Structured Quantum Programming

Bernhard Ömer

first version
26th May 2003

last revision
2nd September 2009

Institute for Theoretical Physics

Vienna University of Technology

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

Contents

Preface

In contrast to quantum circuits, quantum Turing machines or the algebraic definition of unitary transformations, programming languages allow the complete and constructive description of quantum algorithms including their classical control structure for arbitrary input sizes and hardware architectures.

This thesis investigates how the classical formalism of structured programming can be adapted to the field of quantum computing, based on the machine model of a universal computer with a quantum oracle allowing the application of unitary gates and the measurement of single qubits. Starting with the abstract notion of programs as finite automatons *(finite programs)* and in analogy to classical programming languages, the concept of structured quantum programming languages (QPLs) is developed and illustrated by the experimental language QCL.

A QPL is called *imperative* if it provides quantum registers *(quantum variables)*, elementary gates and single qubit measurements. A *procedural* QPL additionally supports unitary subroutines and non-classical concepts like the reverse execution of code to derive the adjoint operator or the unitary "uncomputing" of temporary quantum registers *(scratch space management)*. A procedural QPL is called *structured* it also allows the use of qubits and boolean expression of qubits *(quantum conditions)* in structured flow-control statements *(quantum if-statement)*.

A QCL interpreter for the Linux operating system as well as a numerical simulator for arbitrary quantum oracles are available as free software from

`http://www.itp.tuwien.ac.at/~oemer/qcl.html`

Overview

Chapter 1 gives a general introduction to quantum computing and describes the key concepts and formalism necessary for the discussion of QPLs.

After a short historical overview (1.1), the formalism and the postulates of quantum mechanics are presented in section 1.2. Section 1.3 introduces the key concepts of classical computation and develops a formal notion of machines and programs which differs from the usual formalizations by treating machines and programs as separate entities. In section 1.4 the abstract machine concept is applied to quantum computing and the main components of a quantum computer, namely qubit-registers, unitary gates and qubit measurements, are discussed using a new formalism called *register notation* which allows the compact and abstract description of quantum circuits. Finally, section 1.5 discusses the formal description and the design of quantum algorithms.

Chapter 2 presents the concept of structured quantum programming languages as a new formalism for quantum computing.

After a general introduction to classical and quantum programming languages (2.1), section 2.2 discusses universal quantum computers and introduces the hybrid quantum architecture as the computational model of quantum programming. After an introduction to the key elements of classical structured programming languages (2.3), the remainder of chapter 2 demonstrates how these concepts can be adapted to quantum computing: Section 2.4 discusses the minimal requirements for a universal QPL *(imperative quantum programming)*, section 2.5 introduces unitary subroutine *(procedural quantum programming)* and, finally, section 2.6 demonstrates how conditional operators can be used to realize the semantics of conditional branching on qubits and quantum conditions *(structured quantum programming)*.

Chapter 1

Quantum Computing

1.1 The Way to Quantum Computing

1.1.1 From Huygens to Planck

Before the adoption of quantum theory, one of the main problems of what now is referred to as classical physics was the dual nature of light. While its linear propagation and the lack of a physical medium seemed to suggest a particle-like behavior, phenomena like interference and diffraction are well known properties of waves.

In 1690, Christiaan Huygens explained optical birefringence in his *Traité de la lumière* where he developed a comprehensive wave-theory of light. [35] 14 years later, Isaac Newton published his *Opticks* in which he explained phenomena like reflection, dispersion, color and polarization by interpreting light as a stream of differently sized particles.

The corpuscular-theory of light dominated the scientific discussion until the beginning of the 19$^{\text{th}}$ century when Young and Fresnel demonstrated several shortcomings of the theory which can be resolved assuming transversal light-waves in a universal medium called *Ether*. In 1873 in his *Treatise on Electricity and Magnetism*, Maxwell published a set of 4 partial differential equations which lay the foundation to classical electrodynamics and elegantly explains light as electromagnetic waves. Maxwell's theory however was still unable to explain the radiation of black bodies as well the the discrete energy spectra of atoms. Both shortcomings would prove crucial in the development of quantum theory.

1.1.2 The Century of Quantum Physics

Classical electodynamics predict that the energy distribution in a cavity – and therefore the spectrum of a black body – is proportional to the number of vibrational modes, which leads to an energy density of

$$U_\lambda(T) = 8\pi kT\lambda^{-4},$$

known as *Rayleigh-Jeans Law*, which is not integrable. [14]

In the year 1900, Max Planck found a way to avoid this contradiction, which is also known as *ultraviolet catastrophe*, by the ad-hoc assumption, that the possible energy states are restricted to $E = nh\nu$, where n is an integer, ν the frequency and h the Planck constant, the fundamental constant of quantum physics, with a value of

$$h = 2\pi\hbar = 6.626075 \cdot 10^{-34}\text{Js}$$

This restriction causes the probability of frequencies $\nu \gg kT/h$ to decrease exponentially and leads to the integrable distribution

$$U_\nu(T) = \frac{8\pi\nu^2}{c^3}\frac{h\nu}{\mathrm{e}^{h\nu/kT} - 1}$$

which is also in exact accordance with the experimental data.

Five years later, Albert Einstein explained the photo-electric effect by postulating the existence of light particles, later called *photons*, with the energy $E = h\nu$.

In 1913, Niels Bohr calculated the value of the Rydberg constant, by assuming that the angular momentum of electrons orbiting the nucleus satisfies the quantization condition $L = n\hbar = nh/2\pi$. This restriction could be justified by attributing wave properties to the electron and demanding that their corresponding wave functions form a standing wave; however this kind of hybrid theory remained unsatisfactory.

A complete solution for the problem came in 1923 from Werner Heisenberg who used a matrix-based formalism; two years later Erwin Schrödinger published an equivalent solution using complex wave functions.

In 1927, Heisenberg formulated the uncertainty principle, which formalized the *complementarity* of the wave and the particle picture, claimed by Bohr which, while being mutually exclusive, are both essential for a complete description of quantum events. Together with a statistical interpretation of Schrödinger's wave function, they lay the theoretical foundation for the *Copenhagen interpretation* of quantum mechanics. [17]

> *"We regard quantum mechanics as a complete theory for which the fundamental physical and mathematical hypotheses are no longer susceptible of modification."*
>
> Werner Heisenberg and Max Born, Solvay Congress of 1927

While its explanation of quantum phenomena like entanglement or measurement still seems somewhat unsatisfactory, even after 75 years, the Copenhagen interpretation can still be regarded as the mainstream in quantum physics. Apparent contradictions like the famous EPR paradox [29] have not only been verified by experiment, but also serve as fundamental principles for new fields of research like quantum cryptography and quantum computing.

1.1.3 Beyond the Church-Turing Thesis

The basic idea of modern computing science is the view of computation as a mechanical, rather than a purely mental process. In 1936, Alan Turing formalized this concept by constructing an abstract device, now called *Turing-Machine*, which he proved to be capable of performing any effective (i.e. mechanical, algorithmic) computation. At about the same time, Alonzo Church showed that any function of positive integers is effectively calculable only if recursive. Both findings are, in fact, equivalent and are commonly referred to as the *Church-Turing Thesis.* In its strong form, it can be summarized as

> *Any algorithmic process can be simulated efficiently using a Turing machine*

This means that, no matter what type of machine is actually used for a certain computation, an equivalent Turing Machine can be found which solves the same problem with only polynomial overhead.

The strong Church-Turing Thesis came under attack when in 1977 Robert Solovay and Volker Strassen published a fast Monte-Carlo test for primality [55, 43], a problem for which no efficient deterministic algorithm was known at that time.[1] While this challenge could easily be resolved by using a probabilistic Turing Machine, it raises the question whether even more powerful models of computation exists.

In 1985, David Deutsch adopted a more general approach and tried to develop an abstract machine, the Universal Quantum Computer, which is not targeted at some formal notion of computability, but should be capable

[1] In 2002, Manindra Agrawal, Neeraj Kayal and Nitin Saxena eventually found a deterministic primality test [40] with a worst case time complexity of $O(n^{12})$.

of effectively simulating an arbitrary *physical system* and consequently any realizable computational device [24, 56]. Deutsch also described a simple quantum algorithm which would be capable of determining in a single step whether a given one-bit oracle function $f : \mathbf{B} \to \mathbf{B}$ is either constant or balanced. The algorithm was later generalized for n-bit functions $f : \mathbf{B}^n \to \mathbf{B}$ (Deutsch-Jozsa problem [26]) and demonstrates that a quantum computer is indeed more powerful than a probabilistic Turing machine.

At the same time, Richard Feynman showed how local Hamiltonians can be constructed to perform arbitrary classical computations [31].

In 1994, Peter Shor demonstrated how prime factorization and the calculation of the discrete logarithm could be efficiently performed on a quantum computer [54]. The immense practical importance of these problems for cryptography made Shor's algorithm the "killer-application" of quantum computing.

One year later, Lov Grover designed a quantum algorithm for finding a unique solution to $Q(x) = 1$ in an unstructured search space of size n, requiring only $O(\sqrt{n})$ evaluations of the black-box oracle function Q [32].

At this time, Peter Zoller and Ignacio Cirac demonstrated how a linear ion trap can be used to store qubits and perform quantum computations [19]. In 2001, a team at IBM succeeded to implement Shor's algorithm on an NMR based 7-qubit quantum computer to factorize the number 15 [18].

1.2 Quantum Mechanics

1.2.1 Quantum Computation as Quantum Mechanical Theory

Strictly spoken, the algebraic formulation of quantum mechanics, which shall be introduced in this section, is not a physical theory in its own right, but rather provides a framework to formulate physical theories within. Depending on how exactly the Hilbert spaces and Hamiltonians are constructed, different theories emerge, from non-relativistic quantum electrodynamics, which still maintains many formal analogies to classical physics, to quantum chromodynamics which introduces entities like quarks and gluons which are completely meaningless outside the scope of quantum mechanics.

Quantum computing is yet another theory on top of the abstract quantum mechanical formalism. It is, however, not a physical theory in the sense that it tries to accurately describe natural processes, but is built on abstract concepts like qubits and quantum gates, without regard to the underlying physical quantum-dynamical model.

This top-down approach is at the same time the greatest strength and the greatest weakness of quantum computation. While it guarantees that its computational model is in fact the most general which is physically realizable in a quantum mechanical universe, the lack of a concrete and scalable "reference implementation", like the Turing machine is for classical computing, leaves open the question whether quantum computers with more than a handful of qubits are in fact possible, under realistic assumptions for noise and experimental accuracy.

1.2.2 Linear Algebra

1.2.2.1 Braket Notation

The "braket" notation is a very compact formalism for linear algebra and was introduced by Dirac. Table 1.1 lists the most commonly used expressions.

Notation	Description
$\|\psi\rangle$	general "ket" vector, e.g. $\|\psi\rangle = (c_0, c_1, \ldots)^T$
$\langle\psi\|$	dual "bra" vector to $\|\psi\rangle$, e.g. $\langle\psi\| = (c_0^*, c_1^*, \ldots)$
$\|n\rangle$	n^{th} basis vector of some standard basis $N = \{\|0\rangle, \|1\rangle, \ldots\}$
$\|\tilde{n}\rangle$	basis vector of an alternate basis $\tilde{N} = \{\|\tilde{0}\rangle, \|\tilde{1}\rangle, \ldots\}$
$\langle\phi\|\psi\rangle$	inner product of $\|\phi\rangle$ and $\|\psi\rangle$
$\|\phi\rangle \otimes \|\psi\rangle$	tensor product of $\|\phi\rangle$ and $\|\psi\rangle$
$\|\phi\rangle\|\psi\rangle$	abbreviated tensor product $\|\phi\rangle \otimes \|\psi\rangle$
$\|i, j\rangle$	abbreviated tensor product of the basis vectors $\|i\rangle$ and $\|j\rangle$
$M^\dagger$	adjoint operator (matrix) $M^\dagger = (M^T)^*$
$\langle\phi\|M\|\psi\rangle$	inner product of $\|\phi\rangle$ and $M\|\psi\rangle$
$\\|\psi\\|$	abbreviated norm $\\|\|\psi\rangle\\|$

Table 1.1: *Dirac Notation*

1.2.2.2 Hilbert Space

Definition 1 *A set* $\mathbf{V}$ *is called vector space over a scalar field* $\mathbf{F}$ *iff the operations* $+ : \mathbf{V} \times \mathbf{V} \to \mathbf{V}$ *(vector addition) and* $\cdot : \mathbf{F} \times \mathbf{V} \to \mathbf{V}$ *(scalar multiplication) are defined, and*

(i) $(\mathbf{V}, +)$ *is a commutative group,*

(ii) $\lambda|\psi\rangle = |\psi\rangle\lambda$,

(iii) $\lambda(\mu|\psi\rangle) = (\lambda\mu)|\psi\rangle$,

(iv) $(\lambda + \mu)|\psi\rangle = \lambda|\psi\rangle + \mu|\psi\rangle$,

(v) $\lambda(|\psi\rangle + |\phi\rangle) = \lambda|\psi\rangle + \lambda|\phi\rangle$.

From now on, we will only consider complex vector spaces, i.e. $\mathbf{F} = \mathbf{C}$.

Definition 2 *Let* $\mathbf{V}$ *be a complex vector space. A function* $\langle\cdot|\cdot\rangle : \mathbf{V}\times\mathbf{V} \to \mathbf{C}$ *is called inner product iff*

(i) $\langle\psi|(\lambda|\phi\rangle + \mu|\chi\rangle) = \lambda\langle\psi|\phi\rangle + \mu\langle\psi|\phi\rangle$,

(ii) $\langle\psi|\phi\rangle = \langle\phi|\psi\rangle^*$,

(iii) $\langle\psi|\psi\rangle \in \mathbf{R}$, $\langle\psi|\psi\rangle \geq 0$, $\langle\psi|\psi\rangle = 0 \Leftrightarrow |\psi\rangle = \mathbf{o}$.

An inner product also defines the norm $\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle}$ (also written as $\|\psi\|$). The following inequalities apply:

$$|\langle\psi|\phi\rangle| \leq \|\psi\|\|\phi\| \qquad \textit{(Schwarz inequality)} \tag{1.1}$$

$$\||\psi\rangle + |\phi\rangle\| \leq \|\psi\| + \|\phi\| \qquad \textit{(triangle inequality)} \tag{1.2}$$

Definition 3 (Completeness) *Let* $\mathbf{V}$ *be a vector space with the norm* $\|\cdot\|$ *and* $|\psi_n\rangle \in \mathbf{V}$ *a sequence of vectors.*

(i) $|\psi_n\rangle$ *is a Cauchy sequence iff* $\forall\epsilon > 0\ \exists N > 0$ *such that*

$$\forall n, m > N, \||\psi_n\rangle - |\psi_m\rangle\| < \epsilon \tag{1.3}$$

(ii) $|\psi_n\rangle$ *is convergent iff there is a* $|\psi\rangle \in \mathbf{V}$ *such that*

$$\forall\epsilon > 0\ \exists N > 0\ \forall n > N, \||\psi_n\rangle - |\psi\rangle\| < \epsilon \tag{1.4}$$

$\mathbf{V}$ *is complete iff every Cauchy sequence converges.*

Definition 4 *A complete vector space* $\mathcal{H}$ *with an inner product* $\langle\cdot|\cdot\rangle$ *and the corresponding norm* $\|\psi\| = \sqrt{\langle\psi|\psi\rangle}$ *is called Hilbert space.*

A Hilbert space $\mathcal{H}$ is *separable* if there exists an enumerable set $\mathcal{S} \subseteq \mathcal{H}$ which is dense in $\mathcal{H}$, i.e. for any $|\psi\rangle \in \mathcal{H}$ and $\epsilon > 0$ there exists a $|\sigma\rangle \in \mathcal{S}$ with $\||\psi\rangle - |\sigma\rangle\| < \epsilon$. From now on, we will only consider separable Hilbert spaces.

A vector $|\psi\rangle \in \mathcal{H}$ is *normalized* or a *unit-vector* iff $\|\psi\| = 1$. An enumerable set of normalized vectors $B = \{|\psi_0\rangle, |\psi_0\rangle, \ldots\}$ is called *orthonormal* *system* iff $\langle\psi_i|\psi_j\rangle = \delta_{ij}$. An orthonormal system B is an *(orthonormal) basis* of $\mathcal{H}$ iff any vector $|\psi\rangle \in \mathcal{H}$ can be written as

$$|\psi\rangle = \sum_i \lambda_i |\psi_i\rangle \quad \text{with} \quad |\psi_i\rangle \in B$$

Since Hilbert spaces are complete by definition, any separable, complex Hilbert space $\mathcal{H}$ with some basis B is algebraically isomorphic and isometric to either

(i) $\mathbf{C}^n$ with the basis $\{|0\rangle, |1\rangle, \ldots |n-1\rangle\}$ with $|k\rangle = (\delta_{0k}, \delta_{1k}, \ldots \delta_{n-1,k})^T$ if $\dim \mathcal{H} = |B| = n$ or

(ii) $\mathbf{l}_2$ (i.e. the space of complex sequences $\langle\xi_n\rangle$ for which $\sum_k |\xi_k|^2$ is defined) with the basis vectors $|k\rangle = (\delta_{0k}, \delta_{1k}, \ldots)^T$ if $\dim \mathcal{H} = |B| = \aleph_0$

In quantum computing we usually deal with finite dimensional Hilbert spaces, so unless otherwise noted we will always assume $\mathcal{H} = \mathbf{C}^n$. In $\mathbf{C}^n$, a "ket" vector $|\psi\rangle$ can be written as column vector and the dual "bra" vector $\langle\psi|$ can be written as row vector $\langle\psi| = (|\psi\rangle^*)^T$, which allows the inner product $\langle\cdot|\cdot\rangle$ to be formally expressed as ordinary matrix multiplication.

Definition 5 *Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be Hilbert spaces with the bases B_1 and B_2, then the tensor product*

$$\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2 = \Big\{ \sum_{|i\rangle \in B_1} \sum_{|j\rangle \in B_2} c_{ij} |i,j\rangle \;\Big|\; c_{ij} \in \mathbf{C} \Big\} \tag{1.5}$$

is also a Hilbert space with the basis $B = B_1 \times B_2$ and the inner product

$$\langle i,j|i',j'\rangle = \langle i|i'\rangle\langle j|j'\rangle = \delta_{ii'}\delta_{jj'} \quad \text{with} \quad |i\rangle, |i'\rangle \in B_1 \quad \text{and} \quad |i\rangle, |i'\rangle \in B_2$$

.

1.2.2.3 Linear Operators

Definition 6 *Let $\mathbf{V}$ be a vector space and A be function $A : \mathbf{V} \to \mathbf{V}$. A is called linear operator on $\mathbf{V}$ iff*

$$A(\lambda|\psi\rangle + \mu|\phi\rangle) = \lambda A(|\psi\rangle) + \mu A(|\phi\rangle) = \lambda A|\psi\rangle + \mu A|\phi\rangle. \tag{1.6}$$

In $\mathbf{C}^n$, a linear operator A can be written as a $n \times n$ matrix with the matrix elements $a_{ij} = \langle i|A|j\rangle$

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{pmatrix} = \sum_{i,j} a_{ij} |i\rangle\langle j| \tag{1.7}$$

Because of (1.6), a linear operator on a vector space $\mathbf{V}$ with the basis B is completely defined by its effect on the basis vectors, so the above operator A could also be written as

$$A : |n\rangle \to \sum_k a_{kn}|k\rangle \quad \text{with} \quad |k\rangle \in B \tag{1.8}$$

Definition 7 *The operator* $A^\dagger = (A^T)^* = \sum_{i,j} a_{ji}{}^*|i\rangle\langle j|$ *is called adjoint operator of* A*.*

Definition 8 *A linear operator* A *is called*

(i) *normal iff* $A^\dagger A = AA^\dagger$*,*

(ii) *self-adjoint or Hermitian iff* $A^\dagger = A$*,*

(iii) *positive iff* $\langle\psi|A|\psi\rangle \in \mathbf{R}_0^+ \,\forall\, |\psi\rangle \in \mathcal{H}$*,*

(iv) *unitary iff* $A^\dagger A = I$*, with* I *being the identity operator,*

(v) *idempotent iff* $A^2 = A$*,*

(vi) *self-inverse iff* $A^2 = I$*,*

(vii) *an (orthogonal) projection iff* A *is self-adjoint and idempotent.*

$SU(n)$ is the group of unitary operators on $\mathbf{C}^n$ with determinant 1. Since for each unitary U on $\mathbf{C}^n$ there exists a physically equivalent $U' = \mathrm{e}^{\mathrm{i}\varphi}U \in SU(n)$ (see 1.2.3.1), we will also use $SU(n)$ to denote any set of operators physically equivalent to $SU(n)$.

Definition 9 *An* $a \in \mathbf{C}$ *with at least one non-zero solution* $|a\rangle$ *of the equation* $A|a\rangle = a|a\rangle$ *is called eigenvalue of* A*, with* $|a\rangle$ *being an eigenvector for* a*. The set* $\{|\psi\rangle \in \mathcal{H} | A|\psi\rangle = a|\psi\rangle\}$ *is known as the eigenspace of* A *for the eigenvalue* a*.*

Any linear operator A can be written in terms of its eigenvectors as

$$A = \sum_i a_i |\tilde{\imath}\rangle\langle\tilde{\imath}| \quad \text{with} \quad \langle\tilde{\imath}|\tilde{\jmath}\rangle = \delta_{ij}. \tag{1.9}$$

This form is called *spectral decomposition* of A.

Definition 10 *Let* A *be a linear operator on* $\mathcal{H}_1$ *and* B *a linear operator on* $\mathcal{H}_2$*, then the tensor product*

$$A \otimes B = \sum_{i,j}\sum_{i',j'} |i,j\rangle\langle i|A|i'\rangle\langle j|B|j'\rangle\langle i',j'| \tag{1.10}$$

is a linear operator on $\mathcal{H}_1 \otimes \mathcal{H}_2$*.*

Definition 11 *Let* A *and* B *be linear operators on* $\mathcal{H}$*. The operator* $[A,B] = AB - BA$ *is called commutator and* $\{A,B\} = AB + BA$ *is called anti-commutator of* A *and* B*.*

1.2.3 The Postulates of Quantum Mechanics

1.2.3.1 Quantum States

Postulate 1 *Associated to any physical system S is a complex Hilbert space $\mathcal{H}$ known as the state space of S. The state of S is completely described by a unit vector $|\psi\rangle \in \mathcal{H}$ with $\|\psi\| = 1$, which is called the state vector of S. Two state vectors $|\psi\rangle$ and $|\psi'\rangle$ are equivalent ($|\psi\rangle \simeq |\psi'\rangle$) iff $|\psi'\rangle = \mathrm{e}^{\mathrm{i}\varphi}|\psi\rangle$ with real φ.*

How exactly the state space of for a given physical system is constructed, is beyond the scope of this postulate.

