pseudo-classic operator U, is by explicitly defining the underlying permutation π of base-vectors:

$$U_{pc.} = \sum_{i=0}^{2^n-1} |\pi_i\rangle\langle i| = \langle \pi_0, \pi_1 \ldots \pi_{2^n-1}\rangle \tag{3.7}$$

QCL provides external operators for vector permutations for $|\pi| = 2, 4, 8, 16, 32$ and 64 which the programmer can use to directly implement a custom set of 1 to 6 qubit pseudo-classical operators:

```
extern qufunct Perm2(int p0 ,int p1 ,qureg q);
extern qufunct Perm4(int p0 ,int p1 ,int p2 ,int p3 ,qureg q);
extern qufunct Perm8(...,qureg q);
extern qufunct Perm16(...,qureg q);
extern qufunct Perm32(...,qureg q);
extern qufunct Perm64(...,qureg q);
```

Base permutations are checked for unitarity before they are applied (i.e. it is verified that the given integer sequence is in fact a permutation)

```
qcl> qureg q[3];
qcl> Perm8(0,0,1,2,3,4,5,6,q);
! external error: no permutation
```

3.1.2.2 Fanout

The *FANOUT* operation is a quantum function (see section 1.3.3.2) and stands for a class of transformations with the characteristic $FANOUT$: $|x, 0\rangle \to |x, x\rangle$ (see section 2.5.6.2 for details).

The external fanout operator of QCL is defined as

$$FANOUT : |x, y\rangle \to |x, x \oplus y\rangle, \tag{3.8}$$

however, it is considered bad programming style to rely on this particular implementation.

```
extern qufunct Fanout(quconst a,quvoid b);
```

3.1.2.3 Swap

The $SWAP$ operator exchanges the qubits of two equal sized registers ($SWAP : |x, y\rangle \to |y, x\rangle$). A one to one qubit $SWAP$ operator has the transformation matrix

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

```
extern qufunct Swap(qureg a,qureg b);
```

3.1.2.4 Not and Controlled Not

The not operator C inverts a qubit. Its transformation matrix is

$$C = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.10}$$

The controlled-not operator $C_{[[\mathbf{e}]]}$ is the conditional operator (see section 1.3.5) to C with the enable register $\mathbf{e}$:

$$C_{[[\mathbf{e}]]} : |b\rangle|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} |1-b\rangle\,|\epsilon\rangle_{\mathbf{e}} & \text{if } \epsilon = 111\ldots \\ |b\rangle|\epsilon\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{3.11}$$

```
extern qufunct Not(qureg q);
extern qufunct CNot(qureg q,quconst c);
```

The QCL versions of `Not` and `CNot` also work on target registers:

```
qcl> qureg q[4]; qureg p[4];
qcl> Not(q);
[8/8] 1 |00001111>
qcl> CNot(p,q);
[8/8] 1 |11111111>
```

3.2 Composed Operators

This section introduces the unitary operators needed by the Shor algorithm presented in section 3.3.

3.2.1 Pseudo-classic Operators

3.2.1.1 Simple Bit-Manipulations

Reverting Registers The `flip` operator reverts the bit order of a register.

$$\texttt{flip} : |b_1, b_2 \ldots b_n\rangle \to |b_n, b_{n-1} \ldots b_1\rangle \tag{3.12}$$

```
qufunct flip(qureg q) { // pseudo classic op to swap bit order
  int i;                // declare loop counter
  for i=0 to #q/2-1 {   // swap 2 symmetric bits
    Swap(q[i],q[#q-i-1]);
  }
}
```

Conditional Exclusive Or The `cxor` operator has the functionality of a conditional `CNot`-based *FANOUT* operation:

$$\texttt{cxor} : |a\rangle_{\mathbf{a}}|b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} |a\rangle_{\mathbf{a}}|a \oplus b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} & \text{if } \epsilon = 111\ldots \\ |a\rangle_{\mathbf{a}}|b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{3.13}$$

```
qufunct cxor(quconst a,qureg b,quconst enable) {
  int i;
  for i=0 to #a-1 {
    CNot(b[i],a[i] & enable);
  }
}
```

3.2.1.2 Comparing Registers

Comparing two classical binary numbers a and b can be simply achieved by comparing form the highest to the lowest bits and returning at the first mismatch.

```
for i=n-1 to 0 { // check whether b<a
  if bit(b,i)<bit(a,i) { return true; }
}
return false;
```

It is, however, not so trivial if one of the values is a quantum register **b**, due to the lack of conditional branching. So, since we can't simply return form the loop, we have to look for another solution.

One possibility is to emulate an early return from the loop by using conditional operators (see section 1.3.5): For each bit comparison, we use an enable bit which is set to 1 if the bits are equal (and the loop has to continue) or to 0 if the result is decided and further comparisons should be disabled.

To compare an n qubit register $\mathbf{b}$ to a classical integer a, we have to use an $n-1$ junk register $\mathbf{j}$ to store the enable bits. The main loop of the quantum comparison $\mathbf{b} < a$ then runs a follows.

```
for i=#b-2 to 1 step -1 {       // continue for lower bits
  if bit(a,i) {                 // set new junk bit if undecided
    CNot(j[i-1],j[i] & b[i]);
    Not(b[i]);                  // honour last junk bit and
    CNot(flag,j[i] & b[i]);     // set result flag if a[i]>b[i]
  } else {
    Not(b[i]);
    CNot(j[i-1],j[i] & b[i]);
  }
  Not(b[i]);                    // restore b[i] again
}
```

For the complete implementation of the `lt` operator, as used in modular addition, please refer to appendix 3.2.1.2.

3.2.1.3 Multiplexed Adder

A multiplexed adder adds one of two classical bits a_0 and a_1 to a qubit $\mathbf{b}$, depending on the content of a selection qubit $\mathbf{s}$. The target register $\mathbf{y}_{sum} = (\mathbf{c}_{in}, \mathbf{c}_{out})$ consists of a carry-in and a carry-out qubit, to allow cascading. The truth table for the operation is:

$\mathbf{s}$	a_0	a_1	$a_\mathbf{s}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$a_\mathbf{s}$	$\mathbf{b}$	$\mathbf{c}_{in}$	$\mathbf{c}'_{in}$	$\mathbf{c}'_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The implementation of the truth-table with controlled-not gates (`muxaddbit`) is straightforward and can be found in appendix A.3.[1] A multiplexed adder for registers can be constructed by cascading several single bit adders:

[1]`muxaddbit` is actually a conditional version with an additional enable register $\mathbf{e}$, as it is needed for modular multiplication

```
qufunct muxadd(int a0,int a1,qureg sel,quconst b,quvoid sum,quconst e) {
  int i;
  for i=0 to #b-2 {           // fulladd first #b-1 bits
    muxaddbit(bit(a0,i),bit(a1,i),sel,b[i],sum[i:i+1],e);
  }                           // half add last bit
  muxaddbit(bit(a0,#b-1),bit(a1,#b-1),sel,b[#b-1],sum[#b-1],e);
}
```

3.2.2 Modular Arithmetic

Many number theoretic algorithms describe calculations in the remainder class $\mathbf{Z}_n$. On example for a quantum algorithm using modular arithmetic, is Shor's method of polynomial time quantum factoring (see section 3.3).[11].

For a more detailed discussion of unitary operators for modular arithmetic, please refer to [13].

3.2.2.1 Modular Addition

The addition modulo n of a classic integer a and a quantum register $\mathbf{b}$ can result in either $a+b$ or $(a-n)+b)$, depending on the particular base-vector $|b\rangle$.

While for $b<n$ the operation is revertible, this is not the case for $b \geq n$, so, if n doesn't happen to be a power of 2, besides the target resister $\mathbf{y}_s$ for the sum, we need an additional flag-qubit $\mathbf{y}_y$ to allow for a quantum function `addn` which is both, unitary and invariant to $\mathbf{b}$:

$$\mathtt{addn}_{a,n} : |b\rangle_{\mathbf{b}}|0\rangle_{\mathbf{y}_s}|0\rangle_{\mathbf{y}_f} \to \begin{cases} |b\rangle_{\mathbf{b}}|a+b\rangle_{\mathbf{y}_s}|1\rangle_{\mathbf{y}_{flag}} & \text{if}\, a+b<n \\ |b\rangle_{\mathbf{b}}|a+b-n\rangle_{\mathbf{y}_s}|0\rangle_{\mathbf{y}_{flag}} & \text{if}\, a+b \geq n \end{cases} \tag{3.14}$$

By using the less-than operator `lt` and the multiplexing adder `muxadd`, the implementation is rather straightforward. (The enable register $\mathbf{e}$ has been added to allow the use for modular multiplication; see below.)

```
qufunct addn(int a,int n,quconst b,quvoid flag,quvoid sum,quconst e) {
  qureg s=sum[0\#b-1];
  qureg f=sum[#b-1];
  qureg bb=b;                     // "abuse" sum and b as scratch
  lt(n-a,bb,f,s);                 // for the less-than operator
  CNot(flag,f & e);               // save result of comparison
  !lt(n-a,bb,f,s);                // restore sum and b
  muxadd(2^#b+a-n,a,flag,b,sum,e); // add either a or a-n
}
```

The only trick here is, that we redeclare the `quconst b` as `qureg`, so that we can use a "dirty" implementation of `lt` which doesn't perform any cleanup on $\mathbf{b}$ or $\mathbf{y}_s$ (`sum`), which would be pointless anyway, since the comparison gets uncomputed after the result has been saved.

Since $\mathtt{addn}_{n-a,n}$ is a quantum function for modular subtraction and thus implements the inverse function $f_{a,n}^{-1}(b) = b - a \bmod n$ to $f_{a,n} = a + b \bmod n$, we can construct an overwriting version `oaddn` of modular addition, by using the method introduced in section 1.3.4.3:

$$F' : |i, 0\rangle \xrightarrow{U_f} |i, f(i)\rangle \xrightarrow{SWAP} |f(i), i\rangle \xrightarrow{U^\dagger_{f^{-1}}} |f(i), 0\rangle \tag{3.15}$$

$\mathtt{addn}_{n-a,n}$ doesn't invert the overflow flag $\mathbf{y}_f$, so we have to switch it manually:

$$U^\dagger_{f^{-1}} = \mathtt{addn}_{n-a,n}(\mathbf{b}, \mathbf{y}_s, \mathbf{y}_f) \tag{3.16}$$

The original target registers $\mathbf{y}_s$ and $\mathbf{y}_f$ can now be allocated as unmanaged local scratch.

```
qufunct oaddn(int a,int n,qureg sum,quconst e) {
  qureg j[#sum];
  qureg f[1];

  addn(a,n,sum,f,j,e);                // junk -> a+b mod n
  Swap(sum,j);                        // swap junk and sum
  CNot(f,e);                          // toggle flag
  !addn(n-a,n,sum,f,j,e);             // uncompute b to zero
}
```

3.2.2.2 Modular Multiplication

Modular multiplication is merely a composition of conditional additions for each qubit of $\mathbf{b}$ since

$$ab \bmod n = \sum_{i=0}^{\lceil \mathrm{ld}\, b \rceil} b_i \left(2^i a \bmod n\right) \quad \text{with} \quad b_i \in \mathbf{B} \tag{3.17}$$

The first summand can be slightly optimised, since the accumulator (`prod`) is still empty.

```
qufunct muln(int a,int n,quconst b,qureg prod,quconst e) {
  int i;

  for i=0 to #prod-1 {
    if bit(a,i) { CNot(prod[i],b[0] & e); }
  }
  for i=1 to #b-1 {
    oaddn(2^i*a mod n,n,prod,b[i] & e);
  }
}
```

As above, we can construct an overwriting version, if an implementation of the inverse function exists. This is the case if $\gcd(a, n) = 1$ so a and n are relatively prime, because then the modular inverse a^{-1} with $a^{-1}a \bmod n = 1$ exists. Since we intend to use the operator for the Shor algorithm which demands that $\gcd(a^k, n) = 1$, this is good enough for us.

By using two conditional XOR gates (see section 3.2.1.1) for swapping the registers[2] we can implement a conditional $\texttt{omuln}_{[[\mathsf{e}]],a,n}|b\rangle \to |ab \bmod n\rangle$

```
qufunct omuln(int a,int n,qureg b,quconst e) {
  qureg j[#b];

  muln(a,n,b,j,e);
  !muln(invmod(a,n),n,j,b,e);
  cxor(j,b,e);
  cxor(b,j,e);
}
```

3.2.2.3 Modular Exponentiation

As with `muln`, we can construct modular exponentiation by conditionally applying `omuln` with the qubits of the exponents as enable string, according to

$$a^b \bmod n = \prod_{i=0}^{\lceil \mathrm{ld}\, b \rceil} \left(a^{2^i\, b_i} \bmod n \right) \quad \text{with} \quad b_i \in \mathbf{B} \tag{3.18}$$

Before we can start the iteration, the accumulator (`ex`) has to be initialised by 1.

```
qufunct expn(int a,int n,quconst b,quvoid ex) {
  int i;

  Not(ex[0]);                        // start with 1
  for i=0 to #b-1 {
    omuln(powmod(a,2^i,n),n,ex,b[i]);    // ex -> ex*a^2^i mod n
  }
}
```

3.2.3 Quantum Fourier Transform

For a q dimensional vector $|\psi\rangle$, the discrete Fourier transform is defined as

$$DFT : |x\rangle \;\to\; \frac{1}{\sqrt{q}} \sum_{y=0}^{q-1} e^{\frac{2\pi i}{q}\, xy}\, |y\rangle \tag{3.19}$$

---

[2]normally, 3 XOR operations are necessary to swap a register, but since one register is empty, 2 XORs suffice.