Qubits The simplest non-trivial quantum mechanical system is the *quantum bit* or *qubit* with a state space $\mathcal{B} = \mathbf{C}^2$. The state $|\psi\rangle$ of a qubit can be described by a linear combination (also called *superposition*) of just two basis states labeled $|0\rangle$ and $|1\rangle$

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{with} \quad \alpha, \beta \in \mathbf{C} \quad \text{and} \quad |\alpha|^2 + |\beta|^2 = 1 \tag{1.11}$$

1.2.3.2 Evolution

Postulate 2 *The temporal evolution of the state of a closed quantum system is described by the Schrödinger equation*

$$\mathrm{i}\hbar\frac{\partial}{\partial t}|\psi\rangle = H|\psi\rangle \tag{1.12}$$

with the (experimental) Planck constant $\hbar \approx 1.05457 \cdot 10^{-34}\mathrm{Js}$ and a fixed self-adjoint operator H on the state space $\mathcal{H}$ known as the Hamiltonian of the system.

In quantum physics, it is common to use a system of measurement where $\hbar = 1$, so that (1.12) can be written in the dimensionless form $i|\dot{\psi}\rangle = H|\psi\rangle$.

The Hamiltonian H completely describes the dynamics of a closed quantum system. As with the state space, the concrete form of H (or an approximation thereof) must be determined by the physical theory used to describe the system.

Unitary Evolution If we know the system to be in some initial state $|\psi_0\rangle$ at the time $t = 0$, we can define an operator $U(t)$ such that

$$HU(t)\,|\psi_0\rangle = \mathrm{i}\frac{\partial}{\partial t}U(t)\,|\psi_0\rangle \quad \text{and} \quad U(0)|\psi\rangle = |\psi\rangle \tag{1.13}$$

and get the operator equation

$$H\,U(t) = \mathrm{i}\frac{\partial}{\partial t}U(t) \tag{1.14}$$

with the solution

$$U(t) = \mathrm{e}^{-\mathrm{i}Ht} = \sum_{n=0}^{\infty} \frac{1}{n!}\,(-\mathrm{i})^n t^n H^n \tag{1.15}$$

$U(t)$ is the *operator of temporal evolution* and satisfies the criterion

$$U(t)\,|\psi(t_0)\rangle = |\psi(t_0+t)\rangle \tag{1.16}$$

$U(t)$ is unitary because $H = H^\dagger$ and therefore

$$U(t)U^\dagger(t) = \mathrm{e}^{-\mathrm{i}Ht}\mathrm{e}^{+\mathrm{i}Ht} = I, \tag{1.17}$$

In fact, $U(t)$ and the Hamiltonian are equivalent descriptions of a system's dynamics. Since any unitary operator U can be expressed as the exponential of a self-adjoint operator H such that $U = \mathrm{e}^{-\mathrm{i}H}$, we can reformulate the 2$^{\text{nd}}$ postulate in a non-continuous, discrete-time version:

The temporal evolution of a closed quantum system from the state $|\psi\rangle$ at time t_1 to state $|\psi'\rangle$ at time t_2 can be described by a unitary operator $U = U(t_2 - t_1)$ such that $|\psi'\rangle = U|\psi\rangle$.

In either formulation, the postulate only applies to closed systems, so H or $U(t)$ are fixed operators. It is however often possible to interact with a quantum system in such a manner that it can still be treated as isolated, while the effect of the interaction can mathematically be described by a *time-varying Hamiltonian.* Even in that case, the discrete evolution of the system between two points in time t_1 and t_2 can still be described by a unitary operator $U = U(t_1, t_2)$. In this context, we would speak of *applying* the operator U to a quantum state $|\psi\rangle$.

1.2.3.3 Measurements

Postulate 3 *A (projective) measurement[2] is described by a self-adjoint operator M, called observable, with the spectral decomposition $M = \sum_m mP_m$, where P_m is the projector onto the eigenspace of the eigenvalue m.*

The eigenvalues m of M correspond to the possible outcomes of the measurement. Measuring $|\psi\rangle$ will give the result m with probability $p(m) =$

[2] There is also a more general formulation of quantum measurement allowing non-projective measurement operators. See [43] for details.

$\langle\psi|P_m|\psi\rangle$, *thereby reducing* $|\psi\rangle$ *to the post-measurement state*

$$|\psi'\rangle = \frac{1}{\sqrt{p(m)}} P_m|\psi\rangle \tag{1.18}$$

For a qubit state, the self-adjoint operator N

$$N = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = 0 \cdot |0\rangle\langle 0| + 1 \cdot |1\rangle\langle 1| \tag{1.19}$$

is known as the *standard observable*. Generally, for a system with the state space $\mathcal{H} = \mathbf{C}^n$, the standard observable N can be defined as $N = \sum_i i|i\rangle\langle i|$.

Definition 12 *The weighted average* $\langle M\rangle$ *over all possible outcomes of a measurement of* M *is called expectation value and is defined as*

$$\langle M\rangle = \sum_m p(m)m = \sum_m \langle\psi|mP_m|\psi\rangle = \langle\psi|M|\psi\rangle \tag{1.20}$$

Definition 13 *The standard deviation* ΔM *of all possible outcomes of a measurement is defined as*

$$\Delta M = \sqrt{\langle(M - \langle M\rangle)^2\rangle} = \sqrt{\langle M^2\rangle - \langle M\rangle^2} \tag{1.21}$$

The Heisenberg Uncertainty Principle The destructive nature of measurement raises the question whether 2 observables A and B can be measured simultaneously. This can only be the case if the post-measurement state $|\psi'\rangle$ is an eigenvector of A and B

$$A|\psi'\rangle = a|\psi'\rangle \quad \text{and} \quad B|\psi'\rangle = b|\psi'\rangle \tag{1.22}$$

This is equivalent to the condition $[A, B] = 0$. If A and B do not commute, then the *uncertainty product* $(\Delta A)(\Delta B) > 0$.

To find a lower limit for $(\Delta A)(\Delta B)$ we introduce the operators $\delta A = A - \langle A\rangle$ and $\delta B = B - \langle B\rangle$ and can express the squared uncertainty product as

$$(\Delta A)^2(\Delta B)^2 = \langle(\delta A)^2\rangle\langle(\delta B)^2\rangle = \langle\psi|(\delta A)(\delta A)|\psi\rangle\langle\psi|(\delta B)(\delta B)|\psi\rangle \tag{1.23}$$

Since δA and δB are self adjoint, we can express the above as

$$(\Delta A)^2(\Delta B)^2 = \|\delta A|\psi\rangle\|^2\|\delta B|\psi\rangle\|^2. \tag{1.24}$$

Using (1.1) and $[A, B] = [\delta A, \delta B]$ we get

$$(\Delta A)(\Delta B) \geq \frac{1}{2}\Big|\langle[A, B]\rangle\Big| \tag{1.25}$$

1.2.3.4 Composite Systems

Postulate 4 *The state space $\mathcal{H}$ of a composite physical system is the tensor product of the state spaces $\mathcal{H}_i$ of its components. Moreover, if the subsystems are in the states $|\psi_i\rangle \in \mathcal{H}_i$, then the joint state $|\Psi\rangle \in \mathcal{H}$ of the composite system is $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$.*

Let S be a composite system of S_1 and S_2 with the state space $\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2$. A measurement of the observable $M : \mathcal{H}_1 \to \mathcal{H}_1$ in S_1, is equivalent to measuring the observable $M^{(1)} = M \otimes I$ in S with I being the identity operator on $\mathcal{H}_2$. Equivalently, a unitary transformation $U : \mathcal{H}_1 \to \mathcal{H}_1$ of S_1 is described by the padded operator $U^{(1)} = U \otimes I$ on $\mathcal{H}$.

A joint state of the form $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$ is called *product state*, which can be expanded to

$$|\Psi\rangle = \sum_i \sum_j a_i b_j |i, j\rangle \quad \text{with} \quad \sum_i |a_i|^2 = 1 \quad \text{and} \quad \sum_j |b_j|^2 = 1 \tag{1.26}$$

In product states, unitary transformations or measurements performed on one system, do not affect the state of the other system.

Entanglement Not any joint state is a product state. A state

$$|\Psi\rangle = \sum_i \sum_j c_{ij} |i, j\rangle \quad \text{with} \quad \sum_{i,j} |c_{ij}|^2 = 1 \tag{1.27}$$

where the coefficients c_{ij} cannot be written as $c_{ij} = a_i b_j$ is called *entangled*.

Consider the following joint states of two qubits

$$|\Psi_A\rangle = \frac{1}{2}|0,0\rangle + \frac{1}{2}|1,0\rangle + \frac{1}{2}|0,1\rangle + \frac{1}{2}|1,1\rangle \quad \text{and} \tag{1.28}$$

$$|\Psi_B\rangle = \frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle \tag{1.29}$$

A single measurement of either qubit (using the standard observable as defined in (1.19)) will give 0 or 1 with equal probability $p = 1/2$. Assuming that a measurement of the first qubit gave the result m, the respective post measurement states are

$$|\Psi'_A\rangle = \frac{1}{\sqrt{2}}|m,0\rangle + \frac{1}{\sqrt{2}}|m,1\rangle \quad \text{and} \tag{1.30}$$

$$|\Psi'_B\rangle = |m,m\rangle \tag{1.31}$$

A measurement of the second qubit of $|\Psi'_A\rangle$ will still give a random result, while in the case of $|\Psi'_B\rangle$, the outcome is correlated to the previous measurement and will always produce m. $|\Psi_B\rangle$ is also known as *Bell state*.

1.3 Classical Computing

1.3.1 The Church-Turing Thesis

As already mentioned in 1.1.3, computing science is based on the paradigm of computation being a mechanical, rather than a purely mental process. A method, or procedure $\mathcal{P}$ for achieving some desired result is called *effective* or *mechanical* if [21]

1. $\mathcal{P}$ is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);

2. $\mathcal{P}$ will, if carried out without error, always produce the desired result in a finite number of steps;

3. $\mathcal{P}$ can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;

4. $\mathcal{P}$ demands no insight or ingenuity on the part of the human being carrying it out.

Alan Turing and Alonzo Church both formalized the above definition by introducing the concept of *computability by Turing machine* and the mathematically equivalent concept of *recursive functions* with the following conclusions:

Turing's Thesis LCMs *[logical computing machines i.e. Turing machines]* can do anything that could be described as "rule of thumb" or "purely mechanical". [58]

Church's Thesis A function of positive integers is effectively calculable only if recursive. [50]

As the above statements are equivalent, they are commonly referred to as the *Church-Turing Thesis* which defines the scope of classical computing.

1.3.1.1 Partial Recursive Functions

The class P of *partial recursive functions* mathematically captures the concept of "effective" functions $f : \mathbf{N}^n \to \mathbf{N}^m$. P can be constructed from simpler classes in the following way: [39]

1. A *basic function* $f : \mathbf{N}^n \to \mathbf{N}^m$ is a function $f : x \to y$ where y_i is either a constant $y_i = c_i, c_i \in \mathbf{N}$ or an element of the input vector $y_i = x_{\sigma(i)}$. The class BF of basic function is closed under the *basic operators* BO $= \{\circ, \times\}$, where "$\circ$" denotes function composition and "$\times$" the usual cross-product.

2. The class PR of *primitive recursive functions* is generated from BF$\cup\{S\}$ by closure under BO $\cup$ {Pr} where

   (i) $S : \mathbf{N} \to \mathbf{N}$ is the *successor function* $S(n) = n + 1$ and

   (ii) Pr denotes the *primitive recursion* $h = \mathrm{Pr}[f, g]$

$$h(x, 0) = f(x),\ h(x, n + 1) = g(x, n, h(x, n)) \tag{1.32}$$

3. P is generated from PR by closure under BO $\cup \{\mu_0\}$. The operator μ_0 is called μ_0-recursion (minimization) and defined as

$$\mu_0[f] : x \to \min_{k\in\mathbf{N}} [f(x, k) = 0] \quad \text{with} \quad f \in \mathrm{PR} \tag{1.33}$$

As $\mu_0[f](x)$ is only defined if $\exists k \in \mathbf{N}, f(x, k) = 0$, P is a class of partial functions. The class R $\subset$ P of total functions in P is called *recursive functions*.

1.3.2 Machines

1.3.2.1 General Machines

Definition 14 *A machine $\mathcal{M}$ is a 5-tuple* $(\mathbf{S}, O, T, \delta, \beta)$ *where* [39]

(i) $\mathbf{S}$ *is a set of of computational states*

(ii) $O = \{f_i : \mathbf{S} \to \mathbf{S}\}$ *is an enumerable set of operations on* $\mathbf{S}$ *(memory commands)*

(iii) $T = \{t_i : \mathbf{S} \to \mathbf{B}\}$ *is an enumerable set of predicates on* $\mathbf{S}$ *(test commands)*

(iv) $\delta : \mathcal{I} \to \mathbf{S}$ *is an input function for the enumerable input set* $\mathcal{I}$

(v) $\beta : \mathbf{S} \to \mathcal{O}$ *is an output function for the enumerable output set* $\mathcal{O}$

By providing a set of (simple) elementary operations and predicates, a machine defines a framework for the description of effective procedures. The enumerability of O and T guarantees that such a description, called *program*, can be finite and represented as a string over a finite set of symbols.

1.3.2.2 Discrete Machines

A more rigid interpretation of effectivity also requires $\mathbf{S}$ to be enumerable, so that not only the program but also the computational state can be expressed "by finite means" and the whole computation can in fact be carried out by manipulating symbols "on paper". Such machines are known as *discrete.* For any machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$ with $O = \{f_1, f_2, \ldots\}$ and the input set $\mathcal{I} = \{x_1, x_2, \ldots\}$ an equivalent discrete machine $\mathcal{M}' = (\mathbf{S}', O, T, \delta, \beta)$ can be constructed by the diagonalization

$$\mathbf{S}' = \bigcup_{n=1}^{\infty} \{(g_1 \circ g_2 \circ \ldots \circ g_n)(\delta(x)) \,|\, g \in O_n^n, x \in \mathcal{I}_n\} \tag{1.34}$$

where $O_0 = \{I\}$, $O_{n+1} = O_n \cup \{f_n\}$ and $\mathcal{I}_n = \{x_0, \ldots x_n\}$.

Turing Machines A Turing Machine (TM) consists of a head operating on an infinite tape of memory cells. In the simplest case, each cell can only adopt one of two possible states labeled `0` (also called *blank*) and `1`.

If we index the cells by their relative position to the head, we can describe the content of the tape content as a function $\mathbf{s} : \mathbf{Z} \to \mathbf{B}$ and write

$$\mathbf{s} = (\ldots s_{-2} s_{-1} | s_0 s_1 s_2 \ldots). \tag{1.35}$$

The state space $\mathbf{T}$ is the set of all tapes containing only a finite number of `1`s.[3]

$$\mathbf{T} = \Big\{\mathbf{s} : \mathbf{Z} \to \mathbf{B} \;\Big|\; \sum_{i=-\infty}^{\infty} s_i < \infty\Big\} \tag{1.36}$$

We can now define a TM as a machine $\mathcal{M} = (\mathbf{T}, \{\mathrm{S}, \mathrm{E}, \mathrm{L}, \mathrm{R}\}, \{\mathrm{T}\}, \delta, \beta)$ with the commands

(i) $\mathrm{S}(\ldots s_{-1} | s_0 s_1 s_2 \ldots) = (\ldots s_{-1} | \mathtt{1} s_1 s_2 \ldots)$ *(set)*

(ii) $\mathrm{E}(\ldots s_{-1} | s_0 s_1 s_2 \ldots) = (\ldots s_{-1} | \mathtt{0} s_1 s_2 \ldots)$ *(erase)*

(iii) $\mathrm{L}(\mathbf{s}) = \mathbf{s}'$ where $s'_i = s_{i+1}$ *(move left)*

(iv) $\mathrm{R}(\mathbf{s}) = \mathbf{s}'$ where $s'_i = s_{i-1}$ *(move right)*

(v) $\mathrm{T}(\mathbf{s}) = s_0$ *(test)*

For $\mathcal{I} = \mathbf{N}^m$ and $\mathcal{O} = \mathbf{N}^n$ we can define a Turing machine $\mathrm{TM}_n^m = (\mathbf{T}, \{\mathrm{S}, \mathrm{E}, \mathrm{L}, \mathrm{R}\}, \{\mathrm{T}\}, \delta_m, \beta_n)$ with the *unary encoding*

$$\delta_m(x_1, x_2, \ldots x_n) = (\mathtt{0}^\omega | \mathtt{1}^{x_1} \mathtt{0} \mathtt{1}^{x_2} \mathtt{0} \ldots \mathtt{0} \mathtt{1}^{x_m} \mathtt{0}^\omega) \quad \text{and} \tag{1.37}$$

$$\beta_n(\ldots | \mathtt{1}^{y_1} \mathtt{0} \mathtt{1}^{y_2} \mathtt{0} \ldots \mathtt{0} \mathtt{1}^{y_n} \mathtt{0} \ldots) = (y_1, y_2, \ldots y_n) \tag{1.38}$$

[3] This *zero tails state* condition is necessary as the set $\mathbf{T}' = \{\mathbf{s} : \mathbf{Z} \to \mathbf{B}\}$ would not be enumerable.

1.3.2.3 Finite Machines

For a discrete machine $\mathcal{M}$ with an infinite $\mathbf{S}$, the number of symbols to represent a computational state $s \in \mathbf{S}$ can get arbitrarily large so any realization of $\mathcal{M}$ would require unlimited memory. If the amount of memory is limited, so is the number of computational states. A discrete machine $\mathcal{M}$ with limited memory is called *finite*.

The *memory capacity* of a finite machine $\mathcal{M}$ with the finite state space $\mathbf{S}$ is $S = \log_2 |\mathbf{S}|$ bit.

1.3.2.4 Oracles

If a machine $\mathcal{M}_1 = (\mathbf{S}_1, O_1, T_1, \delta_1, \beta_2)$ is extended to allow computations on another machine $\mathcal{M}_2 = (\mathbf{S}_2, O_2, T_2, \delta_2, \beta_2)$, then the resulting machine $\mathcal{M} = \mathcal{M}_1 \bowtie \mathcal{M}_2$ is referred to as an $\mathcal{M}_1$-machine with an $\mathcal{M}_2$-*oracle*.

The interaction between $\mathcal{M}_1$ and $\mathcal{M}_2$ is described by *oracle commands* of the form

$$f_O : \mathbf{S}_1 \times \mathbf{S}_2 \to \mathbf{S}_1 \times \mathbf{S}_2 \quad \text{or} \quad t_O : \mathbf{S}_1 \times \mathbf{S}_2 \to \mathbf{B} \tag{1.39}$$

and $\mathcal{M}$ can be written as

$$\mathcal{M} = (\mathbf{S}_1 \times \mathbf{S}_2, O_1 \times \{I_1\} \cup \{f_O\}, T_1 \cup \{t_O\}, \delta, \beta) \tag{1.40}$$

with I_2 being the identity on $\mathbf{S}_2$.

Depending on the definition of f_O and t_O, oracle calls can correspond to single $\mathcal{M}_2$-commands up to the execution of complete finite programs (see 1.3.3.2) on $\mathcal{M}_2$. Still, with regard to time complexity, an oracle-call counts as a single computational step.

1.3.2.5 Probabilistic Machines

A machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$ is *probabilistic* if it provides at least one random test command $c \in T$. A random predicate $c : \mathbf{S} \mapsto \mathbf{B}$ can be mathematically described by the probability distribution $p(c|s)$ where $s \in \mathbf{S}$, so

$$c : s \mapsto \begin{cases} \texttt{true} & \text{with} \quad p = p(c|s) \\ \texttt{false} & \text{with} \quad p = 1 - p(c|s) \end{cases} \tag{1.41}$$

We can generalize this definition by also allowing for random memory commands $f : \mathbf{S} \mapsto \mathbf{S}$

$$f : s \mapsto s' \quad \text{with} \quad p = p_f(s'|s) \quad \text{where} \quad \forall s \in \mathbf{S}, \sum_{s' \in \mathbf{S}} p_f(s'|s) = 1 \tag{1.42}$$

and output functions $\beta : \mathbf{S} \mapsto \mathcal{O}$. In the latter case, $p_\beta(y|s)$ with $y \in \mathcal{O}$ is also called *(probability) spectrum* of s.

For any probabilistic machine $\mathcal{M}$ it is possible to formally construct a corresponding deterministic machine $\hat{\mathcal{M}}$ operating on the distribution space

$$\hat{\mathbf{S}} = \Big\{ p_{\hat{s}} : \mathbf{S} \to [0,1] \;\Big|\; \sum_{s \in \mathbf{S}} p_{\hat{s}}(s) = 1 \Big\} \tag{1.43}$$

A state $\hat{s} \in \hat{\mathbf{S}}$ has the *(Shannon) entropy*

$$H(\hat{s}) = -\sum_{s \in \mathbf{S}} p(s|\hat{s}) \log_2 p(s|\hat{s}), \quad p(s|\hat{s}) = p_{\hat{s}}(s). \tag{1.44}$$

Probabilistic Turing Machine A probabilistic Turing Machine (PTM) can be constructed from a deterministic TM by adding a stateless *random oracle* which provides a “fair coin toss” test command C with $p(\mathrm{C}) = p(\neg\mathrm{C}) = 1/2$.

1.3.3 Programs

A program π for some machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$ is a finite set of instructions (also called statements) that determines how to iteratively transform an input state $s_0 = \delta(x)$ using the machine's memory and test commands until some halting condition is met. If the program halts, the resulting state s_h is called *output state*.

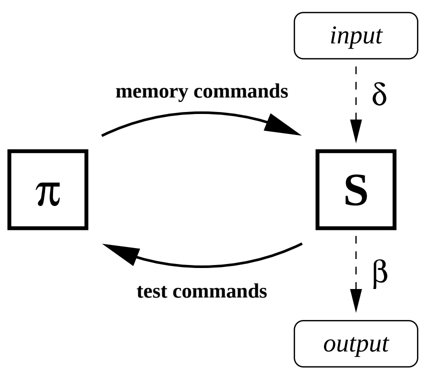


Figure 1.1: *A program π controlling a machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$*

The *transfer function* $\tau_\pi : s_0 \xrightarrow{\pi} s_h$ is a partial function on $\mathbf{S}$ defined for all input states $s_0 \in \mathbf{S}$ for which π halts. $F(\pi, \mathcal{M})$ denotes the partial function $x \to \beta(\tau_\pi(\delta(x)))$ *implemented* by π on $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$.

The interpretation of a program π of a program class Π is specified by a *step function* $\rho : \Pi \times \mathbf{P} \times \mathbf{S} \to \mathbf{P} \times \mathbf{S}$. $\mathbf{P} = \mathbf{P}(\pi)$ is the set of possible *control-states* with a unique $p_0 \in \mathbf{P}$ called *initial state* and a subset $\mathbf{P}_h \subseteq \mathbf{P}$ called *halting states*.