Since $|\psi\rangle$ is a combined state of n qubits, q is always a power of 2. The classical fast Fourier Transform (FFT) uses a binary decomposition of the exponent to perform the transformation in $O(n2^n)$ steps.

As suggested by Coppersmith [7], the same principle could adapted be to quantum computers by using a combination of Hadamard transformations H and conditional phase gates V (indices indicate the qubits operated on):

$$DFT' = \prod_{i=1}^{n-1} \left(H_{n-i-1}(\frac{\pi}{2}) \prod_{j=0}^{i-1} V_{n-i-1,n-j-1}(\frac{2\pi}{2^{i-j+1}}) \right) H_{n-1} \tag{3.20}$$

DFT' iterates the qubits form the MSB to the LSB, "splits" the qubits with the Hadamard transformation and then conditionally applies phases according to their relative binary position ($e^{\frac{2\pi i}{2^{i-j+1}}}$) to each already split qubit.

The base-vectors of the transformed state $|\tilde{\psi}'\rangle = DFT' |\psi\rangle$ are given in reverse bit order, so the get the actual DFT, the bits have to be flipped.

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

3.3 Shor's Algorithm for Quantum Factorisation

3.3.1 Motivation

In contrast to finding and multiplying of large prime numbers, no efficient classical algorithm for the factorisation of large number is known. An algorithm is called efficient if its execution time i.e. the number of elementary operations is assymtotically polynomial in the length of its input measured in bits. The best known (or at least published) classical algorithm (the *quadratic sieve*) needs $O\left(\exp\left((\frac{64}{9})^{1/3} N^{1/3} (\ln N)^{2/3}\right)\right)$ operations for factoring a binary number of N bits [12] i.e. scales exponentially with the input size.

The multiplication of large prime numbers is therefore a one-way function i.e. a function which can easily be evaluated in one direction, while its inversion is practically impossible. One-way functions play a major roll in cryptography and are essential to public key crypto-systems where the key for encoding is public and only the key for decoding remains secret.

In 1978, Rivest, Shamir and Adleman developed a cryptographic algorithm based on the one-way character of multiplying two large (typically above 100 decimal digits) prime numbers. The RSA method (named after the initials of their inventors) became the most popular public key system and is implemented in many communication programs.

While it is generally believed (although not formally proved) that efficient prime factorisation on a classical computer is impossible, an efficient algorithm for quantum computers has been proposed in 1994 by P.W. Shor [11].

3.3.2 The Algorithm

This section describes Shor's algorithm from a functional point of view which means that it doesn't deal with the implementation for a specific hardware architecture. A detailed implementation for the Cirac-Zoller gate can be found in [13]. For a more rigid mathematical description, please refer to [14].

3.3.2.1 Modular Exponentiation

Let $N = n_1 n_2$ with the greatest common divisor $\gcd(n_1, n_2) = 1$ be the number to be factorised, x a randomly selected number relatively prime to N (i.e. $\gcd(x, N) = 1$) and expn the modular exponentiation function with the period r:

$$\mathrm{expn}(k, N) = x^k \bmod N,\ \mathrm{expn}(k+r, N) = \mathrm{expn}(k, N),\ x^r \equiv 1 \bmod N \tag{3.21}$$

The period r is the order of $x \bmod N$. If r is even, we can define a $y = x^{r/2}$, which satisfies the condition $y^2 \equiv 1 \bmod N$ and therefore is the solution of one of the following systems of equations:

$$\begin{aligned} y_1 \equiv&\ \ 1 \bmod n_1 &\equiv 1 \bmod n_2 \\ y_2 \equiv&\ -1 \bmod n_1 &\equiv -1 \bmod n_2 \\ y_3 \equiv&\ \ 1 \bmod n_1 &\equiv -1 \bmod n_2 \\ y_4 \equiv&\ -1 \bmod n_1 &\equiv 1 \bmod n_2 \end{aligned} \tag{3.22}$$

The first two systems have the trivial solutions $y_1 = 1$ and $y_2 = -1$ which don't differ from those of the quadratic equation $y^2 = 1$ in $\mathbf{Z}$ or a Galois field $\mathrm{GF}(p)$ (i.e. $\mathbf{Z}_p$ with prime p). The last two systems have the non-trivial solutions $y_3 = a$, $y_4 = -a$, as postulated by the *Chinese remainder* *theorem* stating that a system of k simultaneous congruences (i.e. a system of equations of the form $y \equiv a_i \bmod m_i$) with coprime moduli $m_1, \ldots, m_k$ (i.e. $\gcd(m_i, m_j) = 1$ for all $i \neq j$) has a unique solution y with $0 \leq x < m_1 m_2 \ldots m_k$.

3.3.2.2 Finding a Factor

If r is even and $y = \pm a$ with $a \neq 1$ and $a \neq N-1$, then $(a+1)$ or $(a-1)$ must have a common divisor with N because $a^2 \equiv 1 \bmod N$ which means that $a^2 = cN + 1$ with $c \in \mathbf{N}$ and therefore $a^2 - 1 = (a+1)(a-1) = cN$. A factor of N can then be found by using *Euclid's algorithm* to determine $\gcd(N, a+1)$ and $\gcd(N, a-1)$ which is defined as

$$\gcd(a,b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{if } a \bmod b \neq 0 \end{cases} \quad \text{with} \quad a > b \tag{3.23}$$

It can be shown that a random x matches the above mentioned conditions with a probability $p > \frac{1}{2}$ if N is not of the form $N = p^\alpha$ or $N = 2p^\alpha$. Since there are efficient classical algorithms to factorise pure prime powers (and of course to recognise a factor of 2), an efficient probabilistic algorithm for factorisation can be found if the period r of the modular exponentiation can be determined in polynomial time.

3.3.2.3 Period of a Sequence

Let F be quantum function $F : |x, 0\rangle \to |x, f(x)\rangle$ of the integer function $f : \mathbf{Z} \to \mathbf{Z}_{2^m}$ with the unknown period $r < 2^n$.

To determine r, we need two registers, with the sizes of $2n$ and m qubits, which should be reset to $|0, 0\rangle$.

As a first step we produce a homogenous superposition of all base-vectors in the first register by applying an operator U with

$$U|0,0\rangle = \sum_{i=0}^{2^{2n}-1} c_i |i, 0\rangle \quad \text{with} \quad |c_i| = \frac{1}{2^n} \tag{3.24}$$

This can e.g. be achieved by the Hadamard transform H. Applying F to the resulting state gives

$$|\psi\rangle = F\,H\,|0,0\rangle = F\,\frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i, 0\rangle = \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i, f(i)\rangle \tag{3.25}$$

A measurement of the second register with the result $k = f(s)$ with $s < r$ reduces the state to

$$|\psi'\rangle = \sum_{j=0}^{\lceil q/r\rceil-1} c'_j |rj+s,k\rangle \quad \text{with} \quad q=2^{2n} \quad \text{and} \quad c'_j = \sqrt{\left\lceil \frac{r}{q} \right\rceil} \tag{3.26}$$

The post-measurement state $|\psi'\rangle$ of the first register consists only of base-vectors of the form $|rj+s\rangle$ since $f(rj+s)=f(s)$ for all j. It therefore has a discrete, homogenous spectrum.

It is not possible to directly extract the period r or a multiple of it by measurement of the first register because of the random offset s. The result of a Fourier transform, however, is invariant (except for phase factors which don't effect the probability spectrum) to offsets of a periodic distribution.

$$|\tilde{\psi}'\rangle = DFT|\psi'\rangle = \sum_{i=0}^{q-1} \tilde{c}'_i |i,k\rangle \tag{3.27}$$

$$\tilde{c}'_i = \frac{\sqrt{r}}{q} \sum_{j=0}^{p-1} \exp\left(\frac{2\pi \mathrm{i}}{q} i(jr+s)\right) = \frac{\sqrt{r}}{q} \mathrm{e}^{\phi_i} \sum_{j=0}^{p-1} \exp\left(2\pi \mathrm{i}\, \frac{ijr}{q}\right) \tag{3.28}$$

$$\text{with} \quad \phi_i = 2\pi \mathrm{i}\, \frac{is}{q} \quad \text{and} \quad p = \left\lceil \frac{q}{r} \right\rceil$$

If $q=2^{2n}$ is a multiple of r then $\tilde{c}'_i = \mathrm{e}^{\phi_i}/\sqrt{r}$ if i is a multiple of q/r and 0 otherwise. But even if r is not a power of 2, the spectrum of $|\tilde{\psi}'\rangle$ shows distinct peaks with a period of q/r because

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n-1} \mathrm{e}^{2\pi \mathrm{i} k\alpha} = \begin{cases} 1 & \text{if } \alpha \in \mathbf{Z} \\ 0 & \text{if } \alpha \notin \mathbf{Z} \end{cases} \tag{3.29}$$

This is also the reason why we use a first register of $2n$ qubits when $r<2^n$ because it guarantees at least 2^n elements in the above sum and thus a peak width of order $O(1)$.

If we now measure the first register, we will get a value c close to $\lambda q/r$ with $\lambda \in \mathbf{Z}_r$. This can be written as $c/q = c\cdot 2^{-2n} \approx \lambda/r$. We can think of this as finding a rational approximation a/b with $a,b<2^n$ for the fixed point binary number $c\cdot 2^{-2n}$. An efficient classical algorithm for solving this problem using continued fractions is described in [15] and is implemented in the `denominator` function (appendix A.2).

Since the form of a rational number is not unique, λ and r are only determined by $a/b=\lambda/r$ if $\gcd(\lambda,r)=1$. The probability that λ and r are coprime is greater then $1/\ln r$, so only $O(n)$ tries are necessary for a constant probability of success as close at 1 as desired.[3]

[3]If the supposed period $r'=b$ derived form the rational approximation $a/b \approx c\,2^{-2m}$ is odd or $\gcd(x^{r'/2}\pm 1, N)=1$, then one could try to expand a/b by some integer factor k in order to guess the actual period $r=kb$.

3.3.3 QCL Implementation

3.3.3.1 Auxiliary Functions

The implementation of the Shor algorithm uses the following functions:

- `boolean testprime(int n)`
  Tests whether n is a prime number [4]

- `boolean testprimepower(int n)`
  Tests whether n is a prime power

- `int powmod(int x,int a,int n)`
  Calculates $x^a \bmod n$

- `int denominator(real x,int qmax)`
  Returns the denominator q of the best rational approximation $\frac{p}{q} \approx x$ with $p, q < q_{max}$

For the actual implementations of these functions, please refer to appendix A.2.

3.3.3.2 The Procedure shor

The procedure `shor` checks whether the integer `number` is suitable for quantum factorisation, and then repeats Shor's algorithm until a factor has been found.

```
procedure shor(int number) {
  int width=ceil(log(number,2));   // size of number in bits
  qureg reg1[2*width];             // first register
  qureg reg2[width];               // second register
  int qmax=2^width;
  int factor;                      // found factor
  int m; real c;                   // measured value
  int x;                           // base of exponentiation
  int p; int q;                    // rational approximation p/q
  int a; int b;                    // possible factors of number
  int e;                           // e=x^(q/2) mod number

  if number mod 2 == 0 { exit "number must be odd"; }
  if testprime(number) { exit "prime number"; }
  if testprimepower(number) { exit "prime power"; };
```

[4]Since both testfunctions are not part of the algorithm itself, short but inefficient implementations with $O(\sqrt{n})$ have been used

```
  {
    {                             // generate random base
      x=floor(random()*(number-3))+2;
    } until gcd(x,number)==1;
    print "chosen random x =",x;
    Mix(reg1);                    // Hadamard transform
    expn(x,number,reg1,reg2);     // modular exponentiation
    measure reg2;                 // measure 2nd register
    dft(reg1);                    // Fourier transform
    measure reg1,m;               // measure 2st register
    reset;                        // clear local registers
    if m==0 {                     // failed if measured 0
      print "measured zero in 1st register. trying again ...";
    } else {
      c=m*0.5^(2*width);          // fixed point form of m
      q=denominator(c,qmax);      // find rational approximation
      p=floor(q*m*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax { // odd q ? try expanding p/q
        print "odd denominator, expanding by 2";
        p=2*p; q=2*q;
      }
      if q mod 2==1 {             // failed if odd q
        print "odd period. trying again ...";
      } else {
        print "possible period is",q;
        e=powmod(x,q/2,number);   // calculate candidates for
        a=(e+1) mod number;       // possible common factors
        b=(e+number-1) mod number; // with number
        print x,"^",q/2,"+ 1 mod",number,"=",a,",",
              x,"^",q/2,"- 1 mod",number,"=",b;
        factor=max(gcd(number,a),gcd(number,b));
      }
    }
  } until factor>1 and factor<number;
  print number,"=",factor,"*",number/factor;
}
```

3.3.3.3 Factoring 15

15 is the smallest number that can be factorised with Shor's algorithm, as it's the product of smallest odd prime numbers 3 and 5. Our implementation of the modular exponentiation needs $2l+1$ qubits scratch space with $l = \lceil ld(15+1) \rceil = 4$. The algorithm itself needs $3l$ qubits, so a total of 21 qubits must be provided.

```
$ qcl -b21 -i shor.qcl
qcl> shor(15)
: chosen random x = 4
: measured zero in 1st register. trying again ...
: chosen random x = 11
: measured 128 , approximation for 0.500000 is 1 / 2
: possible period is 2
: 11 ^ 1 + 1 mod 15 = 12 , 11 ^ 1 - 1 mod 15 = 10
: 15 = 5 * 3
```

The first try failed because 0 was measured in the first register of $|\psi'\rangle$ and $\lambda/r = 0$ gives no information about the period r.