A pair $(p, s) \in \mathbf{P} \times \mathbf{S}$ is called a *configuration*. ρ has the general form

$$\rho_\pi(p, s) = \begin{cases} \big(\nu(\pi, p),\ f_{\mu(\pi,p)}(s)\big) & \text{if } p \in \mathbf{P}_f \\ \big(\nu(\pi, p, t_{\mu(\pi,p)}(s)),\ s\big) & \text{if } p \in \mathbf{P}_t \\ (p,\ s) & \text{if } p \in \mathbf{P}_h \end{cases} \tag{1.45}$$

where $\mathbf{P} = \mathbf{P}_f \cup \mathbf{P}_t \cup \mathbf{P}_h$, $f_{\mu(\pi,p)} \in O$ and $t_{\mu(\pi,p)} \in T$.

For an input state $s_0 = \delta(x)$, $(p_0, s_0) = (p_0, \delta(x))$ is called *initial configuration*. The transfer function τ is defined as

$$(p_h, \tau_\pi(s_0)) = \rho^n(p_0, s_0) \quad \text{with} \quad n = \min_k \rho^k(p_0, s_0) \in \mathbf{P}_h \times \mathbf{S} \tag{1.46}$$

If π halts for a given input $x \in \mathcal{I}$, n is called the *(time) complexity* of the computation.

1.3.3.1 Sequences

Definition 15 *A program $\sigma \in O^n$ for a machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$ consisting of a static list of n memory commands is called a sequence. The transfer function $\tau_\sigma : s \to (\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_n)(s)$ is a total function.*

Definition 16 *Let $\mathbf{S}$ be composed of identical indexed memory cells M_i with the state space $\mathcal{S}$, such that $\mathbf{S} = \mathcal{S}_1 \oplus \mathcal{S}_2 \oplus \ldots$ for some suitable composition $\oplus$ and g a function $g : \mathcal{S}^n \to \mathcal{S}^n$. The class of functions $\Gamma(g) = \{g_{i_1 i_2 \ldots i_n} : \mathbf{S} \to \mathbf{S}\}$ where $g_{i_1 i_2 \ldots i_n}$ denotes the application of g to a permutation of n mutually different cells $M_{i_1}, M_{i_2} \ldots M_{i_n}$ is called an n-ary gate.*

If O is a union of gates, then a sequence $\sigma \in O^\star$ can be interpreted as a feed forward network and is also called a *circuit*.

A set of operations O is *universal* if for any function $f : \mathcal{I}' \to \mathcal{O}$ defined for a finite subset $\mathcal{I}' \subseteq \mathcal{I}$ there exists a sequence $\sigma \in O^\star$ such that $f(x) = \beta(\tau_\sigma(\delta(x)))$ for all $x \in \mathcal{I}'$.

While generally, more powerful programming concepts are required to fully exploit the computational potential of a machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$, sequences are sufficient to implement any function that can possibly be implemented if either

(i) O is universal and (a) $\mathcal{M}$ is finite or (b) $\mathcal{I}$ is a finite set

(ii) $\mathcal{M}$ provides no test-commands, i.e. $T = \emptyset$

1.3.3.2 Finite Programs

Definition 17 *Let $\mathcal{M}$ be a machine $(\mathbf{S}, O, T, \delta, \beta)$ and $\mathbf{L}$ an enumerable set of labels with an element $l_0 \in \mathbf{L}$ called start-label.*

(i) *A triple $(l, f, p) \in \mathbf{L} \times O \times \mathbf{L}$ is called function-statement*

(ii) *A 4-tuple $(l, t, p, q) \in \mathbf{L} \times T \times \mathbf{L} \times \mathbf{L}$ is called test-statement*

The labels l are called statement-, p and q are called jump-labels. A finite program $\pi \in \Pi$ for $\mathcal{M}$ is a finite set of statements with unique labels l.

Function- and test-statements are also written as

```
l:  f then p
l:  if t then p else q
```

A program $\pi \in \Pi$ describes a finite automaton operating on $\mathcal{M}$ with the step function

$$\rho_\pi(l, s) = \begin{cases} (p, f(s)) & \text{if } \pi_l = (l, f, p) \\ (p, s) & \text{if } \pi_l = (l, t, p, q) \wedge t(s) \\ (q, s) & \text{if } \pi_l = (l, t, p, q) \wedge \neg t(s) \\ (l, s) & \text{if } l \notin \mathbf{L}_\pi \end{cases} \tag{1.47}$$

where $\mathbf{L}_\pi$ denotes the set of statement-labels in π and $\pi_l : \mathbf{L}_\pi \to \pi$ the statement with the label l.

According to the Church-Turing thesis, any effective function $f : \mathbf{N}^n \to \mathbf{N}^m$ can be implemented as a finite program π for TM_n^m and the set of *Turing-computable* functions

$$F(\mathrm{TM}) = \{F(\pi, TM_n^m) \,|\, \pi \in \Pi, n, m \in \mathbf{N}\} \tag{1.48}$$

is identical to P.

Definition 18 (Universal Computer) *A machine $\mathcal{M} = (\mathbf{S}, O, T, \delta, \beta)$ with the encodings $\delta : \mathbf{N}^\star \to \mathbf{S}$ and $\beta : \mathbf{S} \to \mathbf{N}^\star$ is a universal computer iff for any $f \in$ P, there exists a finite program π for $\mathcal{M}$ such that $f = F(\pi, \mathcal{M})$.*

1.3.3.3 Programming Languages

Definition 19 *A programming language $L \subseteq \Sigma^\star$ for a machine $\mathcal{M}$ is a class of algorithm descriptions $p \in \Sigma^\star$ over some alphabet Σ, which can be efficiently translated into a finite program $\pi \in \Pi$ for $\mathcal{M}$ by another program π_L for a machine*

$$\mathcal{M}_L = (\mathbf{S}, O, T, \delta : \Sigma^\star \to \mathbf{S}, \beta : \mathbf{S} \to \Pi) \tag{1.49}$$

The pair $(\pi_L, \mathcal{M}_L)$ is called a *compiler* for L. It is usually required that the compilation is efficient, i.e. has polynomial time and space complexity.

A programming language L is universal, if $\mathcal{M}$ is a universal computer and for any $f \in \mathrm{P}$ there exists a program $p \in L$ such that $f = F(p, \mathcal{M}) = F(F(\pi_L, \mathcal{M}_L), \mathcal{M})$.

1.4 Elements of Quantum Computing

Just like a classical machine, a quantum computer, essentially consists of three parts: a memory, which holds the current machine state, a processor, which performs elementary operations on the machine state, and some sort of input/output which allows to set the initial state and extract the final state of the computation.

Formally, we can describe a quantum computer as a probabilistic machine $\mathcal{M} = (\mathcal{H}, O, T, \delta, \beta)$ where

- $\mathcal{H}$ is the state space of the quantum system operated on,
- O a set of (deterministic) unitary transformations,
- T a set of (probabilistic) measurement commands,
- δ is the initialization operator and
- β describes the final measurement.

1.4.1 Quantum Memory

1.4.1.1 Qubits

The quantum analogue to the classical bit is the quantum bit or *qubit* (see 1.2.3.1).

Definition 20 *A qubit or quantum bit is a quantum system whose state can be fully described by a superposition of two orthonormal basis states labeled* $|0\rangle$ *and* $|1\rangle$.

The state space of a qubit is the Hilbert space $\mathcal{B} = \mathbf{C}^2$. The orthonormal system $\{|0\rangle, |1\rangle\}$ is called *computational basis*.

The classical value of a qubit is described by the standard observable $N = |1\rangle\langle 1|$ (1.19). $\langle N\rangle$ gives the probability to find the system in state $|1\rangle$ if a measurement is performed on the qubit.

The Bloch Sphere Ignoring an irrelevant overall phase factor, the general state of a qubit can be written as

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + \mathrm{e}^{\mathrm{i}\varphi}\sin\frac{\theta}{2}|1\rangle \tag{1.50}$$

By interpreting θ and φ as polar coordinates

$$\hat{r} = (\cos\varphi\sin\theta, \sin\varphi\sin\theta, \cos\theta), \tag{1.51}$$

every qubit state has a unique representation as a point on the three-dimensional unit sphere, also known as *Bloch sphere*.

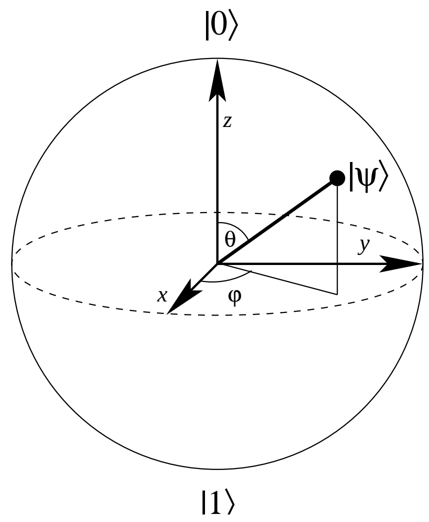


Figure 1.2: *Bloch sphere representation of the qubit state* $|\psi\rangle$

The unit-vector $\hat{r} = \hat{r}_\psi$ is called *Bloch vector* of $|\psi\rangle$. Bloch vectors have the property that

$$\hat{r}_\phi = -\hat{r}_\chi \iff \langle\phi|\chi\rangle = 0. \tag{1.52}$$

1.4.1.2 Machine State

Definition 21 *The state space of a quantum computer is a separable complex Hilbert space* $\mathcal{H}$ *with a designated enumerable orthonormal system* $B = \{|i\rangle\}$ *called computational basis. The state of a quantum computer is a unit-vector* $|\Psi\rangle \in \mathcal{H}$ *known as machine state.*

Classically, the common state space $\mathbf{S}$ of a composite system consisting of n memory cells with the state spaces $\mathbf{S}_i$ is given by the cross-product $\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n$. In quantum mechanics, this is only true for product states (see 1.2.3.4).

The state space $\mathcal{H}$ of a quantum computer composed of n identical subsystems with the state space $\mathcal{S}$ is given as the tensor product

$$\mathcal{H} = S^{\otimes n} = \overbrace{S \otimes \ldots \otimes S}^{n\ times} \tag{1.53}$$

The *machine state* $|\Psi\rangle$ of an n-qubit quantum computer is therefore a unit vector in $\mathcal{H} = \mathcal{B}^{\otimes n} = \mathbf{C}^{2^n}$

$$|\Psi\rangle = \sum_{(d_0 \ldots d_{n-1}) \in \mathbf{B}^n} c_{d_0 \ldots d_{n-1}} |d_0 \ldots d_{n-1}\rangle \quad \text{with} \quad \sum |c_{d_0 \ldots d_{n-1}}|^2 = 1 \tag{1.54}$$

The basis vectors $|d_0 \ldots d_{n-1}\rangle$ can be interpreted as binary numbers and relabeled as $|k\rangle$ with $k = \sum_{i=0}^{n-1} 2^i d_i$ so B can be written as $B = \{|k\rangle | k \in \mathbf{Z}_{2^n}\}$.

A quantum computer $\mathcal{M} = (\mathcal{H}, O, T, \delta, \beta)$ is finite if $\dim \mathcal{H} < \infty$. The memory capacity of a finite quantum computer is $\log_2 \dim \mathcal{H}$ qubit.

Unlimited Memory As the state space of a quantum computer is a separable Hilbert space, it is isomorphic to either $\mathbf{C}^n$ or $\mathbf{l}_2$ (see 1.2.2.2).

The Hilbert space $\mathcal{B}^{\otimes\omega}$ resulting from a composition of an infinite number of qubits is non-separable. We can however construct a separable subspace $\mathcal{B}^\star \subset \mathcal{B}^{\otimes\omega}$ which is isomorphic to $\mathbf{l}_2$ by introducing a *zero tail state* condition.

$$\mathcal{B}^\star = \left\{ |\psi\rangle \otimes |0\rangle^{\otimes\omega} \;\middle|\; |\psi\rangle \in \mathcal{B}^{\otimes n}, n \in \mathbf{N} \right\} \tag{1.55}$$

1.4.1.3 Quantum Registers

Definition 22 *Let $\mathcal{H}$ be the state space of a quantum computer $\mathcal{M}$ with the computational basis $B = \{|i\rangle\}$. A quantum register* $\mathbf{s}$ *is a sub-system of $\mathcal{M}$ with a finite dimensional state space $\mathcal{H}_\mathbf{s}$ and a basis $B_\mathbf{s} = \{|i\rangle_\mathbf{s}\}$ such that $\mathcal{H} = \mathcal{H}_\mathbf{s} \otimes \mathcal{H}_{\bar{\mathbf{s}}}$ and $B = B_\mathbf{s} \times B_{\bar{\mathbf{s}}}$. If $\mathcal{M}$ is finite, then $\bar{\mathbf{s}}$ is known as the complimentary register to* $\mathbf{s}$.

A register $\mathbf{s}$ defines a decomposition for the computational basis B, so any basis vector $|k\rangle \in B$ can be written as the product state $|k\rangle = |i_k\rangle_\mathbf{s} |j_k\rangle_{\bar{\mathbf{s}}}$ with $|i_k\rangle_\mathbf{s} \in B_\mathbf{s}$ and $|j_k\rangle_{\bar{\mathbf{s}}} \in B_{\bar{\mathbf{s}}}$.

In $\mathcal{H}$ the classical value of a register $\mathbf{s}$ is described by the *register observable* $N(\mathbf{s})$

$$N(\mathbf{s}) \equiv N_\mathbf{s} \otimes I_{\bar{\mathbf{s}}} = \sum_{i,j} i\, |i\rangle_\mathbf{s} |j\rangle_{\bar{\mathbf{s}}} \langle i|_\mathbf{s} \langle j|_{\bar{\mathbf{s}}} \tag{1.56}$$

where $N_{\mathbf{s}}$ is the standard observable on $\mathcal{H}_{\mathbf{s}}$ and $I_{\bar{\mathbf{s}}}$ the identity operator on $\mathcal{H}_{\bar{\mathbf{s}}}$. Likewise, a unitary transformation U on $\mathcal{H}_{\mathbf{s}}$ can be expressed as a *register operator* $U(\mathbf{s})$ on $\mathcal{H}$

$$U(\mathbf{s}) \equiv U \otimes I_{\bar{\mathbf{s}}} = \sum_{i,i',j} u_{i,i'} \, |i\rangle_{\mathbf{s}} |j\rangle_{\bar{\mathbf{s}}} \langle i'|_{\mathbf{s}} \langle j|_{\bar{\mathbf{s}}} \tag{1.57}$$

where $u_{i,i'}$ is the matrix element $u_{i,i'} = \langle i|U|i'\rangle$.

Qubit Registers When $\mathcal{H}$ is the state space of a composition of the qubits q_i, i.e. $\mathcal{H} = \mathcal{B}^{\otimes m}$ or $\mathcal{H} = \mathcal{B}^{\star}$, then any q_i defines a register $\mathbf{q}_i$ with the decomposition

$$|d_0 \ldots d_{i-1} d_i d_{i+1} \ldots\rangle = |d_i\rangle_{\mathbf{q}_i} |d_0 \ldots d_{i-1} d_{i+1} \ldots\rangle_{\bar{\mathbf{q}}_i} \tag{1.58}$$

Likewise, any permutation $(s_0 s_1 \ldots s_{n-1}) = (q_{k_0} q_{k_1} \ldots q_{k_{n-1}})$ of mutually different qubits $s_j \in \{q_i\}$ defines an n-qubit register $\mathbf{s} = \mathbf{q}_{k_0} \circ \mathbf{q}_{k_1} \circ \ldots \circ \mathbf{q}_{k_{n-1}}$ with the decomposition

$$|d_0 d_1 \ldots\rangle = \left(\bigotimes_{i=0}^{n-1} |d_{k_i}\rangle_{\mathbf{q}_{k_i}} \right) \otimes |\ldots\rangle_{\bar{\mathbf{s}}} = |d_{k_0} d_{k_1} \ldots d_{k_{n-1}}\rangle_{\mathbf{s}} |\ldots\rangle_{\bar{\mathbf{s}}} \tag{1.59}$$

The order of qubits is important, so while $\mathbf{a} \circ \mathbf{b}$ and $\mathbf{b} \circ \mathbf{a}$ refer to the same 2-qubit subsystem, they are two different registers.

If $\mathbf{s}$ is a n-qubit register and U an operator on $\mathcal{B}^{\otimes n}$, then the register operator $U(\mathbf{s})$ is also referred to as a *quantum gate*.

Register States Strictly speaking, the state of a register $\mathbf{s}$ is only defined if the machine state $|\Psi\rangle$ is of the form $|\Psi\rangle = |\psi\rangle_{\mathbf{s}} |\chi\rangle_{\bar{\mathbf{s}}}$. In that case, we say that $\mathbf{s}$ is in the *pure state* $|\psi\rangle \in \mathcal{H}_{\mathbf{s}}$. Alternatively, the state of $\mathbf{s}$ can be described by the *(reduced) density operator* $\rho_{\mathbf{s}}$.

Definition 23 *Let $\mathcal{H} = \mathcal{H}_{\mathbf{a}} \otimes \mathcal{H}_{\mathbf{b}}$ be the state space of a composite system $\mathbf{a} \circ \mathbf{b}$. For a machine state*

$$|\Psi\rangle = \sum_{i,j} c_{ij} |i\rangle_{\mathbf{a}} |j\rangle_{\mathbf{b}} \quad \textit{with} \quad \sum_{i,j} |c_{ij}|^2 = 1 \tag{1.60}$$

the reduced density operator $\rho_{\mathbf{a}} : \mathcal{H}_{\mathbf{a}} \to \mathcal{H}_{\mathbf{a}}$ is defined as

$$\rho_{\mathbf{a}} = \mathrm{tr}_{\mathbf{b}} \left(|\Psi\rangle\langle\Psi| \right) = \sum_{i,i'} |i\rangle_{\mathbf{a}} \langle i'|_{\mathbf{a}} \sum_j c_{ij} c^*_{i'j} \tag{1.61}$$

If a register $\mathbf{s}$ is in a pure state then $\rho_{\mathbf{s}} = |\psi\rangle\langle\psi|$ and $\mathrm{tr}(\rho_{\mathbf{s}}^2) = \mathrm{tr}\rho_{\mathbf{s}} = 1$. If $\mathbf{s}$ is entangled then $\rho_{\mathbf{s}}$ is a positive operator with the spectral decomposition

$$\rho_{\mathbf{s}} = \sum_k p_k |\psi_k\rangle\langle\psi_k| \quad \text{with} \quad p_k \in [0,1) \quad \text{and} \quad \sum_k p_k = 1 \tag{1.62}$$

and $\mathrm{tr}(\rho_{\mathbf{s}}^2) < 1$. In that case, $\mathbf{s}$ is also said to be in the *mixed state* $\rho_{\mathbf{s}}$.[4]

The density operator allows to treat $\mathbf{s}$ as an isolated system with regard to unitary evolution and measurements as long as no operation on $\bar{\mathbf{s}}$ is performed:

(i) Let U be a unitary operator on $\mathcal{H}_{\mathbf{s}}$, then

$$|\Psi'\rangle = U(\mathbf{s})|\Psi\rangle \implies \rho'_{\mathbf{s}} = U\rho_{\mathbf{s}}U^\dagger. \tag{1.63}$$

(ii) Let M be a Hermitian operator on $\mathcal{H}_{\mathbf{s}}$ with the spectral decomposition $M = \sum_m m P_m$, then $p(m) = \langle\Psi|P_m(\mathbf{s})|\Psi\rangle = \mathrm{tr}(P_m\rho_{\mathbf{s}})$ and

$$|\Psi'\rangle = \frac{P_m|\Psi\rangle}{\sqrt{p(m)}} \implies \rho'_{\mathbf{s}} = \frac{P_m\rho_{\mathbf{s}}P_m^\dagger}{p(m)}. \tag{1.64}$$

Schmidt Decomposition If $\mathbf{s}$ and $\bar{\mathbf{s}}$ are entangled, the machine state can always be written as

$$|\Psi\rangle = \sum_i \lambda_i |\psi_i\rangle_{\mathbf{s}} |\chi_i\rangle_{\bar{\mathbf{s}}} \quad \text{with} \quad \lambda_i \in \mathbf{R}^+ \quad \text{and} \quad \sum_i \lambda^2 = 1 \tag{1.65}$$

such that $|\psi_i\rangle \in \mathcal{H}_{\mathbf{s}}$ and $|\chi_i\rangle \in \mathcal{H}_{\bar{\mathbf{s}}}$ are orthonormal states i.e. $\langle\psi_i|\psi_j\rangle = \delta_{ij}$ and $\langle\chi_i|\chi_j\rangle = \delta_{ij}$. This representation is known as *Schmidt decomposition.*

1.4.2 Quantum Operations

1.4.2.1 Unitary Operators

According to the 2$^{\text{nd}}$ postulate of quantum mechanics (see 1.2.3.2), the evolution of a closed quantum system is unitary and can be described by the operator $U(t) = \mathrm{e}^{-\mathrm{i}Ht}$. Therefore, the memory commands O of a quantum computer $\mathcal{M} = (\mathcal{H}, O, T, \delta, \beta)$ are unitary transformations on the state space $\mathcal{H}$.

A unitary transformations U is a linear operator of the form $UU^\dagger = I$ and describes a basis transformation $B \xrightarrow{U} \tilde{B}$. From a computational point of view, these mathematical properties account for three fundamental differences between classical and quantum computing:

[4]The term “mixed state” refers to the fact that ρ exhibits the same measurement statistics as a system which is known to be in the state $|\psi_k\rangle$ with probability p_k.

- **Reversibility:** Since unitary operators, by definition, match the condition $UU^\dagger = I$, for every transformation U there exists the inverse transformation $U^{(-1)} = U^\dagger$. As a consequence, quantum computation is restricted to reversible functions.[5]

- **Superposition:** A "classical" state $|\Psi\rangle = |k\rangle \in B$ can be transformed into a superposition of several basis vectors

$$|\Psi'\rangle = U\,|k\rangle = |\tilde{k}\rangle = \sum_{k'} u_{k'k}|k'\rangle \tag{1.66}$$

and vice versa.

- **Parallelism:** If the machine state $|\Psi\rangle$ already is a superposition of several basis vectors, then a transformation U is applied to all basis states simultaneously.

$$U \sum_k c_k|k\rangle = \sum_k c_k U|k\rangle \tag{1.67}$$

This feature of quantum computing is called *quantum parallelism* and is a consequence of the linearity of unitary transformations.