One might argue that this is not likely to happen, since the first register has 8 qubits and 256 possible base-vectors, however, if a number n is to be factored, one might expect a period about $\sqrt{n}$ assuming that the prime factors of n are of the same order of magnitude. This would lead to a period $\frac{q}{\sqrt{n}}$ after the DFT and the probability $p = \frac{1}{\sqrt{n}}$ to accidentally pick the basevector $|0\rangle$, would be $p = 25.8\%$.

In the special case of a start value $x = 4$ the period of the modular exponentiation is 2 since $4^2 \bmod 15 = 1$, consequently the Fourier spectrum shows 2 peaks at $|0\rangle$ and $|128\rangle$ and $p = 1/2$.

The second try also had the same probability of failure since $11^2 \bmod 15 = 1$, but this time, the measurement picked the second peak in the spectrum at $|128\rangle$. With $128/2^8 = 1/2 = \lambda/r$, the period $r = 2$ was correctly identified and the factors $\gcd(11^{2/2} \pm 1\,,\, 15) = \{3, 5\}$ to 15 have been found.

Bibliography

[1] Paul Benioff *1997 Models of Quantum Turing Machines, LANL Archive quant-ph/9708054*

[2] J.I. Cirac, P. Zoller *1995 Quantum Computations with Cold trapped Ions, Phys. Rev. Lett. 74, 1995 , 4091*

[3] D. Deutsch, *1985 Proceedings of the Royal Society London A 400, 97-117*

[4] J. Gruska, *1998 Foundations of Computing, chap. 12: "Frontiers - Quantum Computing"*

[5] R. W. Keyes *1988 IBM J. Res. Develop. 32, 24*

[6] D. Deutsch *1989 Quantum computational networks. Proceedings of the Royal Society London A 439, 553-558*

[7] D. Coppersmith *1994 An Approximate Fourier Transform Useful in Quantum Factoring, IBM Research Report No. RC19642*

[8] C. H. Bennet *1973 IBM J. Res. Develop. 17, 525*

[9] C. H. Bennet *1989 SIAM J.Comput. 18, 766*

[10] Johannes Buchmann *1996 Faktorisierung großer Zahlen. Spektrum der Wissenschaft 9/96, 80-88*

[11] P.W. Shor. *1994 Algorithms for quantum computation: Discrete logarithms and factoring*

[12] Samuel L. Braunstein *1995 Quantum computation: a tutorial*

[13] David Beckman et al. *1996 Efficient networks for quantum factoring*

[14] Artur Ekert and Richard Jozsa. *1996 Shor's Quantum Algorithm for Factoring Numbers, Rev. Modern Physics 68 (3), 733-753*

[15] G.H. Hardy and E.M. Wright *1965 An Introduction to the Theory of Numbers (4th edition OUP)*

[16] W. Kummer and R. Trausmuth *1988 Skriptum zur Vorlesung 131.869 - Quantentheorie*

[17] B. Oemer *1996 Simulation of Quantum Computers [unpublished]*

List of Tables

Appendix A

QCL Programs and Include Files

A.1 default.qcl

```
extern qufunct Fanout(quconst a,quvoid b);

extern qufunct Swap(qureg a,qureg b);

extern operator Matrix2x2(
  complex u00,complex u01,
  complex u10,complex u11,
qureg q);

extern operator Matrix4x4(
  complex u00,complex u01,complex u02,complex u03,
  complex u10,complex u11,complex u12,complex u13,
  complex u20,complex u21,complex u22,complex u23,
  complex u30,complex u31,complex u32,complex u33,
qureg q);

extern operator Matrix8x8(
  complex u00,complex u01,complex u02,complex u03,
  complex u04,complex u05,complex u06,complex u07,
  complex u10,complex u11,complex u12,complex u13,
  complex u14,complex u15,complex u16,complex u17,
  complex u20,complex u21,complex u22,complex u23,
  complex u24,complex u25,complex u26,complex u27,
  complex u30,complex u31,complex u32,complex u33,
  complex u34,complex u35,complex u36,complex u37,
  complex u40,complex u41,complex u42,complex u43,
  complex u44,complex u45,complex u46,complex u47,
  complex u50,complex u51,complex u52,complex u53,
```

```
  complex u54,complex u55,complex u56,complex u57,
  complex u60,complex u61,complex u62,complex u63,
  complex u64,complex u65,complex u66,complex u67,
  complex u70,complex u71,complex u72,complex u73,
  complex u74,complex u75,complex u76,complex u77,
qureg q);

extern qufunct Perm2(int p0 ,int p1 ,qureg q);

extern qufunct Perm4(int p0 ,int p1 ,int p2 ,int p3 ,qureg q);

extern qufunct Perm8(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
qureg q);

extern qufunct Perm16(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
qureg q);

extern qufunct Perm32(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
  int p16,int p17,int p18,int p19,int p20,int p21,int p22,int p23,
  int p24,int p25,int p26,int p27,int p28,int p29,int p30,int p31,
qureg q);

extern qufunct Perm64(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
  int p16,int p17,int p18,int p19,int p20,int p21,int p22,int p23,
  int p24,int p25,int p26,int p27,int p28,int p29,int p30,int p31,
  int p32,int p33,int p34,int p35,int p36,int p37,int p38,int p39,
  int p40,int p41,int p42,int p43,int p44,int p45,int p46,int p47,
  int p48,int p49,int p50,int p51,int p52,int p53,int p54,int p55,
  int p56,int p57,int p58,int p59,int p60,int p61,int p62,int p63,
qureg q);

extern qufunct Not(qureg q);

extern qufunct CNot(qureg q,quconst c);

extern operator CPhase(real phi,qureg q);

extern operator Rot(real theta,qureg q);

extern operator Mix(qureg q);

extern qufunct ModExp(int n,int x,quconst a,quvoid b);
```

```
boolean bit(int n,int b) {
  return n/2^b mod 2 == 1;
}

qufunct set(int n,qureg q) {
  int i;
  for i=0 to #q-1 {
    if bit(n,i) { Not(q[i]); }
  }
}

const pi=3.141592653589793238462643383279502884197;
```

A.2 functions.qcl

```
set allow-redefines 1;

// returns the smallest factor > 1 of n or 1 if n is prime

int findfactor(int n) {
  int i;
  if n<=0 { exit "findfactor takes only positive args"; }
  for i=2 to floor(sqrt(n)) {
    if n mod i == 0 { return i; }
  }
  return 1;
}

// test if n is a prime number

boolean testprime(int n) {
  int i;
  if n<=1 { return false; }
  for i=2 to floor(sqrt(n)) {
    if n mod i == 0 { return false; }
  }
  return true;
}

// test if n is a prime power

boolean testprimepower(int n) {
  int i;
  int f;
  i=2;
```

```
  while i<=floor(sqrt(n)) and f==0 {
    if n mod i == 0 { f=i; }
    i=i+1;
  }
  for i=2 to floor(log(n,f)) {
    if f^i==n { return true; }
  }
  return false;
}

// returns x^a mod n

int powmod(int x,int a,int n) {
  int u=x;
  int y=1;
  int i;

  for i=0 to 30 {
    if a/2^i mod 2 == 1 { y=y*u mod n; }
    u=u^2 mod n;
  }
  return y;
}

// return the modular inverse to a mod n or 0 if gcd(a,n)>1

int invmod(int a,int n) {
  int b=a;
  int i;

  if gcd(a,n)>1 { return 0; }
  for i=1 to n {
    if b*a mod n == 1 { return b; }
    b=b*a mod n;
  }
  return 0;
}

// finds the denominator q of the best rational approximation p/q
// for x with q<qmax

int denominator(real x,int qmax) {
  real y=x;
  real z;
  int q0;
  int q1=1;
  int q2;

  while true {
```

```
    z=y-floor(y);
    if z<0.5/qmax^2 { return q1; }
    y=1/z;
    q2=floor(y)*q1+q0;
    if q2>=qmax { return q1; }
    q0=q1; q1=q2;
  }
}

set allow-redefines 0;
```

A.3 qufunct.qcl

```
set allow-redefines 1;

// pseudo classic operator to swap bit order

qufunct flip(qureg q) {
  int i;                // declare loop counter
  for i=0 to #q/2-1 {   // swap 2 symmetric bits
    Swap(q[i],q[#q-i-1]);
  }
}

// Conditional Xor

qufunct cxor(quconst a,qureg b,quconst e) {
  int i;
  for i=0 to #a-1 {
    CNot(b[i],a[i] & e);
  }
}

// Conditional multiplexed binary adder for one of 2 classical
// bits and 1 qubit.
// Full adder if #sum=2, half adder if #sum=1.

qufunct muxaddbit(boolean a0,boolean a1,quconst sel,quconst b,qureg sum,quconst e) {
  qureg s=sel;                        // redeclare sel as qureg

  if (a0 xor a1) {                    // a0 and a1 differ?
    if a0  { Not(s); }                // write a into sect qubit
    if #sum>1 {                       // set carry if available
      CNot(sum[1],sum[0] & s & e);
    }
    CNot(sum[0],s & e);               // add a
```

```
    if a0  { Not(s); }                // restore sect qubit
  } else {
    if a0 and a1 {
      if #sum>1 {                     // set carry if available
        CNot(sum[1],sum[0] & e);
      }
      CNot(sum[0],e);                 // add a
    }
  };
                                      // Add qubit b
  if #sum>1 {                         // set carry if available
    CNot(sum[1],b & sum[0]);
  }
  CNot(sum[0],b);                     // add b
}

// conditional multiplexed binary adder for one of 2 integers
// and 1 qureg. No output carry.

qufunct muxadd(int a0,int a1,qureg sel,quconst b,quvoid sum,quconst e) {
  int i;
  for i=0 to #b-2 {                   // fulladd first #b-1 bits
    muxaddbit(bit(a0,i),bit(a1,i),sel,b[i],sum[i:i+1],e);
  }
                                      // half add last bit
  muxaddbit(bit(a0,#b-1),bit(a1,#b-1),sel,b[#b-1],sum[#b-1],e);
}

// Comparison operator. flag is toggled if b<a.
// b gets overwritten. Needs a #b-1 qubit junk register j
// as argument which is left dirty.

qufunct lt(int a,qureg b,qureg flag,quvoid j) {
  int i;
  if bit(a,#b-1) {              // disable further comparison
    CNot(j[#b-2],b[#b-1]);      // and set result flag if
    Not(b[#b-1]);               // MSB(a)>MSB(b)
    CNot(flag,b[#b-1]);
  } else {
    Not(b[#b-1]);               // disable further comparison
    CNot(j[#b-2],b[#b-1]);      // if MSB(a)<MSB(b)
  }
  for i=#b-2 to 1 step -1 {     // continue for lower bits
    if bit(a,i) {               // set new junk bit if undecided
      CNot(j[i-1],j[i] & b[i]);
      Not(b[i]);                // honor last junk bit and
      CNot(flag,j[i] & b[i]);   // set result flag if a[i]>b[i]
    } else {
      Not(b[i]);
```

```
      CNot(j[i-1],j[i] & b[i]);
    }
  }
  if bit(a,0) {
    Not(b[0]);                   // if still undecided (j[0]=1)
    CNot(flag,j[0] & b[0]);      // result is LSB(a)>LSB(b)
  }
}

set allow-redefines 0;
```

A.4 modarith.qcl

```
set allow-redefines 1;

include "functions.qcl";
include "qufunct.qcl";

// conditional addition mod n for 1 integer and 1 qureg
// flag is set if a+b<n for invertability

qufunct addn(int a,int n,quconst b,quvoid flag,quvoid sum,quconst e) {
  qureg s=sum[0\#b-1];
  qureg f=sum[#b-1];
  qureg bb=b;                     // "abuse" sum and b as scratch
  lt(n-a,bb,f,s);                 // for the less-than operator
  CNot(flag,f & e);               // save result of comparison
  !lt(n-a,bb,f,s);                // restore sum and b
  muxadd(2^#b+a-n,a,flag,b,sum,e); // add either a or a-n
}

// Conditional overwriting addition mod n: sum -> (a+sum) mod n

qufunct oaddn(int a,int n,qureg sum,quconst e) {
  qureg j[#sum];
  qureg f[1];

  addn(a,n,sum,f,j,e);            // junk -> a+b mod n
  Swap(sum,j);                    // swap junk and sum
  CNot(f,e);                      // toggle flag
  !addn(n-a,n,sum,f,j,e);         // uncompute b to zero
}

// Conditional Multiplication mod n of an integer a by the qureg b,
// prod <- ab mod n.
```

```
qufunct muln(int a,int n,quconst b,qureg prod,quconst e) {
  int i;

  for i=0 to #prod-1 {
    if bit(a,i) { CNot(prod[i],b[0] & e); }
  }
  for i=1 to #b-1 {
    oaddn(2^i*a mod n,n,prod,b[i] & e);
  }
}

// Conditional Overwriting multiplication mod n: b-> ab mod n

qufunct omuln(int a,int n,qureg b,quconst e) {
  qureg j[#b];

  if gcd(a,n)>1 {
    exit "omuln: a and n have to be relativly prime";
  }
  muln(a,n,b,j,e);
  !muln(invmod(a,n),n,j,b,e);
  cxor(j,b,e);
  cxor(b,j,e);
}

// Modular exponentiation: b -> x^a mod n

qufunct expn(int a,int n,quconst b,quvoid ex) {
  int i;

  Not(ex[0]);                          // start with 1
  for i=0 to #b-1 {
    omuln(powmod(a,2^i,n),n,ex,b[i]);    // ex -> ex*a^2^i mod n
  }
}

set allow-redefines 0;
```