1.4.2.2 Qubit Operators

The simplest case of unitary transformations are operators which work on a single qubit. A general 2-dimensional complex unitary matrix $U \in SU(2)$ can be written as

$$U = \mathrm{e}^{\mathrm{i}\varphi} \begin{pmatrix} \mathrm{e}^{\frac{\mathrm{i}}{2}(-\alpha-\beta)} \cos\frac{\theta}{2} & -\mathrm{e}^{\frac{\mathrm{i}}{2}(-\alpha+\beta)} \sin\frac{\theta}{2} \\ \mathrm{e}^{\frac{\mathrm{i}}{2}(\alpha-\beta)} \sin\frac{\theta}{2} & \mathrm{e}^{\frac{\mathrm{i}}{2}(\alpha+\beta)} \cos\frac{\theta}{2} \end{pmatrix} \tag{1.68}$$

As we have shown in 1.4.1.1, every qubit state $|\psi\rangle \in \mathcal{B}$ can be represented by a Bloch vector $\hat{r}_\psi$. Rotations about the $\hat{x}$, $\hat{y}$ and $\hat{z}$-axes in the Bloch sphere correspond to the operators

$$R_x(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\mathrm{i}\sin\frac{\theta}{2} \\ -\mathrm{i}\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \tag{1.69}$$

$$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \tag{1.70}$$

$$R_z(\theta) = \begin{pmatrix} \mathrm{e}^{-\mathrm{i}\theta/2} & 0 \\ 0 & \mathrm{e}^{\mathrm{i}\theta/2} \end{pmatrix} \tag{1.71}$$

[5] A classical analogue would be the class of bijective functions on $\mathbf{B}^n$.

on $\mathcal{B}$. It is easy to verify that $R_i^\dagger(\theta) = R_i(-\theta)$, so all R_i are unitary operators.

Using the self-inverse *Pauli matrices*

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.72}$$

a general rotation[6] about a unit vector $\hat{n}$ can be written as

$$R_{\hat{n}}(\theta) = \mathrm{e}^{-\frac{\mathrm{i}}{2}\theta\hat{n}\cdot\vec{\sigma}} = \cos\frac{\theta}{2}I - \mathrm{i}\sin\frac{\theta}{2}(n_x\sigma_x + n_y\sigma_y + n_z\sigma_z) \tag{1.73}$$

Let $U \in SU(2)$ have the orthonormal eigenvectors $|u\rangle$ and $|v\rangle$ and the eigenvalues u and v, then U can be written as

$$U = u|u\rangle\langle u| + v|v\rangle\langle v| = \mathrm{e}^{\mathrm{i}\varphi}R_{\hat{n}}(\delta) \tag{1.74}$$

where $\hat{n}$ is the Bloch vector of $|u\rangle$ and δ is the phase difference between u and v, i.e. $v = \mathrm{e}^{\mathrm{i}\delta}u$.

1.4.2.3 Universal Qubit Operations

As qubit operators correspond to rotations in the Bloch sphere, any unitary U on $\mathcal{B}$ can implemented as a composition of three rotations about two orthogonal axes and a (physically irrelevant) overall phase factor.[7] e.g.

$$U = \mathrm{e}^{\mathrm{i}\varphi}R_z(\alpha)R_y(\beta)R_z(\gamma) \qquad \textit{(Z-Y decomposition)} \qquad \text{or} \tag{1.75}$$

$$U = \mathrm{e}^{\mathrm{i}\varphi}R_x(\alpha)R_y(\beta)R_x(\gamma) \qquad \textit{(X-Y decomposition)} \tag{1.76}$$

This means, for symmetry reasons, that for any two orthogonal unit vectors $\hat{u}$ and $\hat{v}$, the operator set $O_{uv} = \{R_{\hat{u}}(\theta)\,|\,\theta \in \mathbf{R}\} \cup \{R_{\hat{v}}(\theta)\,|\,\theta \in \mathbf{R}\}$ is universal for single qubit operations.

Definition 24 (Universality of operator sets) *Let $\mathcal{H}$ be a separable Hilbert space and O a set of unitary operators on $\mathcal{H}$. O is universal on $\mathcal{H}$ if for any unitary operator $U : \mathcal{H} \to \mathcal{H}$ and for any $\epsilon \in \mathbf{R}^+$, there exists a composition $\sigma = U_1 \circ U_2 \circ \ldots \circ U_k$ with $U_i \in O$ and an overall phase $\mathrm{e}^{\mathrm{i}\varphi}$ such that*

$$\left|\langle\phi|(U - \mathrm{e}^{\mathrm{i}\varphi}\sigma)|\chi\rangle\right| < \epsilon \qquad \forall\,|\phi\rangle, |\chi\rangle \in \mathcal{H}. \tag{1.77}$$

[6] Note that despite the mathematical period of 4π, $R_{\hat{n}}(\theta)$ and $R_{\hat{n}}(\theta + 2\pi) = -R_{\hat{n}}(\theta)$ describe the same physical operation.

[7] Note that expanding (1.75) directly leads to (1.68).

Let $O = \{R_{\hat{n}}(\theta) \,|\, \theta \in \mathbf{R}\}$ be the class of rotations about some axis vector $\hat{n}$. Since

$$R_{\hat{n}}(\alpha+\beta) = R_{\hat{n}}(\alpha)R_{\hat{n}}(\beta) \quad \text{and} \quad R_{\hat{n}}(\theta+2k\pi) = (-1)^k R_{\hat{n}}(\theta), \tag{1.78}$$

the set $\{R^k_{\hat{n}}(\xi) \,|\, k \in \mathbf{N}\} \subset O$ is dense in O iff $\xi \neq 0$ and ξ/π is irrational. So any pair $\{R_{\hat{u}}(q\pi), R_{\hat{v}}(p\pi)\}$ of orthogonal qubit rotations where $\hat{u} \perp \hat{v}$ and $p, q \in \mathbf{I}^+$ already constitutes a universal set of qubit operators.

The above result can be generalized to non-orthogonal rotations: Let $V = R_{\hat{v}}(\beta)$ and $W = R_{\hat{w}}(\gamma)$ be unitary qubit operators where $0 < |(\hat{v}, \hat{w})| < 1$ and let $\hat{u}$ be an axis vector orthonormal to $\hat{v}$. Since W can be written as $W = R_{\hat{v}}(\beta_1)R_{\hat{u}}(\alpha)R_{\hat{v}}(\beta_2)$, we can construct a qubit rotation

$$U = R_{\hat{u}}(\alpha) = R_{\hat{v}}(-\beta_1)\, W\, R_{\hat{v}}(-\beta_2) \tag{1.79}$$

orthogonal to V. Provided that $\beta/\pi, \gamma/\pi \in \mathbf{I}^+$, there exist $k_1, k_2 \in \mathbf{N}$ such that $R_{\hat{v}}(-\beta_i) \approx V^{k_i}$ to arbitrary precision and the operator pair $\{V, W\}$ is universal.

1.4.2.4 Quantum Gates

Definition 25 *Let $\mathcal{H} = \mathcal{B}^{\otimes n}$ or $\mathcal{H} = \mathcal{B}^\star$ be the state space of a composite system of the qubits $S = \{q_i\}$ and $R_k(S) = \{\mathbf{s} \subseteq S \,|\, |\mathbf{s}| = k\}$ denote the ordered k-qubit subsets of S i.e. the set of k-qubit registers. The class*

$$\Gamma(U) = \{U(\mathbf{s}) \,|\, \mathbf{s} \in R_k(S)\} \tag{1.80}$$

of register operators on $\mathcal{H}$ for some unitary k-qubit operator on $\mathcal{B}^{\otimes k}$ is called a k-qubit quantum gate.

Informally, we can describe a k-qubit gate as a unitary operator which can be equally applied to any k-qubit register of a quantum computer. In that case, the term "gate" is also used to refer to a single register operator $U(\mathbf{s}) \in \Gamma(U)$ as well as to the operator U itself.

Common Elementary Gates

Single Qubit Gates

- Pauli gates

$$X \equiv \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y \equiv \sigma_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix}, \quad Z \equiv \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.81}$$

- Hadamard gate

$$H \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.82}$$

- Phase- and $\pi/8$-gate[8]

$$S \equiv \sqrt{Z} = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{i} \end{pmatrix}, \quad T \equiv \sqrt{S} = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{\mathrm{i}\pi/4} \end{pmatrix} \tag{1.83}$$

Two Qubit Gates

- controlled-not gate

$$\texttt{CNot} : |x, y\rangle \to |x \oplus y, y\rangle \tag{1.84}$$

- swap-gate

$$\texttt{Swap} : |x, y\rangle \to |y, x\rangle \tag{1.85}$$

- controlled-phase-gate

$$\texttt{CPhase} : |x, y\rangle \to \mathrm{i}^{xy} |x, y\rangle \tag{1.86}$$

Three Qubit Gates

- Toffoli-gate (controlled-controlled-not)

$$\texttt{CCNot} : |x, y, z\rangle \to |x \oplus (y \wedge z), y, z\rangle \tag{1.87}$$

- Fredkin-gate (controlled-swap)

$$\texttt{CSwap} : |x, y, z\rangle \to \begin{cases} |y, x, z\rangle & \text{if } z = 1 \\ |x, y, z\rangle & \text{if } z = 0 \end{cases} \tag{1.88}$$

1.4.2.5 Controlled Gates

The single most important 2-qubit gate is the *controlled-not-gate*. The `CNot`-gate operates on a *target qubit* **t** and a *control (or enable) qubit* **e** and can be defined using the X-gate, as matrix

$$\texttt{CNot} = C[X] = \begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.89}$$

[8] T is named $\pi/8$-gate as $T \simeq \mathrm{e}^{\mathrm{i}\pi\sigma_x/8}$.

or as the register operator

$$\texttt{CNot}(\mathbf{t},\mathbf{e}) : |d\rangle_{\mathbf{t}}|c\rangle_{\mathbf{e}} \to (X^c|d\rangle_{\mathbf{t}}) \otimes |c\rangle_{\mathbf{e}} \tag{1.90}$$

Informally, we can describe the CNot-gate as conditionally applying the operator X (single bit not) to the target qubit $\mathbf{t}$ in dependence of the control qubit $\mathbf{e}$. This can be generalized to arbitrary gates and multiple control-bits:

Definition 26 (Controlled Gate) *Let U be a unitary m-qubit gate. A controlled U-gate with n control qubits is defined as*

$$C^n[U] = \begin{pmatrix} I & \cdots & 0 & 0 \\ \vdots & \ddots & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & U \end{pmatrix} \tag{1.91}$$

on $\mathcal{B}^{\otimes n+m}$ or in register notation

$$U_{[[\mathbf{e}]]}(\mathbf{t}) \equiv C^n[U](\mathbf{t},\mathbf{e}) : |k\rangle_{\mathbf{t}}|c\rangle_{\mathbf{e}} \to \begin{cases} (U\,|k\rangle_{\mathbf{t}})|c\rangle_{\mathbf{e}} & \textit{if } c = 111\ldots \\ |k\rangle_{\mathbf{t}}|c\rangle_{\mathbf{e}} & \textit{otherwise} \end{cases} \tag{1.92}$$

For any single qubit gate U, $C[U]$ can be implemented using single qubit operations and CNot: Let $U = \mathrm{e}^{\mathrm{i}\varphi}R_z(\alpha)R_y(\beta)R_z(\gamma)$ be the ZY-decomposition (1.75) of U, $A = R_z(\alpha)R_y(\beta/2)$, $B = R_y(-\beta/2)R_z(-(\gamma+\alpha)/2)$ and $C = R_z((\gamma-\alpha)/2)$, then

$$U_{[[e]]}(\mathbf{t}) = \mathrm{e}^{\mathrm{i}\varphi}R_z(\varphi)(\mathbf{e})A(\mathbf{t})\texttt{CNot}(\mathbf{t},\mathbf{e})B(\mathbf{t})\texttt{CNot}(\mathbf{t},\mathbf{e})C(\mathbf{t}) \tag{1.93}$$

Phase Gates Operators of the form $V(\varphi) = C^n[\mathrm{e}^{\mathrm{i}\varphi}]$ are referred to as *(controlled) phase gates*, and are an interesting special case as the operator $U = \mathrm{e}^{\mathrm{i}\varphi}$ is a physically irrelevant overall phase and can technically be considered as a zero-qubit gate, so only controlled versions of U have a non-trivial physical effect.

Examples for controlled phase gates are $R_z(\theta) \simeq C[\mathrm{e}^{\mathrm{i}\theta}]$ (see 1.4.2.2), $Z = C[-1]$, $S = C[\mathrm{i}]$, $T = C[\mathrm{e}^{\mathrm{i}\pi/4}]$ and $\texttt{CPhase} = C^2[\mathrm{i}]$ (see 1.4.2.4).

1.4.2.6 Universal Gates

A well known result from classical boolean logic is that any possible function $f : \mathbf{B}^n \to \mathbf{B}^m$ can be constructed as a composition from a small universal set of operators if we can "wire" the inputs and outputs to arbitrary bits in a feed-forward network. Examples for universal sets of logical gates are $\{\vee, \neg\}$, $\{\to, \neg\}$ or $\{\bar{\wedge}\}$.

In 1.4.2.3 we have already demonstrated how almost any pair of single qubit rotations can be used to approximate an arbitrary unitary operator on $\mathcal{B}$.

It can be shown that any n-dimensional unitary matrix can be decomposed into a product of at most $\binom{n}{2} = n(n-1)/2$ *two-layer unitary matrices* [42, 52],[9] i.e.

$$U = \prod_{j=1}^{n-1} \prod_{i=0}^{j-1} U_{ij} \quad \text{where} \quad U_{ij} : |k\rangle \to \begin{cases} a_{ij}|i\rangle + b_{ij}|j\rangle & \text{if } k = i \\ c_{ij}|i\rangle + d_{ij}|j\rangle & \text{if } k = j \\ |k\rangle & \text{otherwise} \end{cases} \tag{1.94}$$

If $\mathcal{H} = \mathcal{B}^{\otimes n}$ is a composition of qubits, then for any single qubit gate U, $C^{n-1}[U]$ is also a two-layer matrix. Using suitable *basis permutations* $\Pi_{ij} : |k\rangle \to |\pi_k\rangle$ with $\pi_i = 2^n - 2$ and $\pi_j = 2^{n-1}$ the two-layer matrices U_{ij} from (1.94) can be written as

$$U_{ij} = \Pi_{ij}^\dagger \, C^{n-1}[V_{ij}] \, \Pi_{ij}, \qquad V_{ij} = \begin{pmatrix} a_{ij} & b_{ij} \\ c_{ij} & d_{ij} \end{pmatrix} \tag{1.95}$$

Since basis permutations essentially implement bijective functions over $\mathbf{B}^n$ (see 2.5.3), any quantum gate which implements a universal reversible boolean gate, together with controlled single qubit operations, is enough to implement arbitrary unitary operators on $\mathcal{B}^{\otimes n}$.

An example for a universal reversible boolean gate is the classical Toffoli gate $T : (x, y, z) \to (x \oplus (y \wedge z))$ [57]. Unlike T, its quantum counterpart (1.87) can be factorized into 2-qubit operators, e.g.

$$\texttt{CCNot}(\mathbf{x},\mathbf{y},\mathbf{z}) = V_{[[\mathbf{z}]]}(\mathbf{x})\texttt{CNot}(\mathbf{y},\mathbf{z})V^\dagger_{[[\mathbf{y}]]}(\mathbf{x})\texttt{CNot}(\mathbf{y},\mathbf{z})V_{[[\mathbf{y}]]}(\mathbf{x})$$
$$\text{with} \quad V = \sqrt{X} = \mathrm{e}^{\mathrm{i}\pi/4} R_x(\frac{\pi}{2}) \tag{1.96}$$

By choosing V such that $U = V^2$ the above factorization can be used to construct $C^2[U]$ for arbitrary arbitrary single qubit-gates. Moreover, similar decompositions can be found for any number of control qubits, so `CNot` and single qubit operations are universal [28].

Further examples for universal sets of quantum gates are

- the *standard set* [43, 13][10]

$$\{H, S, T, \texttt{CNot}\} \tag{1.97}$$

[9]This is similar to the fact that a general rotation in $\mathbf{R}^n$ can be decomposed into $\binom{n}{2}$ simple rotations in the coordinate planes.

[10]Since $S = T^2$, the phase gate (1.83) is merely included for convenience.

The standard set is universal despite H and T being π and $\pi/4$ rotations in the Bloch sphere, because the rotation angle θ of $R_{\hat{n}}(\theta) = TH$ and $R_{\hat{m}}(\theta) = HT$, which is given by $\cos\theta = \cos^2\frac{\pi}{8}$, can be shown to be an irrational multiple of π [13].

- the *Deutsch gate* [25]

$$D = C^2[\mathrm{i}R_x(\theta)] \quad \text{for} \quad \frac{\theta}{\pi} \in \mathbf{I} \tag{1.98}$$

The universality proof involves the construction of the Toffoli gate which can be used to implement arbitrary basis-permutations $\Pi : |i\rangle \to |\pi_i\rangle$. Those can then be used to construct 3-qubit two-layer rotations between any two basis-vectors, which can be shown to be sufficient to construct arbitrary unitary transformations [41].

1.4.3 Input and Output

1.4.3.1 Quantum Computing and Information Processing

As already mentioned in 1.2.1, the ultimate claim of quantum computing is that the interpretation of computing as a physical process, rather than the abstract manipulation of symbols, leads to an extended notion of computability. In accordance with the postulates of quantum mechanics (see 1.2.3), we also identified the concept of unitary transformations as the most general paradigm for "physical computability".

Unitary transformations describe the transition between machine states and thereby the temporal evolution of a quantum system. The very notion of a (quantum) computer as a "computing machine" requires, however, that the evolution of the physical system corresponds to a processing of information.

Classical information theory requires that any "reasonable" information can be expressed as a series of answers to yes-no questions, i.e. a string of bits. But unlike classical symbolic computation, where every single step of a computation can be mapped onto a bit-string, physical computation requires such a labeling only for the initial and the final machine state, the labels of which make up the input and output of the computation.[11]

If we regard a quantum computer as a probabilistic machine $\mathcal{M}$ (see 1.3.2.5 and 1.4), the above requirements are equivalent to the enumerability of the input and output sets $\mathcal{I}$ and $\mathcal{O}$.

[11] This is in accordance with the Copenhagen interpretation of quantum physics, which states that the setup and the outcome of any experiment has to be described in classical terms.

1.4.3.2 Measurement Operators

In the classical machine definition (see 1.3.2), a test command $t \in T$ is a function $t : \mathbf{S} \to \mathbf{B}$. In the case of a quantum computer $\mathcal{M} = (\mathcal{H}, O, T, \delta, \beta)$, however, the machine state $|\Psi\rangle \in \mathcal{H}$ is not directly accessible and any physically realizable test-command will have to amount to the measurement of some observable M.

According to the 3rd postulate of quantum mechanics (see 1.2.3.3), the measurement of M on $|\Psi\rangle$ is only deterministic and invariant to $|\Psi\rangle$ iff $|\Psi\rangle$ happens to be an eigenstate of M, so the test commands are no longer boolean predicates on $\mathbf{S}$ but probabilistic *measurement operators*, i.e.

$$T = \{\mu_i : \mathcal{H} \mapsto \mathcal{H} \times \mathbf{B}\} \tag{1.99}$$

Definition 27 *Let M be a self-adjoint operator on $\mathcal{H}$ with the spectral decomposition $M = \sum_m \lambda_m P_m$, then the measurement operator $\mu[M]$ is a probabilistic mapping $\mu[M] : \mathcal{H} \mapsto \mathcal{H} \times \mathbf{R}$ defined as*

$$\mu[M] : |\Psi\rangle \mapsto \left(\frac{1}{\sqrt{p_m}} P_m |\Psi\rangle, \lambda_m \right) \quad \text{with probability} \quad p_m = \langle\Psi|P_m|\Psi\rangle \tag{1.100}$$

Since test commands are supposed to deliver boolean results, each μ_i corresponds to a projection operator P_i i.e. a Hermitian with the eigenvalues 0 and 1. So $\mu_i = \mu_i[P_i]$ and

$$\mu[P] : |\Psi\rangle \mapsto \left\{ \begin{array}{ccc} (\frac{1}{\sqrt{p}} P|\Psi\rangle, 1) & \text{with} & p = \langle\Psi|P|\Psi\rangle \\ (\frac{1}{\sqrt{p}} (I-P)|\Psi\rangle, 0) & \text{with} & p = \langle\Psi|I-P|\Psi\rangle \end{array} \right. \tag{1.101}$$

If $P = N(\mathbf{s})$ we also write $\mu(\mathbf{s}) \equiv \mu[N(\mathbf{s})]$. Also, we will occasionally ignore one of the function values if this is convenient and can be done without ambiguity.

Single Qubit Measurements If $\mathcal{H} = \mathcal{B}^{\otimes n}$, then a natural choice for P_i are the standard observable

$$N(\mathbf{q}_i) = I^{\otimes i} \otimes |1\rangle\langle 1| \otimes I^{\otimes n-i-1} \tag{1.102}$$

for each qubit $\mathbf{q}_i$. If $\mathcal{M}$ provides a universal set O of unitary operators, single qubit measurements in the computational basis are sufficient to measure an arbitrary P:

A general projection P has the form

$$P = \sum_n \lambda_n |\tilde{n}\rangle\langle\tilde{n}| \quad \text{with} \quad \lambda_n \in \mathbf{B} \quad \text{and} \quad |\tilde{n}\rangle \in \tilde{B} \tag{1.103}$$

where $\tilde{B}$ is an arbitrary orthonormal basis of $\mathcal{H}$. If O is universal, then it is possible to implement the unitary operator $U = \sum_n |n\rangle\langle\tilde{n}|$ and P can be expressed as $P = U^\dagger P' U$ with $P' = \sum_n \lambda'_n |n\rangle\langle n|$.

To measure P', we can use an additional scratch qubit $\mathbf{s}$ in state $|0\rangle_{\mathbf{s}}$ and use the unitary operator

$$U' : |k\rangle_{\mathbf{s}} |n\rangle_{\bar{\mathbf{s}}} \to |k \oplus \lambda'_n\rangle_{\mathbf{s}} |n\rangle_{\bar{\mathbf{s}}} \tag{1.104}$$

to prepare the entangled machine state

$$|\Psi'\rangle = \sum_n c_n |\lambda'_n\rangle_{\mathbf{s}} |n\rangle_{\bar{\mathbf{s}}} \quad \text{where} \quad c_n = \langle n|\Psi\rangle \tag{1.105}$$

A measurement of $N(\mathbf{s})$ on $|\Psi'\rangle$ is now equivalent to to measuring P' on $|\Psi\rangle$. If the result is 1 and therefore $\mathbf{s}$ is left in the state $|1\rangle_{\mathbf{s}}$, the previous state can be restored by applying $X(\mathbf{s})$.

Output Function To retrieve the classical result of the computation, a final measurement is required. Using the standard observable N on $\mathcal{H}$, we can define the probabilistic output function $\beta : \mathcal{H} \mapsto \mathbf{N}$ as $\beta = \mu[N]$.[12]

1.4.3.3 State Preparation

To set or reset a quantum computer $\mathcal{M}$ to desired initial state $|\Psi_0\rangle$, no additional operations besides unitary transformations and measurements are necessary. Assuming $\mathcal{H} = \mathcal{B}^{\otimes n}$, it suffices to measure all qubits to bring $\mathcal{M}$ into a known state $|\Psi\rangle = |m\rangle$ and then to apply an arbitrary unitary operator U_m which satisfies the condition $\langle\Psi_0|U_m|m\rangle = 1$.