A.5 dft.qcl

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
```

```
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

A.6 shor.qcl

```
include "modarith.qcl";
include "dft.qcl";

procedure shor(int number) {
  int width=ceil(log(number,2));   // size of number in bits
  qureg reg1[2*width];             // first register
  qureg reg2[width];               // second register
  int qmax=2^width;
  int factor;                      // found factor
  int m; real c;                   // measured value
  int x;                           // base of exponentiation
  int p; int q;                    // rational approximation p/q
  int a; int b;                    // possible factors of number
  int e;                           // e=x^(q/2) mod number

  if number mod 2 == 0 { exit "number must be odd"; }
  if testprime(number) { exit "prime number"; }
  if testprimepower(number) { exit "prime power"; };

  {
    {                              // generate random base
      x=floor(random()*(number-3))+2;
    } until gcd(x,number)==1;
    print "chosen random x =",x;
    Mix(reg1);                     // Hadamard transform
    expn(x,number,reg1,reg2);      // modular exponentiation
    measure reg2;                  // measure 2nd register
    dft(reg1);                     // Fourier transform
    measure reg1,m;                // measure 2st register
    reset;                         // clear local registers
    if m==0 {                      // failed if measured 0
      print "measured zero in 1st register. trying again ...";
    } else {
      c=m*0.5^(2*width);           // fixed point form of m
      q=denominator(c,qmax);       // find rational approximation
      p=floor(q*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax { // odd q ? try expanding p/q
```

```
        print "odd denominator, expanding by 2";
        p=2*p; q=2*q;
      }
      if q mod 2==1 {              // failed if odd q
        print "odd period. trying again ...";
      } else {
        print "possible period is",q;
        e=powmod(x,q/2,number);    // calculate candidates for
        a=(e+1) mod number;        // possible common factors
        b=(e+number-1) mod number; // with number
        print x,"^",q/2,"+ 1 mod",number,"=",a,",",
              x,"^",q/2,"- 1 mod",number,"=",b;
        factor=max(gcd(number,a),gcd(number,b));
      }
    }
  } until factor>1 and factor<number;
  print number,"=",factor,"*",number/factor;
}
```

Appendix B

QCL Charts

B.1 Syntax

B.1.1 Expressions

$$
\begin{array}{rcl}
\textit{complex-coord} & \leftarrow & [\,\texttt{+}\,|\,\texttt{-}\,]\ \textit{digit}\ \{\,\textit{digit}\,\}\,[\ \texttt{.}\ \{\,\textit{digit}\,\}] \\
\textit{const} & \leftarrow & \textit{digit}\ \{\,\textit{digit}\,\}\,[\ \texttt{.}\ \{\,\textit{digit}\,\}] \\
 & \leftarrow & \texttt{(}\ \textit{complex-coord}\ \texttt{,}\ \textit{complex-coord}\ \texttt{)} \\
 & \leftarrow & \texttt{true}\,|\,\texttt{false} \\
 & \leftarrow & \texttt{"}\ \{\,\textit{char}\,\}\ \texttt{"} \\
\textit{expr} & \leftarrow & \textit{const} \\
 & \leftarrow & \textit{identifier}\ [\ \texttt{[}\ \textit{expr}\ [(\ \texttt{:}\ |\ \texttt{..}\ )\ \textit{expr}]\ \texttt{]}\ ] \\
 & \leftarrow & \textit{identifier}\ \texttt{(}\ [\,\textit{expr}\ \{\ \texttt{,}\ \textit{expr}\,\}]\ \texttt{)} \\
 & \leftarrow & \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & \leftarrow & \texttt{\#}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,\texttt{\^{}}\ \textit{expr} \\
 & \leftarrow & \texttt{-}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{*}\,|\,\texttt{/}\,)\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{mod}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{+}\,|\,\texttt{-}\,|\,\texttt{\&}\,)\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{==}\,|\,\texttt{!=}\,|\,\texttt{<}\,|\,\texttt{<=}\,|\,\texttt{>}\,|\,\texttt{>=}\,)\ \textit{expr} \\
 & \leftarrow & \texttt{not}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{and}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{or}\,|\,\texttt{xor}\,)\ \textit{expr}
\end{array}
$$

B.1.2 Statements

$$
\begin{array}{rcl}
\mathit{block} & \leftarrow & \texttt{\{}\ \mathit{stmt}\ \{\ \mathit{stmt}\ \}\ \texttt{\}} \\
\mathit{option} & \leftarrow & \mathit{letter}\ \{\ \mathit{letter} \mid \texttt{-}\ \} \\
\mathit{stmt} & \leftarrow & [\ \texttt{!}\ ]\ \mathit{identifier}\ \texttt{(}\ [\ \mathit{expr}\ \{\ \texttt{,}\ \mathit{expr}\ \}]\ \texttt{)}\ \texttt{;} \\
 & \leftarrow & \mathit{identifier}\ \texttt{=}\ \mathit{expr}\ \texttt{;} \\
 & \leftarrow & \mathit{expr}\ (\ \texttt{->} \mid \texttt{<-} \mid \texttt{<->}\ )\ \mathit{expr}\ \texttt{;} \\
 & \leftarrow & \texttt{for}\ \mathit{identifier}\ \texttt{=}\ \mathit{expr}\ \texttt{to}\ \mathit{expr}\ [\ \texttt{step}\ \mathit{expr}\ ]\ \mathit{block} \\
 & \leftarrow & \texttt{while}\ \mathit{expr}\ \mathit{block} \\
 & \leftarrow & \mathit{block}\ \texttt{until}\ \mathit{expr}\ \texttt{;} \\
 & \leftarrow & \texttt{if}\ \mathit{expr}\ \mathit{block}\ [\ \texttt{else}\ \mathit{block}\ ] \\
 & \leftarrow & \texttt{return}\ \mathit{expr}\ \texttt{;} \\
 & \leftarrow & \texttt{input}\ [\ \mathit{expr}\ ]\ \texttt{,}\ \mathit{identifier}\ \texttt{;} \\
 & \leftarrow & \texttt{print}\ \mathit{expr}\ [\ \texttt{,}\ \mathit{expr}\ ]\ \texttt{;} \\
 & \leftarrow & \texttt{exit}\ [\ \mathit{expr}\ ]\ \texttt{;} \\
 & \leftarrow & \texttt{measure}\ \mathit{expr}\ [\ \texttt{,}\ \mathit{identifier}\ ]\ \texttt{;} \\
 & \leftarrow & \texttt{reset}\ \texttt{;} \\
 & \leftarrow & \texttt{dump}\ [\ \mathit{expr}\ ]\ \texttt{;} \\
 & \leftarrow & \texttt{list}\ [\ \mathit{identifier}\ \{\ \texttt{,}\ \mathit{identifier}\ \}]\ \texttt{;} \\
 & \leftarrow & (\ \texttt{load} \mid \texttt{save}\ )\ [\ \mathit{expr}\ ]\ \texttt{;} \\
 & \leftarrow & \texttt{shell}\ \texttt{;} \\
 & \leftarrow & \texttt{set}\ \mathit{option}\ [\ \texttt{,}\ \mathit{expr}\ ]\ \texttt{;} \\
 & \leftarrow & \mathit{stmt}\ \texttt{;}
\end{array}
$$

B.1.3 Definitions

$$
\begin{array}{rcl}
\mathit{type} & \leftarrow & \texttt{int} \mid \texttt{real} \mid \texttt{complex} \mid \texttt{string} \\
 & \leftarrow & \texttt{qureg} \mid \texttt{quvoid} \mid \texttt{quconst} \mid \texttt{quscratch} \\
\mathit{const\text{-}def} & \leftarrow & \texttt{const}\ \mathit{identifier}\ \texttt{=}\ \mathit{expr}\ \texttt{;} \\
\mathit{var\text{-}def} & \leftarrow & \mathit{type}\ \mathit{identifier}\ \texttt{[}\ \mathit{expr}\ \texttt{]}\ \texttt{;} \\
 & \leftarrow & \mathit{type}\ \mathit{identifier}\ [\ \texttt{=}\ \mathit{expr}\ ]\ \texttt{;} \\
\mathit{arg\text{-}def} & \leftarrow & \mathit{type}\ \mathit{identifier} \\
\mathit{arg\text{-}list} & \leftarrow & \texttt{(}\ [\ \mathit{arg\text{-}def}\ \{\ \texttt{,}\ \mathit{arg\text{-}def}\ \}]\ \texttt{)} \\
\mathit{body} & \leftarrow & \texttt{\{}\ \{\ \mathit{const\text{-}def} \mid \mathit{var\text{-}def}\ \}\ \{\ \mathit{stmt}\ \}\ \texttt{\}}
\end{array}
$$

def	←	*const-def* | *var-def*
	←	*type identifier arg-list body*
	←	`procedure` *identifier arg-list body*
	←	`operator` *identifier arg-list body*
	←	`qufunct` *identifier arg-list body*
	←	`extern operator` *identifier arg-list* ;
	←	`extern qufunct` *identifier arg-list* ;

B.2 Error Messages

B.2.1 Typecheck Errors

invalid definition: Invalid length
invalid definition: Length cannot be specified with this type
invalid definition: Quantum variable must be allocated or initialised
invalid definition: Quantum variable can either be allocated or defined as reference, not both

external error: external routine *identifier* not found

internal error: Invalid binary operator
internal error: Invalid unary operator
internal error: Local call within non subroutine definition
internal error: alloc failed
internal error: can't add symbol *identifier* to symtab
internal error: can't add symbol to symtab
internal error: no *operator* operator defined
internal error: uncaught include
internal error: unknown base function
internal error: unknown list function

invalid parameter: Functions may not depend on quantum parameters
invalid parameter: duplicate parameter

invalid type: Can only list symbols
invalid type: Filename must be a string
invalid type: Length operator is only defined for quantum expressions
invalid type: Modulus arguments must be integer
invalid type: Negation: argument must be boolean

invalid type: Non numeric argument to binary arithmetic operator
invalid type: Quantum expression required
invalid type: Unary Minus: argument must be number
invalid type: assignment to quantum variable
invalid type: bit selection not on quantum variable
invalid type: bit selection on on quantum variable
invalid type: can't compare boolean values
invalid type: can't compare quantum expressions
invalid type: cannot input quantum variables
invalid type: comparison operator with unordered type
invalid type: comparison type mismatch
invalid type: concatenation of invalid types
invalid type: constant target register for *operator*
invalid type: for loop parameters must be integer
invalid type: input prompt is no string
invalid type: local scratch registers can't be used with qureg arguments
invalid type: logical operator with non boolean arguments
invalid type: quantum eigenstates are integers
invalid type: quantum expression as option argument
invalid type: quantum state required
invalid type: quantum variables may not be accessed within functions
invalid type: subrange parameters not integer
invalid type: subscript not integer
invalid type: *operator* is only defined on quantum expressions

option error: illegal option *option*
option error: missing argument for option *option*
option error: option *option* takes no argument

parameter mismatch: quconst used as non-const argument to *identifier*
parameter mismatch: unmatching argument number
parameter mismatch: unmatching argument types

illegal scope: Calls not allowed within functions
illegal scope: Global symbol *identifier* already defined
illegal scope: Local symbol *identifier* already defined
illegal scope: Procedure call within operator
illegal scope: Quantum Variables may non be defined within functions
illegal scope: Reset of quantum state is not allowed in this scope
illegal scope: Void Registers have to be arguments
illegal scope: error message is not a string

illegal scope: function random is not allowed in this scope
illegal scope: input not allowed in this scope
illegal scope: measurement is not allowed in this scope
illegal scope: operator called within qufunct
illegal scope: return statement outside function
illegal scope: subroutines may only be defined in global scope
illegal scope: Scratch Space may only be allocated within quantum functions

unknown symbol: Invalid Symbol type
unknown symbol: Undefined function *identifier*
unknown symbol: Undefined procedure or operator *identifier*
unknown symbol: Unknown local variable *identifier*
unknown symbol: Unknown local variable *identifier*
unknown symbol: Unknown variable *identifier*
unknown symbol: Unknown variable or constant *identifier*

syntax error: function random takes no arguments
syntax error: invalid number of argument to log
syntax error: missing argument

type mismatch: Invalid initialisation
type mismatch: Invalid initialiser for quantum register reference
type mismatch: Quantum variables cannot be defined as constant
type mismatch: argument is not a number
type mismatch: argument of unordered type
type mismatch: integer required
type mismatch: invalid assignment
type mismatch: invalid log argument
type mismatch: not boolean if condition
type mismatch: not boolean loop condition
type mismatch: ordered type required
type mismatch: return expression doesn't match function type

B.2.2 Evaluation Errors

general error: function didn't return a value

internal error: argument binding failed
internal error: eval is not implemented on class sExpr

internal error: invalid return value

math error: 0^0 is undefined
math error: division by zero
math error: negative base in non integer real value power
math error: negative exponent in integer power
math error: real logarithm of non positive number
math error: real square root of negative number

range error: invalid quantum subregister
range error: invalid qubit subscript
range error: quantum registers overlap

B.2.3 Execution Errors

internal error: Measurement failed
internal error: argument binding failed
internal error: cannot store constant
internal error: cannot store routine
internal error: cannot store variable
internal error: invalid quantum parameter
internal error: parameter *identifier* not found
internal error: reset failed
internal error: temporary register not found
internal error: uncaught include
internal error: undefined sTrans object

I/O-error: Can't close *filename*
I/O-error: Can't open *filename* for reading
I/O-error: Can't open *filename* for writing
I/O-error: Error while reading *filename*
I/O-error: Error writing reading *filename*

math error: Measured Integer is too long

memory error: can't allocate internal scratch space
memory error: not enough quantum memory
memory error: quantum heap is corrupted
memory error: quantum register of non-positive length
memory error: void or scratch register not empty

option error: illegal option *option*
option error: missing argument for option *option*
option error: option *option* takes no argument

runtime error: infinite for loop
runtime error: quantum arguments overlapping
runtime error: zero increment in for loop
runtime error: *operator* arguments are of different length

user error: *message*