If $|\Psi_0\rangle = |d_0 d_1 \ldots d_{n-1}\rangle$ then at most n X-gates are required for the preparation.[13] It is also convenient to include a special non-unitary memory command

$$\texttt{reset} : |\Psi\rangle \to |0\rangle \tag{1.106}$$

for the initialization of the machine state.

[12] This notation is actually a shorthand for $\beta(|\Psi\rangle) = m \iff \mu[N]|\Psi\rangle = (|\Psi_m\rangle, m)$.

[13] For arbitrary $|\Psi_0\rangle$, the number of necessary gates generally increases exponentially.

Input Function To allow the preparation of “classical” input states we can define the input function $\delta(s) : \mathbf{N} \to \mathcal{H}$ as $\delta(s) = |s\rangle$. For quantum algorithms which take their input in the form of oracle operators[14] and consequently do not require any classical input, it is common to assume an initial state $|\Psi_0\rangle = \delta(s) = |0\rangle$.

1.5 Concepts of Quantum Computation

1.5.1 Models and Formalisms

As we demonstrated in 1.3 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* (1.3.1.1), computer scientists often use the *Turing machine* (1.3.2.2) as their favorite model, an electronic engineer would possibly speak of *logic circuits* while a programmer probably will prefer a *universal programming language* (1.3.3.2).

As for quantum computation, each of these classical concepts has a quantum counterpart: [47, 48]

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

Table 1.2: *Classical and quantum computational models*

1.5.1.1 The Mathematical Model

The paradigm of computation as a physical process requires that algorithms can — in principle — be described by the same means as any other physical system, which, for the field of quantum physics, is the mathematical formalism of Hilbert space algebra. The basics of this formalism, were introduced in 1.2.

The quantum equivalent of partial recursive functions is unitary operators. Just as every classically computable problem can be reformulated as

[14] An oracle function or operator is a special “black-box” memory or test command which can be used either as problem description or to extend the functionality of a machine.

calculating the value of a partial recursive function, every quantum computation can be described by a unitary operator.[15]

The mathematical description of an operator is inherently declarative; the actual implementation for a certain quantum architecture i.e. the algorithmic decomposition into elementary operations, is beyond the scope of this formalism. Also, since the mathematical model treats unitary operators as black boxes, no complexity measure is provided.

Register Notation To simplify the discussion of operators applied to permutations of qubits on $\mathcal{H} = \mathcal{B}^{\otimes n}$ and $\mathcal{H} = \mathcal{B}^{\star}$ we introduced the concept of quantum registers (see 1.4.1.3). Table 1.3 summarizes all important register expressions.

Notation	Description
$\mathbf{s}$	general register i.e. an ordered set of qubits
$\bar{\mathbf{s}}$	complementary register to $\mathbf{s}$
$\vert\ldots\rangle_{\mathbf{s}}$	ket-vector of register $\mathbf{s}$
$\vert\mathbf{s}\vert$	length of $\mathbf{s}$ i.e. number of qubits in $\mathbf{s}$
$\mathcal{H}_{\mathbf{s}}$	$\vert\mathbf{s}\vert$-qubit sub-space $\{\vert\psi\rangle_{\mathbf{s}}\}$ of $\mathcal{H}$, $\mathcal{H}_{\mathbf{s}} \otimes \mathcal{H}_{\bar{\mathbf{s}}} = \mathcal{H}$
$\mathbf{a} \circ \mathbf{b}$	concatenation of registers $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{a} \circ \mathbf{b} \neq \mathbf{b} \circ \mathbf{a}$
$U(\mathbf{s})$	register operator $U \otimes I$ on $\mathcal{H}_{\mathbf{s}} \otimes \mathcal{H}_{\bar{\mathbf{s}}}$
$\mu(\mathbf{s})$	stochastic measurement operator $\mu[N(\mathbf{s})] : \mathcal{H} \mapsto \mathcal{H} \times \mathbf{B}^{\vert\mathbf{s}\vert}$
$\rho_{\mathbf{s}}$	register density operator $\rho_{\mathbf{s}} = \mathrm{tr}_{\bar{\mathbf{s}}}\left(\vert\Psi\rangle\langle\Psi\vert\right)$
$U(\mathbf{a}, \mathbf{b})$	multi-register operator $U(\mathbf{a} \circ \mathbf{b})$
$U_{[[\mathbf{b}]]}(\mathbf{a})$	conditional register operator $C^{\vert\mathbf{b}\vert}[U](\mathbf{a} \circ \mathbf{b})$

Table 1.3: *Register Notation*

1.5.1.2 Quantum Turing Machines

In analogy to the classical Turing Machine (TM) several types of Quantum Turing Machines (QTM) have been proposed as a model of a universal quantum computer [5, 24, 6, 9].

The complete machine-state $|\Psi\rangle \in \mathcal{H}$ of a QTM is given by a superposition of basis states $|l, j, s\rangle$, where $l \in \mathbf{Z}_n$ is the state of the head, $j \in \mathbf{N}$ the head position and

$$s = (\ldots s_{-2}s_{-1}|s_0 s_1 s_2 \ldots) \tag{1.107}$$

[15] This assumes that any measurements are performed at the end of the computation. There are however algorithms, which take advantage of the state-reduction inherent to quantum measurement so the analogy is not universal.

the binary representation of the tape-content. To keep $\mathcal{H}$ separable, s has to meet the *zero tail state* condition (see 1.3.2.2) i.e. only a finite number of bits with $s_m \neq 0$ are allowed, so $s \in \mathbf{B}^\star$ and $\mathcal{H} = \mathbf{C}^n \otimes \mathbf{l}_2 \otimes \mathcal{B}^\star$

The quantum analogue of the transition function of a classical probabilistic TM is the unitary *step operator* T, which has to meet locality conditions for the affected tape-qubit, as well as for head movement.[16]

QTMs provide a measure for execution times, but — as with the classical TM — finding an appropriate step operator can be very hard.

1.5.1.3 Quantum Circuits

Quantum circuits are the quantum equivalent to classical boolean feed-forward networks, with one major difference: since all quantum computations have to be unitary, quantum circuits can be evaluated in both directions (as in classical reversible logic). Quantum circuits are composed of elementary gates and operate on qubits.[17] The "wiring" between the gates defines the register on which the gates operate, so an m-qubit gate U in an n-qubit circuit can describe up to $\frac{n!}{(n-m)!}$ different unitary transformations $U(\mathbf{s})$.

As opposed to the mathematical formalism, the gate-notation is an inherently constructive method and the complexity of the problem is directly reflected in the number of gates necessary to implement it. However, since quantum circuits describe static sequences, the size of the input as well as the number of qubits is fixed, so without additional assumptions, quantum circuits cannot be used to analyze the complexity depending on the size of the problem. For the same reason, quantum circuits are also inadequate for machines with unlimited memory.

Restrictions In comparison with classical boolean feed-forward networks, this imposes the following restrictions:

- Only n-to-n networks are allowed i.e. the total number of inputs has to match the total number of outputs.
- Only n-to-n gates are allowed.

[16] Instead of a unitary step-operator T, it is also possible to directly construct a (local) Hamiltonian H (see 1.2.3.2) [5, 6]. In this case, the computation does not need to be discrete and $T' = U(t_0) = \mathrm{e}^{-\mathrm{i}Ht_0}$ is not required to conform to locality conditions.

[17] There also exist extentions to cover measurements and classical bits [43]. In that case, quantum circuits can also be used to describe irreversible computations.

- No forking of inputs is allowed. This is directly related to the fact that qubits cannot be copied, i.e. that there exists no unitary operation

$$\mathrm{Copy}\,|\psi\rangle|0\rangle \to |\psi\rangle|\psi\rangle \quad \text{with} \quad |\psi\rangle \in \mathbf{C}^2 \tag{1.108}$$

  which can turn a general qubit-state into a product state of itself.

- No "dead ends" are allowed. Again, this is because the erasure of a qubit

$$\mathrm{Erase}\,|\psi\rangle \to |0\rangle \quad \text{with} \quad |\psi\rangle \in \mathbf{C}^2 \tag{1.109}$$

  is not a unitary operation.

Notation Some common gates (see 1.4.2.4) have special symbols. The circuit in fig. 1.3 implements the operator[18]

$$C[\mathrm{e}^{\mathrm{i}\theta}](\mathbf{a})\,\mathtt{CSwap}(\mathbf{a},\mathbf{b},\mathbf{c})\,\mathtt{Swap}(\mathbf{a},\mathbf{b})\,\mathtt{CCNot}(\mathbf{a},\mathbf{b},\mathbf{c})\,\mathtt{CNot}(\mathbf{a},\mathbf{b})\,U_{[[\mathbf{b}]]}(\mathbf{a})\,U(\mathbf{a}) \tag{1.110}$$

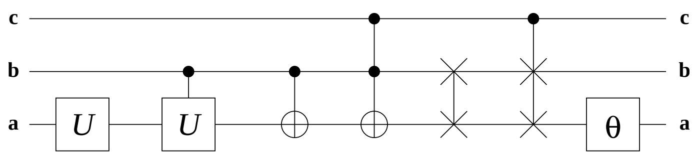


Figure 1.3: *Circuit notation for common gates*

1.5.1.4 Quantum Programming Languages

A possible way to generalize quantum circuits for arbitrary input sizes is to use a classical computer with unlimited memory (such as a TM) to generate the circuits depending on the size of the input. So instead of directly specifying a single circuit in terms of wires and gates, a whole class of quantum circuits is specified by means of a classical program.

Quantum programming languages (QPLs) take this abstraction even further by directly using a quantum computer $\mathcal{M}_q$ as a oracle (see 1.3.2.4) for a classical machine $\mathcal{M}_c$. This not only avoids the need for an intermediate circuit-description, but also allows the computation to depend on previous measurements so quantum programs can describe complete algorithms and not merely unitary transformations.

[18] Note that the order of the operators is inverted.

1.5.2 Quantum Algorithms

1.5.2.1 Classical and Quantum Computability

If we consider a finite quantum computer with the Toffoli gate (see 1.4.2.4) as the only available instruction, then any transformation of the machine state has to be of the form

$$|\Psi\rangle = |i\rangle \longrightarrow |g(i)\rangle = |\Psi'\rangle \quad \text{with} \quad g : \mathbf{B}^n \to \mathbf{B}^n \tag{1.111}$$

Since the Toffoli gate is universal for reversible boolean logic, any bijective binary function g can directly be implemented on a quantum computer.

A general binary function f on $\mathbf{B}^n$, can be implemented by an arbitrary unitary operator F which satisfies the condition $F\,|i, 0\rangle = |i, f(i)\rangle$.

So any function f computable on a (finite) classical machine can also be implemented on a quantum computer with a universal set of gates. Moreover, C. H. Bennet has shown that a reversible implementation of f can be made with a maximal overhead of $O(2)$ in time and $O(\sqrt{n})$ in space complexity [7].

On the other hand, a general n-qubit quantum state consists of maximally 2^n basis-vectors with a non-zero amplitude and can consequently be described by an array of 2^n complex numbers. Also, any m-qubit quantum gate can be described by a complex $2^n \times 2^n$ matrix with the elements $u_{ij} = \langle i|U|j\rangle$.

By encoding the complex amplitudes as a pair of floating point binary numbers a classical computer can simulate any unitary operator to arbitrary precision.[19] This will generally require an overhead of $O(\mathrm{e}^n)$ in time as well as in space complexity. Due to the stochastic nature of quantum measurements, the emulating computer will also need a source of true randomness (like e.g. the probabilistic Turing machine).

So classical and quantum computers are computationally equivalent, but while it is possible to efficiently simulate a classical computer on a quantum computer, the opposite case can involve an exponential overhead. Therefore, while not extending our notion of computability, for certain tasks quantum algorithms might provide a more efficient solutions than classical implementations.

1.5.2.2 Deutsch's Algorithm

In 1985, Deutsch proposed a probabilistic algorithm [24] which for some oracle function $g : \mathbf{B} \to \mathbf{B}$ allows to compute $g(0) \oplus g(1)$ with a probability of 1/2 using only 1 application of G:

[19] The linearity of unitary transformations assures that small errors will not escalate.

Let $G : |x, y\rangle \to |x, y \oplus g(x)\rangle$ be a 2-qubit oracle-operator implementing the boolean oracle function $g : \mathbf{B} \to \mathbf{B}$.

1. Prepare an empty initial state $|\Psi_0\rangle = |0\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}$.

2. Apply $H(\mathbf{x})$
$$|\Psi_1\rangle = (H\,|0\rangle_{\mathbf{x}})\,|0\rangle_{\mathbf{y}} = \frac{1}{\sqrt{2}}\left(|0\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} + |1\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}\right) \tag{1.112}$$

3. Apply the oracle operator G, giving
$$|\Psi_2\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_{\mathbf{x}}|g(0)\rangle_{\mathbf{y}} + |1\rangle_{\mathbf{x}}|g(1)\rangle_{\mathbf{y}}\right) \tag{1.113}$$

4. Apply $H(\mathbf{x} \circ \mathbf{y})$, resulting in
$$|\Psi_3\rangle = \frac{1}{2\sqrt{2}} \sum_{x\in\mathbf{B}} \sum_{y\in\mathbf{B}} \left((-1)^{yg(0)} + (-1)^{x+yg(1)}\right) |x\rangle_{\mathbf{x}}|y\rangle_{\mathbf{y}} \tag{1.114}$$

5. Measure $\mathbf{x}$ and $\mathbf{y}$.

As (1.114) can be simplified to
$$|\Psi_3\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} + (-1)^{g(0)}|g(0) \oplus g(1)\rangle_{\mathbf{x}}|1\rangle_{\mathbf{y}}\right) \tag{1.115}$$
the register $\mathbf{x}$ will contain the value $g(0)\oplus g(1)$ whenever 1 has been measured in $\mathbf{y}$ which will happen in 50% of the cases.

While strictly speaking, this does not provide any speedup over the classical case, if we take into account that, on average, two tries are required to actually measure $g(0) \oplus g(1)$, Deutsch's algorithm was the first proof that quantum computers are capable of performing computations in ways that are impossible on a classical computer.[20]

Generally, in order to achieve any speedup over classical algorithms, it is necessary to take advantage of the unique features of quantum computing, namely

- Superposition (step 2)
- Quantum Parallelism (step 3)
- Interference (step 4)

[20]There exist several improvements and generalizations to the original version of Deutsch's algorithm [26, 20], one of which — the Deutsch-Jozsa Algorithm — is described in 1.5.2.6.

1.5.2.3 Superposition

A key element in any universal programming language is conditional branching. Any classical program (see 1.3.3) can be modeled as a decision tree where each node corresponds to a binary state s_n and leads to one or more successor states $s_{n+1}^{(i)}$. On a deterministic Turing machine (TM), only one of those transitions $s_n \to s_{n+1}^{(k)}$ is possible, so the computational path $\langle s_0, s_1, \ldots s_n \rangle$ is predetermined.

On a probabilistic TM (see 1.3.2.5), the transitions are characterized by probabilities p_i with $\sum_i p_i = 1$ and one of the possible successor states $s_{n+1}^{(i)}$ is chosen accordingly at random.

Since the basis-vectors $|i\rangle$ directly correspond to classical binary states, we might interpret a unitary transformation

$$U : |s\rangle \to \sum_{s'} u_{ss'} |s'\rangle \quad \text{with} \quad s, s' \in \mathbf{B}^n \quad \text{and} \quad u_{ss'} \in \mathbf{C} \tag{1.116}$$

as a probabilistic transition form the classical state s to the successor states s' with the transition probabilities $p_{s'} = |u_{ss'}|^2$, but unless we perform a measurement, the resulting machine state remains in a superposition of all possible classical successor states

$$|\Psi\rangle = |s_n\rangle \xrightarrow{U} |\Psi'\rangle = \sum_i u_{s_n s_{n+1}^{(i)}} |s_{n+1}^{(i)}\rangle \tag{1.117}$$

So from a classical point of view, we can consider a unitary operator which transforms an eigenstate into a superposition of n eigenstates with nonzero amplitudes as a 1–n fork-operation, which enables a quantum computer to follow several classical computational paths at once.

Hadamard-Transform Most non-classical algorithms take advantage of this feature by bringing a register into a even superposition of all basis-states to serve as search space. In Deutsch's algorithm, this is achieved by applying the Hadamard-gate on the first qubit-register.

The H-gate can be generalized to n-qubit registers, by applying H to each individual qubit. The resulting unitary operator is called *Hadamard transform* and defined as:

$$H : |x\rangle \to 2^{\frac{n}{2}} \sum_{y \in \mathbf{B}^n} (-1)^{x \cdot y} |y\rangle \tag{1.118}$$

where $x \cdot y = \sum_i x_i y_i$ denotes the binary inner product. The transformed basis

$$\tilde{B} = \{H|x\rangle \,|\, x \in \mathbf{B}^n\} = \{|+\rangle, |-\rangle\}^n \quad \text{with} \quad |\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle) \tag{1.119}$$

is sometimes referred to as the *dual basis*.

Classically, the Hadamard transform of $|\Psi\rangle = |0\rangle$ can be viewed as a binary decision tree with a 50% chance for each bit to flip. For an n-qubit register, this leads to 2^n classical computational paths all of which are followed simultaneously resulting in a superposition of 2^n eigenvectors.

1.5.2.4 Quantum Parallelism

If we restrict unitary transformations to basis-permutations (i.e. operators of the form $|i\rangle \to |\pi_i\rangle$ then the classical decision tree degenerates into a list and we end up with the functionality of a classical deterministic reversible computer i.e. for any bijective binary function $f : \mathbf{B}^n \to \mathbf{B}^n$ there is a corresponding unitary operator

$$F : |s\rangle \to |f(s)\rangle \quad \text{with} \quad s \in \mathbf{B}^n. \tag{1.120}$$

The restriction to bijective functions is not a severe as it seems, since for any general binary function $g : \mathbf{B}^n \to \mathbf{B}^m$ a corresponding *quantum function*

$$G : |s, 0\rangle \to |s, g(s)\rangle \quad \text{with} \quad s \in \mathbf{B}^n \tag{1.121}$$

can be constructed, which implements g with a maximum overhead of $O(\sqrt{n})$ in space- and $O(2)$ time-complexity.

However, if we use a quantum function on an superposition of eigenstates, the same classical computation is performed on all bit-strings simultaneously.

$$G \sum_s c_s |s, 0\rangle = \sum_s c_s G |s, 0\rangle = \sum_s c_s |s, g(s)\rangle \tag{1.122}$$

In classical terms, this can be described as a SIMD (single instruction, multiple data) vector operation, in quantum terms this feature is referred to as *quantum parallelism*.

In Deutsch's algorithms, quantum parallelism is exploited by applying G on the superposition $|\Psi_1\rangle = (|00\rangle + |10\rangle)/\sqrt{2}$.

1.5.2.5 Interference

While superpositions and quantum parallelism allow us to perform an exponentially large number of classical computations in parallel, the only way to read out any results is by performing a measurement whereby all but one of the superpositioned eigenstates get discarded. Since it does not make any difference if the computational path is determined during the calculation (as with the probabilistic TM) or a-posteriori (by quantum measurement), the use of quantum computers would not provide any advantage over probabilistic classical computers.

Quantum states, however, are not merely a probability distribution of binary values but complex vectors i.e. each basis-state in a superposition is not characterized by a real probability, but a complex amplitude, so

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{1.123}$$

describe different states, even if they have the same probability spectrum.

So, while on a probabilistic TM, the probabilities of two different computational paths leading to the same final state s simply add up, this is not necessarily the case on a quantum computer since generally

$$|\alpha + \beta|^2 \neq |\alpha|^2 + |\beta|^2 \quad \text{for} \quad \alpha, \beta \in \mathbf{C}. \tag{1.124}$$

To illustrate this concept, consider the three states

$$|\psi_1\rangle = |0\rangle, \quad |\psi_2\rangle = |1\rangle \quad \text{and} \quad |\psi_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle). \tag{1.125}$$

If we apply the Hadamard-transform H to the basis states $|\psi_1\rangle$ and $|\psi_2\rangle$ we get

$$|\psi_1'\rangle = H\,|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |\psi_2'\rangle = H\,|\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{1.126}$$

Since $|\psi_1'\rangle$ and $|\psi_2'\rangle$ have the same probability distribution and $|\psi_3\rangle$ is merely a superposition of $|\psi_1\rangle$ and $|\psi_2\rangle$, classically we would assume that $|\psi_3'\rangle$ also shows the same probability spectrum, however

$$|\psi_3'\rangle = H\,|\psi_3\rangle = \frac{1}{\sqrt{2}}(|\psi_1'\rangle + |\psi_2'\rangle) = |0\rangle \tag{1.127}$$

so in case of $|0\rangle$ the probabilities added up while in case of $|1\rangle$, the complex amplitudes had opposing signs leading to a partial probability of 0. This phenomenon is referred to as positive or negative *interference*.

So while the computational paths on a probabilistic TM are independent, interference allows computations on superposition states to interact and it is this interaction which allows a quantum computer to solve certain problems more efficiently than classical computers. The foremost design principle for any quantum algorithm therefore is to use interference to increase the probability of "interesting" basis states while trying to reduce the probability of "uninteresting" states, in order to improve the chance that a measurement will pick one of the former.

Basis Transformations Since any unitary operator U can also be regarded as a basis transformation, the above problem can also be reformulated as finding an appropriate observable for the measurement, thereby effectively replacing the standard observable N by the Hermitian operator

$$M = \tilde{N} = U\, N\, U^{\dagger} = \sum_n n|\tilde{n}\rangle\langle\tilde{n}|. \tag{1.128}$$

This view is especially useful, if global properties of classical functions such as periodicy are of interest for the problem.

In Deutsch's algorithms, the final measurement is performed in the dual basis (1.119), which allows $g(0) \oplus g(1)$ to be extracted in a single measurement.

1.5.2.6 Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm [26] is an improved and generalized version of the original algorithm described in (see 1.5.2.2). For $n = 1$ it gives a deterministic solution to Deutsch's problem:

Let $F : |x, y\rangle \to |x, y \oplus f(x)\rangle$ be an $(n+1)$-qubit oracle-operator implementing a boolean function $f : \mathbf{B}^n \to \mathbf{B}$. Let us further assume that we know that f is either *constant* ($(\forall x) f(x) = b$ where $b \in \mathbf{B}$) or *balanced* ($\sum f(x) = 2^{n-1}$). Classically, $2^{n-1}+1$ evaluations of f would be necessary to find out which of the possibilities applies. The following quantum algorithm can solve this decision problem with a single application of F:

1. Prepare initial state $|\Psi_0\rangle = |0\rangle_{\mathbf{x}}|1\rangle_{\mathbf{y}}$.

2. Apply $H(\mathbf{x} \circ \mathbf{y})$ to set up the search space *superposition*

$$|\Psi_1\rangle = 2^{-\frac{n+1}{2}} \sum_{x \in \mathbf{B}^n} |x\rangle_{\mathbf{x}} \left(|0\rangle_{\mathbf{y}} - |1\rangle_{\mathbf{y}}\right). \tag{1.129}$$

3. Apply the oracle operator, giving *quantum parallelism*

$$|\Psi_2\rangle = 2^{-\frac{n+1}{2}} \sum_{x \in \mathbf{B}^n} (-1)^{f(x)} |x\rangle_{\mathbf{x}} \left(|0\rangle_{\mathbf{y}} - |1\rangle_{\mathbf{y}}\right). \tag{1.130}$$

4. Apply $H(\mathbf{x})$, which results in *interference*

$$|\Psi_3\rangle = \sum_{z \in \mathbf{B}^n} c_z |z\rangle_{\mathbf{x}} \otimes \frac{1}{\sqrt{2}} \left(|0\rangle_{\mathbf{y}} - |1\rangle_{\mathbf{y}}\right), \quad c_z = 2^{-n} \sum_{x \in \mathbf{B}^n} (-1)^{x \cdot z + f(x)} \tag{1.131}$$

5. Measure $\mathbf{x}$.

The probability to measure 0 in $\mathbf{x}$ is $|c_0|^2$ where

$$c_0 = 2^{-n} \sum_{x \in \mathbf{B}^n} (-1)^{f(x)} \tag{1.132}$$

If f is constant $c_0 = \pm 1$, if f is balanced, the summands cancel out and $c_0 = 0$, so $\mu(\mathbf{x})|\Psi_3\rangle = 0 \iff f$ is constant.

Chapter 2

Structured Quantum Programming

2.1 Introduction

2.1.1 Motivation

As quantum computing is on its way to becoming an established discipline of computing science, much effort is being put into the development of new quantum algorithms. This research has usually not got much in common with the experimental work on quantum computers and is rarely tied to a specific hardware, but instead employs an abstract notion of a quantum computer with qubits, registers, and a small set of suitable elementary operations [3], which allows one to concentrate on the problem at hand.

A programming language which integrates these abstractions by design should be a useful tool in situations where the much more general physical formalism of Hilbert space algebra gets unnecessarily complex.

The possibility to formulate and simulate an algorithm in a programming language should also make it easier to optimize the implementation with regard to different quantum architectures and allow for more accurate estimates of time and memory complexity.

2.1.2 Quantum Programming Languages

From a software engineering point of view, we can regard the algebraic formalism as a specification language, as the mathematical description of a quantum algorithm is inherently declarative and provides no means to derive a unique decomposition into elementary operations for a given quantum hardware.

Low level formalisms as e.g. quantum circuits [25], on the other hand, are usually restricted to specific tasks, such as the description of unitary transformations, and thus lack the generality to express all aspects of non-classical algorithms.

The purpose of programming languages is therefore twofold, as they allow the expression of a computation's semantics in an abstract manner, as well as the automated generation of a sequence of elementary operations to control the computing device. Any useful quantum programming language (QPL) therefore needs to be

- **constructive**
- **hardware independent**
- **provide arbitrary levels of abstraction**
- **integrate non-classical features at a semantic level**

While the first three requirements equally apply to classical and quantum programming languages, QPLs also have to reflect the peculiarities of quantum computing, as e.g.

- reversibility of unitary operations
- non-locality of qubits
- non-observability of states
- destructive nature of measurement
- lack of an erase operation

at a design level and consequently have to provide the means to take advantage of these features (e.g. by allowing to run code "in reverse").[1]

2.1.3 Classification of Programming Languages

In traditional CS, programming languages can be categorized as either logical (e.g. Prolog), functional (e.g. LISP) or imperative (e.g. Assembler, Fortran, Pascal, C), the latter being the most widely used, for the description of algorithms, as well as for the actual implementation of real world programs.

[1]Classical languages with additional quantum directives (like e.g.. a C++ simulation or device driver API) are therefore not considered to be QPLs, as they do not generically support these non-classical concepts [61].

2.1.3.1 Imperative Programming Languages

Imperative programming is centered around the concept of a computational state, and commands modifying the state [1]. The state is an abstraction for the modifiable storage of the underlying machine model and is semantically expressed in terms of symbolic variables of various *data types* and the current state of control (instruction counter, stack pointer, etc.), which may or may not have a direct representation within the language.

During its execution, a program generates a sequence of states. The transition from one state to the next is determined by

- **assignment commands**, which modify the state of variables, and
- **sequencing commands**, which modify the state of control.

The direct correspondence between state chances and program statements leads to the concept of *flow of control*, i.e. a computation can be characterized by the actual sequence of commands. This allows an imperative program to be traced by following the current state of control and introduces a notion of locality of execution.

Examples of imperative programming languages are assembly language or BASIC. Concepts of imperative programming are reflected in the hybrid quantum architecture (2.2.3) and the concept of symbolic quantum registers (2.4.1).

2.1.3.2 Procedural Programming Languages

Imperative programming combined with parameterized subroutines (procedures, functions) it is called procedural programming. The availability of subroutines allows for

- **Functional Abstraction:** Code of similar or identical functionality can be generalized to reusable parameterized procedures.
- **Hierarchical Program Structure:** Subroutines can contain calls to other subroutines which provides arbitrary levels of abstraction and supports a top-down approach in software design.
- **Recursion:** Subroutines can also contain calls to themselves which allows for an elegant and efficient implementation of otherwise complicated control structures and directly supports the classical "divide and conquer" approach in software design.

- **Private Scopes:** Subroutines provide their own namespace which reduces interdependencies within the program.
- **Local Variables:** Local namespaces also allow the use of temporary variables of limited lifetime which leads to more efficient usage of the available storage and allows details of the implementation to be hidden from the procedure's calling interface.

Examples of procedural programming languages are Fortran and C. Concepts of procedural programming are reflected in quantum data types (2.4.1.3, 2.5.2.2), quantum subroutines (2.5.1) and scratch space management (2.5.3.5) [47].

2.1.3.3 Structured Programming Languages

An mentioned in 2.1.3.1, imperative programs consist of assignment and sequencing commands. A simple way to provide flow-control is by means of a `goto` command which explicitly transfers control to a labeled command which is to be executed next.

```
goto label;
if condition then goto label;
```

This approach is problematic as it implies a flat control structure where any labeled command can be jumped to from anywhere within the program (or, in the case of procedural languages, the current subroutine). This tends to make larger programs unreadable ("spaghetti code").

In 1966, Corrado Bohm and Guiseppe Jacopini showed [11] that the `goto` command can be replaced by the nesting and stacking (sequencing) of three basic control statements

- **sequence** (block-statements),
- **selection** (if-statements) and
- **iteration** (conditional loops),

with well defined entry- and exit-points. This approach, called structured programming [27, 23], implies a strictly hierarchical control-structure which not only makes programs easier to understand, but also provides useful meta-information for the compiler.

Examples of structured programming languages are Pascal and Modula, but almost any procedural language supports the necessary control statements, even if their exclusive use is not enforced. Concepts of structured programming are reflected in quantum conditions (2.6.3), conditional operators (2.6.1) and quantum if-statements (2.6.2.1).

2.1.4 Goals

Functional abstraction and hierarchical control-structure, together with the very intuitive notion of a flow of control, absent in merely declarative formalisms, seem to fit most people's way of reasoning about computational tasks. This makes structured programming a powerful formalism, not only for the actual task of coding, but also for the description of algorithms in general (flow-charts, pseudo-code).

In the remainder of this chapter we will show how familiar concepts of classical structured programming can be adopted to the field of quantum computing. The programming language QCL [45, 49] will serve as an example to illustrate the principles of structured quantum programming.

Table 2.1 gives an overview of quantum language elements along with their classical semantic counterparts.

Classical concept	**Quantum analogue**
classical machine model	hybrid quantum architecture
variables	quantum registers
variable assignments	elementary gates
classical input	quantum measurement
subroutines	operators
argument and return types	quantum data types
local variables	scratch registers
dynamic memory	scratch space management
boolean expressions	quantum conditions
conditional execution	conditional operators
selection	quantum if-statement
conditional loops	quantum forking

Table 2.1: *Classical and quantum programming concepts*

2.1.5 State-of-the-Art

2.1.5.1 QC Software and Simulators

As the general interest in quantum computing increased during the last few years, so did the number of available simulators and computational tools. J. Wallace ran a detailed survey [61] on 21 quantum computer simulators. A more recent list naming 36 projects can be found online [62].

The available software can be roughly categorized as

- class libraries for existing classical languages
- packages for computer algebra systems
- quantum circuit simulators
- simulations of specific quantum hardware
- simulations of specific algorithms
- other quantum simulations (QTMs, quantum Bayesian nets, etc.)

2.1.5.2 Quantum Programming Languages

Besides QCL [49], there have been at least 3 attempts to design a quantum programming language

- 1996 Q-gol by Greg Baker [2]
- 2000 qGCL by Paolo Zuliani [63]
- 2002 Quantum C Language by Stephen Blaha [10]

From those, the most evolved project is probably Zuliani's qGCL. In his thesis [63] Zuliani proposes an abstract formalism with rigorous semantics and an associated refinement calculus which allows for program derivation and strict proof of correctness. There is, however, no interpreter or compiler available.

2.2 The Computational Model of Quantum Programming

In this section we will demonstrate why quantum circuits and finite programs are insufficient for universal quantum computation and introduce the hybrid quantum architecture as the computational model of quantum programming.

2.2.1 Quantum Circuits

2.2.1.1 Deterministic Sequential Algorithms

In its simplest form, a quantum algorithm $\mathcal{P}$ merely consists of a unitary transformation and a subsequent measurement of the resulting state $|\Psi'\rangle$ in the computational basis B. If $|\Psi'\rangle = |y\rangle$ is a basis-state, then the outcome of the measurement is predetermined and the algorithm $\mathcal{P}$ is deterministic. An example for this class is the Deutsch-Jozsa algorithm presented in 1.5.2.6.

For fixed problem sizes[2], deterministic sequential algorithms can be completely described as quantum circuits (see 1.5.1.3) or, equivalently, as sequences (see 1.3.3.1) for a quantum computer $\mathcal{M}$. The Deutsch-Jozsa algorithm e.g. can be implemented by the following commands (assuming an initial state of $|0,0\rangle$).

$$\sigma_{\mathrm{DJ}} = X(\mathbf{y}) \circ H(\mathbf{x}) \circ H(\mathbf{y}) \circ F(\mathbf{x},\mathbf{y}) \circ H(\mathbf{x}) \circ \mu(\mathbf{x}) \tag{2.1}$$

2.2.1.2 General Probabilistic Algorithms

For many computational problems, efficient quantum implementations have the form of probabilistic algorithms. Fig. 2.1 shows the basic outline of a probabilistic quantum algorithm with a single evaluation loop.

An example for this simple case is Deutsch's algorithm presented in 1.5.2.2. More complex quantum algorithms, as e.g. Shor's algorithm for quantum factoring [54, 30], can also include classical random numbers, partial measurements, nested evaluation loops and multiple termination conditions; thus the actual quantum operations are embedded into a *classical control structure*

As quantum circuits are feed-forward networks and lack internal flow-control, they cannot provide a complete description of probabilistic algorithms. Even in the simple case depicted in fig. 2.1, the decision whether a measurement value is "good" i.e. provides a solution of the computational problem, is outside the scope of the formalism and requires additional assumptions.

2.2.2 Finite Quantum Programs

In 1.3.3.2 we introduced the concept of *finite programs* which define finite automata to determine the flow of control of a classical machine $\mathcal{M}$. The same concept can also be used to describe probabilistic quantum algorithms.

[2]This not only refers to the number of classical input bits, but, in the case of "black-box" algorithms, also to the number of qubit-parameters for any oracle-operators.

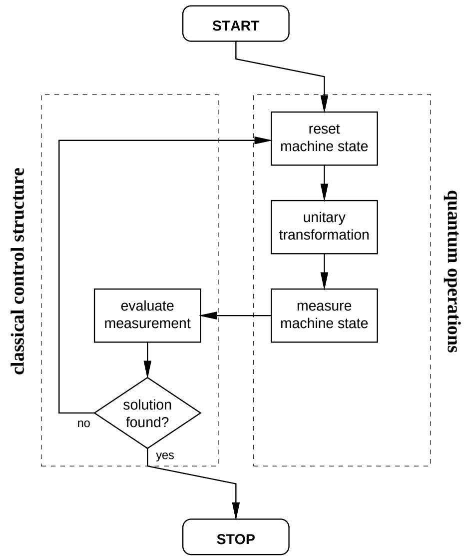


Figure 2.1: *A simple probabilistic quantum algorithm*

For a quantum computer

$$\mathcal{M} = \left(\mathcal{B}^{\otimes 2}, \{H(\mathbf{s}), G(\mathbf{s}), \texttt{reset}\}, \{\mu(\mathbf{q})\}, |0\rangle, \mu[N]\right), \tag{2.2}$$

Deutsch's algorithm (see 1.5.2.2) can e.g. be implemented by the following finite program:

```
1:  reset then 2
2:  H(q0) then 3
3:  G(q0,q1) then 4
4:  H(q0) then 5
5:  H(q1) then 6
6:  if μ(q1) then 7 else 1
7:  μ(q0) then 0
```

Generally, for fixed problem sizes, any probabilistic quantum algorithm can be implemented as a finite program for a quantum computer with a universal set of gates.[3]

[3]This is possible as for fixed problem sizes, there always exists an upper bound for the number of scratch qubits required to implement the classical control structure.

2.2.2.1 Unlimited Memory Machines

In classical computing, a machine $\mathcal{M}$ in universal, if for any partial recursive function $f \in P$, there exists a finite π which implements f on $\mathcal{M}$ (see 1.3.3.2). Since there exists no upper bound in input size, any classical universal machine must have unlimited memory.

Definition 28 *Let $O \subset SU(2)$ be an enumerable set of single qubit gates which is dense in $SU(2)$. A quantum computer*

$$\mathrm{UGM} = (\mathcal{B}^\star, \{U(\mathbf{q}) \,|\, U \in O\} \cup \{\mathtt{CNot}(\mathbf{q}, \mathbf{p})\}, \{\mu(\mathbf{q})\}, n \to |n\rangle, \mu[N]) \quad (2.3)$$

is called unlimited gate machine.

Since $SU(2)$ and `CNot` are a universal set of gates (see 1.4.2.6), any n-qubit unitary operator can be implemented as a sequence for a UGM. However, the UGM is *not* a universal computer in the classical sense. While this might seem surprising at first glance, the non-universality of the UGM (and any other quantum computer operating by means of finite gates) becomes obvious if we consider the following simple decision problem:

The parity of a bit-string $s \in \mathbf{B}^\star$ is *odd* if $\bigoplus_i s_i = 1$ and *even* if $\bigoplus_i s_i = 0$. If the length of s is known to be at most n bits, then the parity of s can be computed by an n-qubit quantum circuit using $n-1$ `CNot`-gates. Fig. 2.2 shows the circuit for $n = 4$.

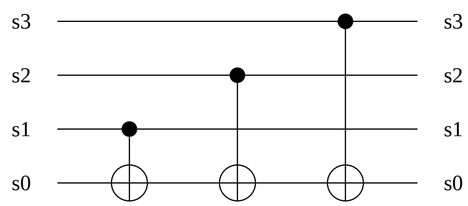


Figure 2.2: *Parity of a bit string of length 4*

Assume that π is a finite quantum program for UGM to compute the parity of arbitrary bit-strings. If π contains $n = |\pi|$ instructions, then — no matter how complex the computation turns out to be for any given input — the transfer function τ_π can only result in a composition of at most n different gates or measurements. Since any memory or test command affects at most 2 qubits, τ_π can only affect a $2n$-qubit register of UGM. Therefore only $2n$ qubits can contribute to the result and there exist at least 2^{2n} pairs of bit-strings $a, b \in \mathbf{B}^{2n+1}$ which differ in a single bit and therefore have opposite parities, for which $F(\pi, \mathrm{UGM})(a) = F(\pi, \mathrm{UGM})(b)$.

2.2.2.2 Universal Quantum Computers

The above results shows that for finite programs to be universal, a quantum computer $\mathcal{M} = (\mathcal{H}, O, T, \delta, \beta)$ has to provide at least one *global* operator $U \in O$ which acts non-trivially on $\mathbf{l}_2$ subspaces of $\mathcal{H}$.[4]

Let $\mathcal{H} = \mathcal{T} = \mathcal{B}^\star \otimes \mathcal{B}^\star$ be the Hilbert space of a double infinite "tape" of qubits with zero tail states i.e.

$$\mathcal{T} = \left\{ |0\rangle^{\otimes\omega} \otimes |\psi\rangle \otimes |0\rangle^{\otimes\omega} \,\Big|\, |\psi\rangle \in \mathcal{B}^{\otimes n}, n \in \mathbf{N} \right\} \tag{2.4}$$

and B be the computational basis of $\mathcal{T}$ with the single qubit registers $\mathbf{l}, \mathbf{h}$ and $\mathbf{r}$

$$B = \left\{ |\ldots s_{-3}s_{-2}\rangle |s_{-1}\rangle_{\mathbf{l}} |s_0\rangle_{\mathbf{h}} |s_1\rangle_{\mathbf{r}} |s_2 s_3 \ldots\rangle \,\Big|\, s : \mathbf{Z} \to \mathbf{B}, \sum_{i=-\infty}^{\infty} s_i < \infty \right\} \tag{2.5}$$

We can now define a unitary shift operator S on $\mathcal{T}$ as

$$S : |\ldots s_{-2}s_{-1}\rangle |s_0\rangle_{\mathbf{h}} |s_1 s_2 s_3 \ldots\rangle \to |\ldots s_{-2}s_{-1}s_0\rangle |s_1\rangle_{\mathbf{h}} |s_2 s_3 \ldots\rangle \tag{2.6}$$

Classical Quantum Turing Machine Let's consider the following quantum computer

$$\mathcal{M} = \left(\mathcal{T}, \{X(\mathbf{h}), S, S^\dagger\}, \{\mu(\mathbf{h})\}, s \to |0\rangle|s\rangle, \mu[N] \right). \tag{2.7}$$

It is obvious that $\mathcal{M}$ does not provide a complete set of unitary operators as each of the three memory commands operates within B, so no superposition can be created. However, $\mathcal{M}$ is a universal computer; in fact $\mathcal{M}$ is a Turing Machine (see 1.3.2.2) in disguise. Table 2.2 gives the homomorphisms to translate finite programs between $\mathcal{M}$ and a TM.

Because of this equivalence, we call $\mathcal{M}$ a *classical quantum Turing machine* or CQTM.

Semi-classical Quantum Turing Machine The CQTM can be extended by adding a universal set of unitary transformations. Because of the availability of the shift operator, those operations can be localized and may be restricted to a small number of fixed qubit positions.[5]

[4] An example would be the step-operator of a QTM (see 1.5.1.2).

[5] This is an important feature and several proposed ion-trap based hardware architectures for quantum computers take advantage of this [37, 53].

TM	**CQTM**
L *(left)*	S
R *(right)*	$S^\dagger$
T *(test)*	$\mu(\mathbf{h})$
S *(set)*	`1:` if $\mu(\mathbf{h})$ then 0 else 2 `2:` $X(\mathbf{h})$ then 0
E *(erase)*	`1:` if $\mu(\mathbf{h})$ then 2 else 0 `2:` $X(\mathbf{h})$ then 0
`1:` if T then 2 else 3 `2:` E then 0 `3:` S then 0	$X(\mathbf{h})$

Table 2.2: *Equivalent TM and CQTM commands*

Definition 29 *The quantum computer*

$$\mathrm{SQTM} = (\mathcal{T}, O, \{\mu(\mathbf{h})\}, s \to |0\rangle|s\rangle, \mu[N]) \tag{2.8}$$

with the memory commands

$$O = \{H(\mathbf{h}), T(\mathbf{h}), \mathtt{CNot}(\mathbf{h}, \mathbf{l}), \mathtt{CNot}(\mathbf{h}, \mathbf{r}), S, S^\dagger\} \tag{2.9}$$

is called semi-classical (or simple) quantum Turing machine.

Since $X = HT^4H$, the SQTM is capable of emulating a CQTM and is also a universal computer.

Let $\mathbf{q}_k$ denote the k^{th} qubit relative to $\mathbf{h}$ and let U be an arbitrary single qubit operator $U \in SU(2)$. Since H and T are universal for single qubit operations (see 1.4.2.6), there exists a composition $U' = H \prod_i T^{n_i} H$ to approximate U to arbitrary precision. [6] The register operator $U(\mathbf{q}_k)$ can then be realized by

$$U(\mathbf{q}_k) \approx (S^\dagger)^k H(\mathbf{h}) \left(\prod_i T^{n_i}(\mathbf{h}) H(\mathbf{h}) \right) S^k. \tag{2.10}$$

Moreover, since

$$\mathtt{Swap}(\mathbf{q}_k, \mathbf{q}_{k+1}) = (S^\dagger)^k \mathtt{CNot}(\mathbf{h}, \mathbf{r}) \mathtt{CNot}(\mathbf{r}, \mathbf{h}) \mathtt{CNot}(\mathbf{h}, \mathbf{r}) S^k \quad \text{and} \tag{2.11}$$

$$\mathtt{Swap}(\mathbf{q}_k, \mathbf{q}_{k+l+1}) = \mathtt{Swap}(\mathbf{q}_k, \mathbf{q}_{k+l}) \mathtt{Swap}(\mathbf{q}_k + l, \mathbf{q}_{k+l+1}) \mathtt{Swap}(\mathbf{q}_k, \mathbf{q}_{k+l}) \tag{2.12}$$

any permutation of qubits can be realized and the controlled-not can be applied to arbitrary registers. This means that a SQTM can emulate a UGM and allows to implement arbitrary unitary transformations.

[6] Note, that H is self-inverse, so the factorization of U' does not contain higher powers of H as $H^{k+2} = H^k$. By the same reasoning $n_i < 8$ as $T^{k+8} = T^k$.

General Quantum Algorithms Non-classical algorithms usually require the creation of quantum circuits depending on classical parameters such as the problem-size. In the most general case, this description is given by a recursive function (see 1.3.1.1), so the actual gate-sequence can be determined by a finite-program on a universal computer.

To demonstrate that the SQTM is able to run general quantum algorithms, we need to show how tape-qubits can be addressed as the result of a classical computation. A possible way to achieve this, is to treat all *even* tape-qubits as classical bit and use them to emulate a TM while using all *odd* qubits to carry out the actual quantum computation (*memory interleaving*).

The example below shows how to apply a Hadamard gate to the k^{th} odd qubit ($\mathbf{q}_{2k+1}$), whose position k is unary encoded (see 1.3.2.2) into the even qubits which are used classically.

1:	`if` $\mu(\mathbf{h})$ `then 2 else 4`	*end mark (0) reached?*
2:	S `then 3`	*move to the next classical bit*
3:	S `then 1`	*and reiterate*
4:	S `then 5`	*move head over qubit*
5:	$H(\mathbf{h})$ `then 6`	*apply Hadamard gate*
6:	$S^\dagger$ `then 7`	*move back to end mark*
7:	$S^\dagger$ `then 8`	*move to the previous*
8:	$S^\dagger$ `then 9`	*classical bit*
9:	`if` $\mu(\mathbf{h})$ `then 7 else 10`	*begin mark (0) reached?*
10:	S `then 11`	*move head to original*
11:	S `then 0`	*position*

2.2.3 Hybrid Architecture

In 2.2.2.1 we showed that gate-based quantum computers are not universal. On the other hand, global unitary operations like the shift operator (see 2.2.2.2) cannot be expressed within the circuit model, cannot be equally applied to machines with unlimited and limited memory and cannot be assumed to be equally available on different quantum hardware architectures.

To overcome the above restrictions, quantum programming uses a classical universal language to define the actual sequence of elementary instructions for a quantum computer, so a program is not intended to run on a quantum computer itself, but on a (probabilistic) classical computer, which in turn controls a quantum computer and processes the results of measurements.

From the perspective of the user, a quantum program behaves exactly like any other classical program, in the sense that it takes classical input, such as startup parameters or interactive data, and produces classical output.

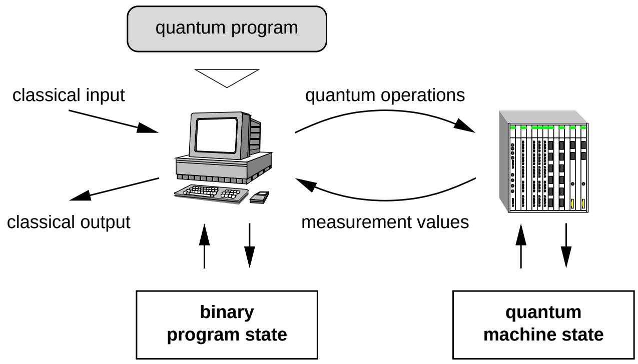


Figure 2.3: *Hybrid quantum architecture*

The state of the controlling computer (i.e. variable values, execution stack, but also the mapping of quantum registers) is referred to as *program state*. The quantum computer itself does not require any control logic, its computational state can therefore be fully described by the common quantum state $|\Psi\rangle$ of its qubits (*machine state*).

2.2.3.1 Machine Model

Formally, the above architecture can be described as a universal computer with a quantum oracle (see 1.3.2.4):

Let $\mathcal{M}_c = (\mathbf{S}, O_c, T_c, \delta, \beta)$ be a universal classical computer with an encoding $\beta : \mathbf{S} \to \mathbf{N}^\star$ and $\mathcal{M}_q = (\mathcal{H}, O_q, T_q, |0\rangle, \mu[N])$ a quantum computer with unlimited memory $\mathcal{H} = \mathcal{B}^\star$, a universal set of finite gates

$$O_q = \bigcup_k \Gamma(G_k) = \bigcup_k \{G_k(\mathbf{s})\} = \{U_0, U_1, \ldots\} \tag{2.13}$$

and single qubit-measurements

$$T_q = \{\mu(\mathbf{q}_i)\} = \{\mu_0, \mu_1, \ldots\}. \tag{2.14}$$

As the number of qubits in $\mathcal{M}_q$ is unlimited, O_c and T_c are infinite sets. Since any $U_i \in O_c$ and $\mu_i \in T_c$ only operates on a finite number of qubits, we need a method to allow $\mathcal{M}_c$ to address arbitrary commands of $\mathcal{M}_q$. Using the encoding

$$\beta_0(s) = \begin{cases} y_0 & \text{if } \beta(s) = (y_0, \ldots) \\ 0 & \text{if } \beta(s) = (\,) \end{cases} \qquad \text{with} \quad s \in \mathbf{S} \tag{2.15}$$

we define the oracle commands

$$U \quad : \quad (s, |\Psi\rangle) \to (s, U_{\beta_0(s)}|\Psi\rangle) \qquad \text{and} \tag{2.16}$$

$$M \quad : \quad (s, |\Psi\rangle) \to \mu_{\beta_0(s)}|\Psi\rangle \tag{2.17}$$

With U and M as interface, we can now define the hybrid machine $\mathcal{M} = \mathcal{M}_c \bowtie \mathcal{M}_q$ as

$$\mathcal{M} = (\mathbf{S} \times \mathcal{H}, O_c \times \{I\} \cup \{U\}, O_c \times \{I\} \cup \{M\}, \delta, \beta) \tag{2.18}$$

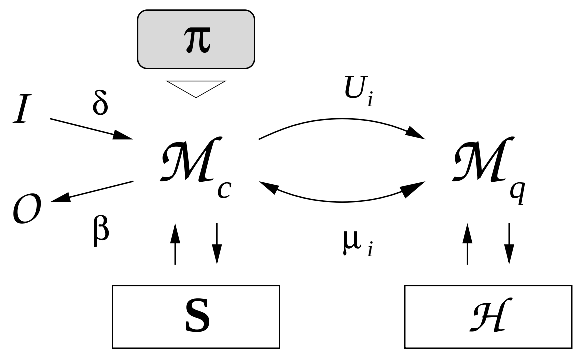


Figure 2.4: *Classical computer with quantum oracle*

2.2.3.2 The Quantum Oracle

For our formal machine model, we assumed $\mathcal{M}_q$ to have unlimited memory. However, the restriction to finite gates and single qubit measurements also allows for quantum oracles with a finite number of qubits.

The concept of quantum programming is intended to be hardware independent and applicable to any qubit-based quantum architecture, so we will not assume a specific set of elementary gates, as long as the following requirements are met:

- The set of gates is universal.
- Each gate can be equally applied to arbitrary qubits.
- For any non self-inverse gate U, the gate $U^\dagger$ is also available.

Elementary gates may take any number of classical parameters and may or may not be restricted in the number of qubits they operate on.

Some concepts of structured quantum programming will require certain operators to be available either as elementary gates or as user-defined operators i.e. procedural compositions of gates which can be defined in the programming language itself. Those operators are

- a `Fanout` operation i.e. an arbitrary unitary operator which matches the condition

$$\texttt{Fanout} : |n\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} \rightarrow |n\rangle_{\mathbf{x}}|n\rangle_{\mathbf{y}} \quad \text{where} \quad |\mathbf{x}| = |\mathbf{y}| \tag{2.19}$$

  `Fanout` is usually implemented as $|a, b\rangle \rightarrow |a, a \oplus b\rangle$ using `CNot`-gates and is required for the transparent use of local scratch registers (see 2.5.3.5).

- the `Not`- (or X-) gate and the controlled-not gate $C^n[X]$ with an arbitrary number of control qubits. The implementation of $C^n[X]$ may use scratch qubits. X and $C^n[X]$ are required for quantum conditions (see 2.6.3) and the quantum if-statement (see 2.6.2.1).

2.3 Structured Programming

Since the computational model of QPLs is that of a classical computer with a quantum oracle, any QPL is also a universal classical programming language. In this section, we will summarize the key elements of classical structured programming languages.

2.3.1 Program Structure

A structured program is a sequence *(block)* of *statements* and *definitions*, which are processed top-down and may contain blocks themselves (control-statements, subroutine-definitions).

2.3.1.1 Statements

Statements range from simple commands, through subroutine calls to nested control-statements and are executed when they are encountered.

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

2.3.1.2 Definitions

Definitions are not executed but bind a value *(variable- or constant-definition)* or a block of code *(subroutine-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);} }
```

Each symbol has an associated type, which can either be a *data type* or a *subroutine type*.

2.3.1.3 Expressions

Statements and subroutine-calls can take arguments of certain data types. Expressions can be composed of literals, variables and constants combined by operators and function calls.

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

2.3.2 Expressions and Variables

Unlike untyped formalisms like finite programs or circuits, in programming languages, an expression is usually associated with a *data type* T. A value $v \in T$ is called an *instance* of T.

2.3.2.1 Atomic Expressions

The direct symbolic representation of an instance $v \in T$ is called a *literal*.

Type	Description	Examples
`int`	integer	`1234, -1`
`real`	real number	`3.14, -0.001`
`complex`	complex number	`(0,-1), (0.5, 0.866)`

Table 2.3: *Scalar arithmetic types and literals in QCL*

A symbol which is permanently bound to a value $v \in T$ is called a *(symbolic) constant* of type T. A symbol which can be bound to arbitrary instances of T is called a *variable* of type T.

2.3.2.2 Definitions

Constants and variables have to be defined (i.e. declared and initialized) before they can be used. To assure deterministic behaviour (see 2.5.1.2) we require that a variable is bound to a type dependent default value if no initial value is provided.

```
qcl> const pi = 3.141592653589793238462643383279502884197;
qcl> const I  = (0,1);
qcl> complex z=exp(I*pi/4);
qcl> string msg="Hello World";
qcl> real vector v[3];    // v is initialized with [0,0,0]
```

After its definition, a constant or variable is visible and can be used in any subsequent statements and definitions until the end of the current subroutine *(local symbol)* or the end of the program *(global symbol)*.[7]

2.3.2.3 Composite Expressions

Literals, constants and variables are *atomic* expressions. General expressions can be recursively constructed from atomic expression by *operators* and *function calls*.

$$\begin{array}{rcl} \mathit{expr} & \leftarrow & \mathit{atomic\text{-}expression} \\ & \leftarrow & \mathit{function}(\,\mathit{expr}, \ldots) \\ & \leftarrow & \mathit{unary\text{-}operator}\ \mathit{expr} \\ & \leftarrow & \mathit{expr}\ \mathit{binary\text{-}operator}\ \mathit{expr} \\ & \leftarrow & \mathit{n\text{-}ary\text{-}operator}(\,\mathit{expr}, \ldots) \end{array}$$

So general expressions can be described as trees with literals, constants and variables as leaf-nodes and operators and functions as inner nodes.

```
qcl> print (3^2+4^2)*sin(log(z)/I);
: 17.6777
```

2.3.3 Subroutines

A subroutine S is a named block (also referred to as *body* of S) preceded by a list of *parameter-declarations*. The body of a subroutine may contain calls to itself *(recursion)*.

Classical procedural languages usually provide two types of subroutines:

[7] There also exist languages with finer grained scoping rules, e.g. C++, where variables can be local to a block.

2.3.3.1 Procedures

A procedure is a general subroutine with arbitrary dependencies and side-effects on the program state. This means that besides its declared arguments, a procedure can also depend on hidden parameters, external input and random events. Moreover, procedures can also change the binding of global variables.

```
int cash;
procedure roulette(int bet) {
  int n;
  input "pick a number:",n;
  cash=cash-bet;
  if n==floor(37*random()) { cash=cash+36*bet; };
}
```

In procedural quantum programming, procedures, being the most general subroutine type (see 2.5), are used to implement the classical control structure of quantum algorithms (see 2.2.1.2).

2.3.3.2 Functions

A function is a subroutine which returns a value $v \in T$ of a certain data-type T.

In procedural quantum programming, functions have strict mathematical semantics, which means that v must exclusively and deterministically depend on the declared arguments and that the computation of v must not exhibit any side-effects on the program state.[8] This implies that neither global variables nor calls of less restricted subroutine types may appear within the body of a function.

```
int fibonacci(int n) {
  if n<2 {
    return 1;
  } else {
    return fibonacci(n-1)+fibonacci(n-2);
  }
}
```

2.3.4 Statements

2.3.4.1 Assignment

An assignment binds a variable (or an element of a data structure such as an array) of type T to a value specified by an expression of the same type.

[8] Many classical procedural languages (e.g. C) are less strict and treat functions as procedures with a return value.

```
qcl> z=z^2;
qcl> v=vector(cos(pi/6),sin(pi/6),0);
qcl> v[2]=1;
qcl> print z,v;
: (0,1) [0.866025,0.5,1]
```

Assignments and atomic expressions are sufficient to implement the class BF of basic functions (see 1.3.1.1).

2.3.4.2 Control-Statements

According to the principles of structured programming, flow-control is executed in terms of control-statements which execute blocks according to the value of a boolean expression and have well defined entry- and exit-points.

The two basic control-structures are *if-statements* and *(conditional) loops.*[9] Conditional loops differ in whether they evaluate the loop condition before *(while-loop)* or after *(until-loop)* the body. In the latter case, the body is executed at least once.

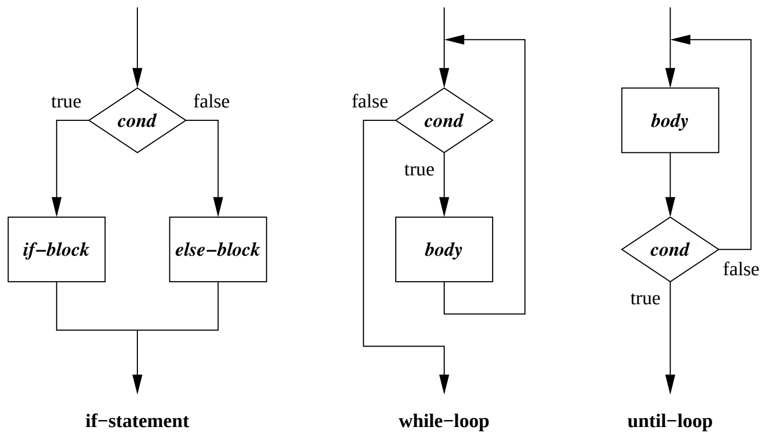


Figure 2.5: *Basic control structures*

As conditional-loops allow the implementation of μ_0-recursion, a programming language with while-loops, assignments, atomic expressions and the operators "+" and "=" is universal (see 1.3.1.1).

Most structured languages also provide a special statement for counting loops over a known range of integers *(for-loops).*

[9]Conceptually, a single block (sequence) can be regarded as a third control-structure [11].

```
qcl> for i=a to b { body(i); };           // for-loop and equivalent
qcl> i=a; while i<=b { body(i); i=i+1; }  // while-loop
```

As counting loops allow the implementation of primitive recursion, a programming language with for-loops, assignments, atomic expressions and addition is sufficient to implement the class PR of primitive recursive functions (see 1.3.1.1).

Break Statements A *break-statement* allows the innermost loop to be immediately exited, regardless of the loop condition. While the possibility to exit a loop from any point within the body can be considered to be against the spirit of structured programming as it compromises the concept of well defined exit points, most structured languages do provide a break-statement (as does QCL).

2.3.4.3 Calls

A *call* of a procedure S binds a list of *arguments* to the parameters of S and executes the body of S. The parameter bindings as well as any symbol definitions within S are local.

2.3.4.4 Input and Output

Any programming language has to provide means to communicate with the outside world. This can be realized by designated input- and output-variables at the start and termination of the program or at runtime via I/O-commands.

```
qcl> input "length in cm:",x;
? length in cm: 192
qcl> print x,"cm =",x/2.54,"inches";
: 192 cm = 75.5906 inches
```

2.4 Elementary Quantum Operations

As introduced in 2.2.3, the computational model of quantum programming is a universal computer with a quantum oracle. Formally, the interface between the classical front-end and the quantum backend can be described by oracle commands which allow elementary gates to be applied to and measurements to be performed on arbitrary target qubits.

In this section, we will show how this interface can be represented within the scope of an imperative (see 2.1.3.1) programming language *(imperative quantum programming)*.[10]

[10]Even if the classical language provides subroutines and structured flow-control, the

2.4.1 Quantum Registers

In 1.4.1.3, we defined a quantum register $\mathbf{s}$ as an arbitrary sub-system with a finite dimensional state space $\mathcal{H}_\mathbf{s}$ and a well-defined computational basis $B_\mathbf{s}$. We will also make the following assumptions about the quantum oracle $\mathcal{M}_q$ (see 2.2.3.2):

- The state space $\mathcal{H}$ of $\mathcal{M}_q$ is a composition of identical qubit subsystems, i.e. $\mathcal{H} = \mathcal{B}^{\otimes n}$ or $\mathcal{H} = \mathcal{B}^\star$.
- Elementary gates and measurements operate on a finite number of qubits.[11]
- All operations of $\mathcal{M}_q$ can be equally applied to arbitrary qubits.
- The qubits of $\mathcal{M}_q$ are numbered in ascending order starting with zero.

The restriction to qubits implies that any register $\mathbf{s}$ is either a single qubit ($\mathbf{s} = \mathbf{q}_i$), a composition of single qubit registers ($\mathbf{s} = \mathbf{q}_{k_0} \circ \mathbf{q}_{k_1} \circ \ldots \circ \mathbf{q}_{k_{n-1}}$) or the *null-register* ($\mathbf{s} = \mathbf{o}$). An n-qubit quantum computer thus allows for

$$\sum_{k=0}^{n} \frac{n!}{(n-k)!} = n! \sum_{k=0}^{n} \frac{1}{k!} = \lfloor \mathrm{e}\, n! \rfloor \tag{2.20}$$

different registers.

Notation We write R_k^n to denote the set of k-qubit registers for an n-qubit quantum computer. As R_k^n is equivalent to the set of possible k-permutations of n elements,

$$|R_n^k| = P_k^n = \frac{n!}{(n-k)!}. \tag{2.21}$$

Further, we define

$$R_k = \bigcup\nolimits_{n=1}^{\infty} R_k^n \qquad \textit{(set of k-qubit registers)} \tag{2.22}$$

$$R^n = \bigcup\nolimits_{k=0}^{n} R_k^n \qquad \textit{(set of registers over n qubits)} \tag{2.23}$$

$$R = \bigcup\nolimits_{n=1}^{\infty} R^n \qquad \textit{(set of general registers)} \tag{2.24}$$

resulting QPL would still be regarded as imperative as it lacks semantic support for operators and quantum if-statements.

[11] For convenience, QCL also supports a (global) reset-command (see 1.4.3.3).

2.4.1.1 Language Representation of Registers

Within a structured QPL, quantum registers are instances of a *quantum data type* (see 2.4.1.3) Q.[12] Semantically, an n-qubit register $\mathbf{s} \in Q$ is a finite sequence of mutually different *qubit positions* $\mathbf{s} = (s_0, s_1 \ldots s_{n-1})$, so $\mathbf{s}$ does not denote the physical register itself, but serves as a pointer *(logical register)* to the actual n-qubit sub-system $\mathbf{q}_{s_0} \circ \ldots \circ \mathbf{q}_{s_{n-1}}$ of $\mathcal{M}_q$ *(physical register)*.

This distinction allows registers to be treated just like any other classical data type, so register variables can be declared, printed and combined to expressions, just like classical variables.[13]

```
qcl> qureg q[1];           // allocate single qubit register
qcl> qureg p[4];           // allocate 4-qubit register
qcl> qureg qp = q & p;     // declare combined register qp
qcl> print q,p,qp;         // print register mappings
: <0> <1,2,3,4> <0,1,2,3,4>
qcl> print p[0..2] & q;    // registers can be combined to expressions
: <1,2,3,0>
```

Registers are the formal interface between the classical front-end and the quantum oracle. All operations on the machine state take quantum registers as operands[14] and are restricted to their corresponding sub-spaces.

```
qcl> H(q);                 // apply Hadamard gate to register q
[5/32] 0.70711 |0,0> + 0.70711 |1,0>
qcl> Not(p);               // invert qubits in register p
[5/32] 0.70711 |0,15> + 0.70711 |1,15>
qcl> measure q;            // measure register q
[5/32] 1 |1,15>
```

2.4.1.2 The Quantum Heap

The mapping between logical registers and physical qubits is handled transparently by allocation and deallocation of registers from the set of all available qubits, also referred to as the *quantum heap* [46].[15] Quantum registers are explicitely allocated, when a register variable is defined, but also implicitly

[12] Since quantum data types are only used to restrict the possible operations on registers, mathematically, any quantum data type Q denotes the same set of qubit permutations, i.e. $Q = R$.

[13] To allow for the static allocation of qubits, QCL does not allow register assignments, so in QCL, registers variables are treated as symbolic constants.

[14] Depending on our point of view, registers can be thought of being passed by value (logical registers) or reference (physical registers).

[15] In classical programming, the section of memory reserved for dynamic variables is called the heap.

for scratch space management (see 2.5.3.5) and for the evaluation of quantum conditions (see 2.6.3).

Just as classical variables are initialized with type-dependent default values, newly allocated registers and consequently all free (i.e. unallocated) quantum memory has to be *empty*.

Definition 30 (Empty Registers) *A quantum register* $\mathbf{e}$ *is empty iff the machine state* $|\Psi\rangle$ *is of the form* $|\Psi\rangle = |0\rangle_{\mathbf{e}}|\psi\rangle_{\bar{\mathbf{e}}}$ *or, equivalently,* $\rho_{\mathbf{e}} = |0\rangle\langle 0|$.

Also, temporary registers must be emptied before deallocation, which, in the case of local register variables, takes place when the symbol leaves scope. This can be achieved either by measurement or by uncomputing (see 2.5.3.5).

At startup, the whole machine state is empty, thus $|\Psi\rangle = |0\rangle$.

```
qcl> dump;
: STATE: 0 / 32 qubits allocated, 32 / 32 qubits free
1 |0>
```

The concept of the quantum heap allows for two important abstractions:

- Since the allocation of registers is transparent, no qubit positions need to be specified and no register literals need to be defined. This also leaves room for architecture-dependent optimizations.
- Since allocated and unallocated qubits are in a product state, the definition of quantum algorithms is independent of the total number of qubits.

2.4.1.3 Quantum Data Types

Different quantum data-types can be used to restrict the way unitary operators may affect quantum registers. This is not only done to prevent programming errors and to make the code more readable, but also to provide information to the compiler or interpreter to allow for more efficient optimizations and to designate the argument-, target- and scratch-registers of quantum functions (see 2.5.3.3), to allow for transparent scratch space management.

General Registers impose no restrictions and can be used as arguments to arbitrary operators.

Type	**Restriction**
`qureg`	none
`quconst`	invariant to all suboperators
`quvoid`	has to be empty when the uninverted operator is called
`quscratch`	has to be empty before and after the call

Table 2.4: *Quantum data types in QCL*

Constant Registers must be invariant to all operators. They are used to designate argument registers of quantum functions (see 2.5.3.3). The enable registers of controlled gates (see 1.4.2.5) and conditional operators (see 2.6.1) are also of this data type.

Definition 31 (Invariance of Registers) *A quantum register* $\mathbf{c}$ *is invariant to a register operator* $U(\mathbf{s}, \mathbf{c})$ *iff*

$$U\,|i\rangle_{\mathbf{s}}|j\rangle_{\mathbf{c}} = (U_j\,|i\rangle_{\mathbf{s}})\ |j\rangle_{\mathbf{c}} \tag{2.25}$$

where U_j *are arbitrary* $|\mathbf{s}|$*-qubit unitary operators, so* U *can be written as*

$$U = \begin{pmatrix} U_0 & 0 & \cdots & 0 \\ 0 & U_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & U_{n-1} \end{pmatrix} \quad \text{with} \quad n = 2^{|\mathbf{c}|}. \tag{2.26}$$

An operator U of the above form is also called *selection operator* (see 2.5.2.2).

Target Registers are used to designate the result register $\mathbf{t}$ for quantum functions (see 2.5.3.3) of the form

$$F : |n\rangle_{\mathbf{c}}|0\rangle_{\mathbf{t}} \to |n\rangle_{\mathbf{c}}|f(n)\rangle_{\mathbf{t}}. \tag{2.27}$$

Since $F\,|x, y\rangle$ is undefined[16] for $y \neq 0$, target registers need to be empty when F is called or else the effect of the operator is undefined.

[16] This is done in order to allow for different ways to accumulate the result, so $F : |x, y\rangle \to |x, y \oplus f(x)\rangle$ and $F : |x, y\rangle \to |x, (y + f(x)) \bmod 2^n\rangle$ are merely considered to be different implementations of the same quantum function.

Scratch Registers are temporary registers which can be used within the implementation of an operator but are required to be empty before and after the call.[17]

2.4.1.4 Register Expressions

On an n-qubit quantum computer, a single k-qubit gate U can be used to implement up to $\frac{n!}{(n-k)!}$ different unitary transformation $U(\mathbf{s})$ by applying U to different registers. In order to realize all possible qubit permutations, a QPL has to provide means to extract subregisters and to combine disjoint registers.

Definition 32 *A register* $\mathbf{a} = (a_0 \dots a_{n-1}) \in R_n$ *is a subregister* $\mathbf{a} \subseteq \mathbf{b}$ *of* $\mathbf{b} = (b_0 \dots b_{m-1}) \in R_m$ *iff*

$$\bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{m-1} a_i = b_j. \tag{2.28}$$

Definition 33 *Two registers* $\mathbf{a} = (a_0 \dots a_{n-1}) \in R_n$ *and* $\mathbf{b} = (b_0 \dots b_{m-1}) \in R_m$ *are disjoint iff*

$$\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{m-1} a_i \neq b_j. \tag{2.29}$$

Table 2.5 shows the available operators in QCL to manipulate quantum registers.

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

Table 2.5: *Register expressions in QCL*

2.4.2 Elementary Gates

Just as assignments (see 2.3.4.1) represent atomic changes of the program state, elementary gates are the fundamental primitives to manipulate the quantum machine state.

[17] In QCL, the type `quscratch`, when used for a local register, denotes *managed* scratch registers (see 2.5.3.5) which are uncomputed automatically, while a local `qureg` is an unmanaged scratch (see 2.5.2.4) register which has to be emptied by the operator itself.

2.4.2.1 Register Operators

A k-qubit (elementary) gate (see 1.4.2.4) is an atomic unitary transformation U which can be applied to arbitrary k-qubit registers $\mathbf{s} \in R_k$ of the quantum oracle (see 2.2.3.2). The resulting transformation of the machine state is formally described by the *register operator* (see 1.4.1.3)

$$U(\mathbf{s}) = U \otimes I_{\bar{\mathbf{s}}} = \sum_{i,j,k} u_{ij} \, |i\rangle_{\mathbf{s}} |k\rangle_{\bar{\mathbf{s}}} \langle j|_{\mathbf{s}} \langle k|_{\bar{\mathbf{s}}} \quad \text{with} \quad u_{ij} = \langle i|U|j\rangle. \tag{2.30}$$

In the circuit model, a register operator describes a gate which is wired to operate on certain qubits. In (2.30), the "wiring" is hidden in the basis decomposition $|\ldots\rangle_{\mathbf{s}} \otimes |\ldots\rangle_{\bar{\mathbf{s}}}$, however it can be made explicit, by the introduction of a register-dependent *reordering operator*.

Definition 34 (Reordering Operator) *Let $\mathcal{H} = \mathcal{B}^{\otimes n}$ (or $\mathcal{H} = \mathcal{B}^\star$) and $\mathbf{s} = (s_0 \ldots s_{k-1}) \in R_k^n$ (or R_k) be a k-qubit register of $\mathcal{H}$. A qubit permutation*

$$\Pi_{\mathbf{s}} \, |d_0, d_1 \ldots d_{n-1}\rangle = |d_{\pi_0}, d_{\pi_1} \ldots d_{\pi_{n-1}}\rangle \tag{2.31}$$

on $\mathcal{H}$ with π being a bijective function on $\mathbf{Z}_n$ (or $\mathbf{N}$) and $\pi_i = s_i$ for $i < k$ is called reordering operator for $\mathbf{s}$.

The definition of $\Pi_{\mathbf{s}}$ is not unique; for $\mathcal{H} = \mathcal{B}^{\otimes n}$, there exist $(n-k)!$ different $\Pi_{\mathbf{s}}^{(i)}$ and infinitely many for $\mathcal{H} = \mathcal{B}^\star$.

Functionally, the reordering operator can be compared to the shift- and swap-operations used to address arbitrary tape-qubits on the *SQTM* (see 2.2.2.2).

Definition 35 (Register Operator) *The register operator $U(\mathbf{s})$ for a k-qubit operator U and a k-qubit quantum register $\mathbf{s} \in R_k^n$ of $\mathcal{H} = \mathcal{B}^{\otimes n}$ (or $\mathbf{s} \in R_k$ of $\mathcal{H} = \mathcal{B}^\star$) is defined as*

$$U(\mathbf{s}) = \Pi_{\mathbf{s}}^\dagger \, (U \otimes I) \, \Pi_s \quad \text{on} \quad \mathcal{H}. \tag{2.32}$$

Fig. 2.6 shows a quantum circuit for $U(\mathbf{s})$ on a 5-qubit machine with U being a 3-qubit gate and $\mathbf{s} = (0, 2, 3)$.

2.4.2.2 Language Representation of Gates

Since quantum programming is supposed to be a hardware independent formalism, besides the three basic requirements named in 2.2.3.2 (universality, functional equivalence of qubits, availability of adjoint gates), no assumptions can be made on the operators provided by the quantum oracle and there is no specific set of elementary gates.

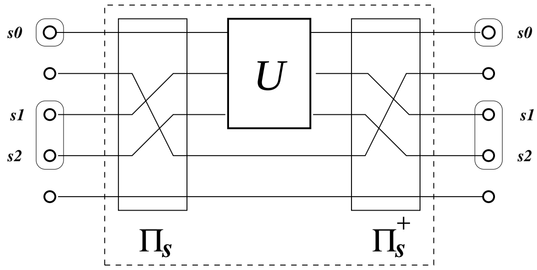


Figure 2.6: *Register operator* $U(\mathbf{s})$

Consequently, any QPL has to allow the declaration of arbitrary elementary gates and the transparent replacement of gates by user-defined operators i.e. non-classical subroutines (see 2.5). Therefore, within a procedural QPL (see 2.5), elementary gates are treated as *external subroutines* with at least one parameter of a quantum data-type to indicate the register to operate upon.

```
extern operator H(qureg q);                // Hadamard gate
extern operator RotX(real theta,qureg q);  // X-Rotation
extern qufunct CNOT(qureg q,quconst c);    // controlled-not
```

The calling syntax of gates is identical to procedures, except that calls can be inverted as for any gate U there also exists the gate $U^\dagger$. In QCL, this is done by preceding the name of the gate the with the adjungation prefix "`!`".

```
qcl> qureg a[1]; qureg b[1];          // allocate 2 qubits
qcl> H(a);                            // apply H to 1st qubit
[2/32] 0.7071 |0,0> + 0.7071 |1,0>
qcl> CNOT(b,a);                       // controlled-not
[2/32] 0.7071 |0,0> + 0.7071 |1,1>
qcl> RotX(pi/3,b);                    // rotate 2nd qubit by pi/3
[2/32] 0.6123 |0,0> - 0.3535i |1,0> - 0.3535i |0,1> + 0.6137 |1,1>
qcl> !RotX(pi/3,b);                   // undo last operation
[2/32] 0.7071 |0,0> + 0.7071 |1,1>    // equiv. to RotX(-pi/3,b)
```

2.4.2.3 Operator Types

Besides quantum data types, procedural QPLs also provide different operator types which can be used for gates as well as non-classical subroutines (see 2.5). Generally, for any given gate U, the most restrictive operator type and the most restrictive register types which match the definition of U should be used, as restricted gates can be called by less restricted subroutines but not the other way around (see 2.5.1.1).

General Operators allow arbitrary unitary transformations on the argument registers.

```
extern operator RotX(real theta,qureg q);  // X-Rotation
```

Basis Permutations are operators of the form $F : |n\rangle \to |f(n)\rangle$ with f being a bijective boolean function.

```
extern qufunct CNOT(qureg q,quconst c);    // controlled-not
```

Conditional Operators are a generalization of controlled gates (see 1.4.2.5) for enable registers of arbitrary size (including the null-register i.e. the uncontrolled operator).

Definition 36 *Let* $\mathbf{s}, \mathbf{e} \in R$ *be disjoint registers and* $U(\mathbf{s})$ *be a register operator, then the operator*

$$U_{[[\mathbf{e}]]}(\mathbf{s}) : |k\rangle_{\mathbf{s}}|c\rangle_{\mathbf{e}} \to \begin{cases} (U\,|k\rangle_{\mathbf{s}})|c\rangle_{\mathbf{e}} & \textit{if } c = 111\ldots \\ |k\rangle_{\mathbf{s}}|c\rangle_{\mathbf{e}} & \textit{otherwise} \end{cases} \tag{2.33}$$

is called conditional operator with the enable (or control) register $\mathbf{e}$

General operators as well as basis permutations can be declared conditional.

```
extern cond operator Phase(real phi);      // conditional phase gate
extern cond qufunct Not(qureg q);          // conditional not gate
```

The enable register is passed as an implicit parameter if the operator is used within the body of a *quantum if-statement* (see 2.6.2.1), so operators without an explicit argument register can be useful (conditional phase gates).

```
qcl> qureg s[1]; qureg e[2];   // allocate 2 registers
qcl> H(e);                     // create test state in e
[3/32] 0.5 |0,0> + 0.5 |0,1> + 0.5 |0,2> + 0.5 |0,3>
qcl> if e[0] { Phase(pi); }    // flip sign if LSB of e is set
[3/32] 0.5 |0,0> - 0.5 |0,1> + 0.5 |0,2> - 0.5 |0,3>
qcl> if e { Not(s); }          // invert target qubit s for e=11
[3/32] 0.5 |0,0> - 0.5 |0,1> + 0.5 |0,2> - 0.5 |1,3>
```

2.4.3 Measurements

2.4.3.1 Register Observable

In quantum mechanics, classical physical quantities *(observables)* are described by Hermitian operators (see 1.2.3.3). In quantum programming, we restrict measurements to the computational basis $B = \{|n\rangle\}$, expressed by the *standard observable* $N = \sum_n n|n\rangle\langle n|$. Using the register basis $B_{\mathbf{s}} = \{|k\rangle_{\mathbf{s}}\}$ the register observable $N(\mathbf{s})$ can be defined as (see 1.4.1.3)

$$N(\mathbf{s}) = N_{\mathbf{s}} \otimes I_{\bar{\mathbf{s}}} = \sum_{i,j} i\,|i\rangle_{\mathbf{s}}|j\rangle_{\bar{\mathbf{s}}}\langle i|_{\mathbf{s}}\langle j|_{\bar{\mathbf{s}}}. \tag{2.34}$$

Again, we can avoid the use of the register basis $B_{\mathbf{s}}$ by defining $N(\mathbf{s})$ by means of a reordering operator $\Pi_{\mathbf{s}}$ (see 2.4.2.1).

Definition 37 (Register Observable) *The register observable $N(\mathbf{s})$ for a k-qubit quantum register $\mathbf{s} \in R_k^n$ on $\mathcal{H} = \mathcal{B}^{\otimes n}$ (or $\mathbf{s} \in R_k$ on $\mathcal{H} = \mathcal{B}^{\star}$) is defined as*

$$N(\mathbf{s}) = \Pi_{\mathbf{s}}^{\dagger}\,(N \otimes I)\,\Pi_s \quad \text{with} \quad N = \sum_{i=0}^{2^k-1} i|i\rangle\langle i|. \tag{2.35}$$

Further, we assume the basis-vectors are labeled as *little-endian* binary numbers (i.e. LSB first), so for two disjoint registers $\mathbf{a} \in R_n$ and $\mathbf{b} \in R_m$

$$N(\mathbf{a} \circ \mathbf{b}) = N(\mathbf{a}) + 2^n N(\mathbf{b}). \tag{2.36}$$

2.4.3.2 Language Representation of Measurements

Since QPLs are intended to control a classical computer which serves as a front-end between the user and the quantum oracle (see 2.2.3), quantum measurements are treated analogously to external classical input (see 2.3.4.4) and are implemented as a quantum input command with the side effect that the machine state gets reduced (see 1.2.3.3).

```
qcl> qureg q[8];          // allocate an 8-qubit register q
qcl> int m;               // declare a classical input variable
qcl> H(q);                // prepare an even superposition in q
[8/32] 0.0625 |0> + ... + 0.0625 |255> (256 terms)
qcl> measure q[0..5],m;   // measure the first 6 qubits of q
[8/32] 0.5 |50> + 0.5 |114> + 0.5 |178> + 0.5 |242>
qcl> print m;             // print measurement result
: 50
```

2.4.3.3 Initialization

Since many quantum algorithms are probabilistic (see 2.2.1.2) and involve iterating over the same computation until a solution is found, it is convenient to provide a command to reset the machine state $|\Psi\rangle$ to the initial state $|0\rangle$ and empty all allocated registers, while not affecting the program state, so that variable bindings and register mappings remain valid.

```
[8/32] 0.5 |50> + 0.5 |114> + 0.5 |178> + 0.5 |242>
qcl> reset;              // reset quantum state as generated above
[8/32] 1 |0>
qcl> print q,m;          // variables q and m are not affected
: <0,1,2,3,4,5,6,7> 50
```

Since the initialization of $|\Psi\rangle$ can be implemented by measuring all allocated registers and inverting qubits $\mathbf{q}_i$ found in state $\rho_{\mathbf{q}_i} = |1\rangle\langle 1|$ to zero by applying the not-gate $X(\mathbf{q}_i)$, it is not necessary for the quantum oracle to provide a generic `reset`-command (see 1.4.3.3).

```
procedure resetregister(qureg q) {
  int i;
  int m;
  for i=0 to #q-1 {          // iterate over qubits
    measure q[i],m;          // measure i-th qubit
    if m==1 { Not(q[i]); }   // invert if measured 1
  }
}
```

2.5 Operators

While registers, elementary gates and measurements are already sufficient to implement arbitrary quantum algorithms, a programming language restricted to those concepts would not be much different from a classical language with a quantum device driver as it would lack a semantic representation for unitary operators, which are the essence of all quantum algorithms.

In this section, we will demonstrate how unitary operators can be integrated into the framework of a procedural programming language *(procedural quantum programming)*.

2.5.1 Quantum Subroutines

Procedural programming languages provide arbitrary levels of abstraction by allowing to group simple computational tasks into parameterized subroutines which can be used recursively as primitives for the definition of more complex subroutines.

Within a procedural QPL, unitary operators are represented as quantum subroutines which allow the recursive construction of complex quantum circuits from elementary gates.

2.5.1.1 Hierarchy of Subroutines

In addition to classical procedures and functions (see 2.3.3), we provide two quantum subroutine types (see 2.4.2.3) for general unitary operators and basis permutations.

Both quantum subroutine types (which together we will refer to as *operators*) have mathematical semantics (see 2.5.1.2) and can be inverted to produce the adjoint operator. Quantum functions additionally allow the transparent use of (managed) scratch registers (see 2.5.3.5).

Subroutine	**QCL**	**S**	$\mathcal{H}$	**inv.**	**scratch**
procedure	`procedure`	all	all	no	no
general unitarian	`operator`	none	unitary	yes	no
basis permutation	`qufunct`	none	rev. boolean	yes	yes
functions	*return type*	none	none	no	no

Table 2.6: *Hierarchy of subroutines*

The 4 subroutine types form a call hierarchy, which means that a routine may only invoke subroutines of the same or a lower level. Table 2.6 lists the subroutines together with their QCL type, their allowed classical (**S**) and quantum ($\mathcal{H}$) side effects, their invertibility and support for scratch space management.

Deutsch's Algorithm To illustrate the above concept, the following QCL implementation of Deutsch's algorithm (see 1.5.2.2) uses all 4 subroutine types:

```
/* Define Oracle */

const coin1=(random()>=0.5);       // Define two random boolean
const coin2=(random()>=0.5);       //   constants

boolean g(boolean x) {             // Oracle function g
  if coin1 {                       // coin1=true  -> g is constant
    return coin2;
  } else {                         // coin1=false -> g is balanced
    return x xor coin2;
  }
}
```

```
qufunct G(quconst x,quvoid y) {    // Construct oracle op. G from g
  if g(false) xor g(true) { CNot(y,x); }
  if g(false) { Not(y); }
}

/* Deutsch's Algorithm */

operator U(qureg x,qureg y) {      // Bundle all unitary operations
  H(x);                            //   of the algorithm into one
  G(x,y);                          //   operator U
  H(x & y);
}

procedure deutsch() {              // Classical control structure
  qureg x[1];                      // allocate 2 qubits
  qureg y[1];
  int m;
  {                                // evaluation loop
    reset;                         //   initialize machine state
    U(x,y);                        //   do unitary computation
    measure y,m;                   //   measure 2nd register
  } until m==1;                    // value in 1st register valid?
  measure x,m;                     // measure 1st register which
  print "g(0) xor g(1) =",m;       //   contains g(0) xor g(1)
  reset;                           // clean up
}
```

2.5.1.2 Mathematical Semantics

In classical routines, subroutines are executed when they are called *(linear execution)* i.e. when control reaches the corresponding call-statement (see 2.3.4.3). Operators, however, support non-classical concepts like invertibility (see 2.5.2.3), scratch space management (see 2.5.3.5) and quantum if-statements (see 2.6.2.1) with the result that neither the number nor the execution order of suboperators necessarily corresponds to the classical flow of control *(non-linear execution)*.

Therefore operators have mathematical semantics i.e. their effect is completely described by the unitary transformation they implement as a function of their declared parameters; so they must be reproduceable and neither depend on nor exhibit side-effects on the program state. This specifically excludes

- the use of global variables
- user input and classical random numbers
- measurements and resetting of the machine state
- calls of procedures

2.5.1.3 Language Representation of Operators

Formally, just like a classical procedure, an operator is a named block (see 2.3.1) with a list of symbolic parameters of classical or quantum types. The latter are used to indicate the registers the operator is applied to.

The QCL operator below implements the basis permutation

$$\mathtt{xreg}_b(\mathbf{a}) : |x\rangle_{\mathbf{a}} \to |x \oplus b\rangle_{\mathbf{a}} \tag{2.37}$$

```
qufunct xreg(qureg a,int b) {    // xor reg. a with binary number b
  int i;
  for i=0 to #a-1 {              // iterate over qubits of a
    if bit(b,i) { Not(a[i]); }   // invert i-th qubit of a if
  }                              //   i-th bit of b is set
}
```

The calling syntax of operators is identical to gates, including the ability to call the adjoint operator. Since operators have mathematical semantics, an operator is completely equivalent to an elementary gate with the same declaration and functionality. So it makes e.g. no difference whether the Z-gate (see 1.4.2.4) is provided by the quantum oracle

```
extern operator Z(qureg q);      // pi-rotation about the Z-axis
```

or implemented as the sequence $Z = HXH$ by the operator

```
operator Z(qureg q) { H(q); Not(q); H(q); }
```

2.5.1.4 Polymorphism

On an n-qubit quantum computer, an k-qubit gate can implement up to $\frac{n!}{(n-k)!}$ unitary transformations $U(\mathbf{s})$ with $\mathbf{s} \in R_k^n$. In procedural quantum programming, this *polymorphism* is further extended by additional abstractions:

- **Register Size:** Argument registers can be of different sizes, so for every quantum parameter $\mathbf{s}$, the register size $|\mathbf{s}|$ is passed as an implicit classical parameter. In QCL, the size of a register is given by the *size-operator* "#", so #s $= |\mathbf{s}|$. An operator with a single argument register $\mathbf{s} \in R^n$ can implement up to $\lfloor e\, n! \rfloor$ different unitary transformations.

- **Multiple Registers:** Operators can take multiple argument registers. Since for any $\mathbf{s} \in R_k$, there exist $\binom{k+q-1}{q-1}$ possible decompositions $\mathbf{s} = \mathbf{s_1} \circ \ldots \circ \mathbf{s_q}$ (including null-registers), an operator with p register arguments $\mathbf{s}_i$ with the total size $k = \sum_{i=1}^{q} |\mathbf{s}_i|$ can implement up to $\binom{k+q-1}{q-1}$ different unitary gates.

- **Classical Parameters:** Besides argument registers, operators can also take an arbitrary number of classical parameters. Any classical parameter of type T increases the number of possible unitary operations by the factor $|T|$.

Generally, on an n-qubit quantum computer, an operator with q register arguments and p classical parameters of the datatypes T_i can implement up to

$$1 + \sum_{k=1}^{n} \frac{n!}{(n-k)!} \binom{k+q-1}{q-1} \prod_{i=1}^{p} |T_i| \tag{2.38}$$

different unitary transformations.

2.5.2 General Operators

2.5.2.1 Quantum Circuits

General operators can be regarded as procedural descriptions of quantum circuits depending on the size of their argument registers as well as classical parameters.

Fig. 2.7 shows the 4-qubit quantum circuit which is generated by the following QCL implementation of the discrete Fourier transform

$$DFT : |x\rangle_{\mathbf{q}} \to \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \mathrm{e}^{\frac{2\pi \mathrm{i}}{N} xy} |y\rangle_{\mathbf{q}} \quad \text{with} \quad N = 2^{|\mathbf{q}|} \tag{2.39}$$

using an FFT-like algorithm suggested by Coppersmith [22].

```
operator dft(qureg q) {       // Discrete Fourier transform
  const n=#q;                 // set n to length of input
  int i; int j;
  for i=1 to n {
    for j=1 to i-1 {          // apply conditional phase gates
      V(pi/2^(i-j),q[n-i] & q[n-j]);
    }
    H(q[n-i]);                // Hadamard gate
  }
  flip(q);                    // swap qubit order of the output
}
```

Since an operator may call other suboperators, this descriptions can be nested. The above implementation of the DFT e.g. uses the suboperator

$$\texttt{flip} : |d_0 d_1 \dots d_{n-1}\rangle \to |d_{n-1} \dots d_1 d_0\rangle \tag{2.40}$$

to generate the last two `Swap`-gates in fig. 2.7.

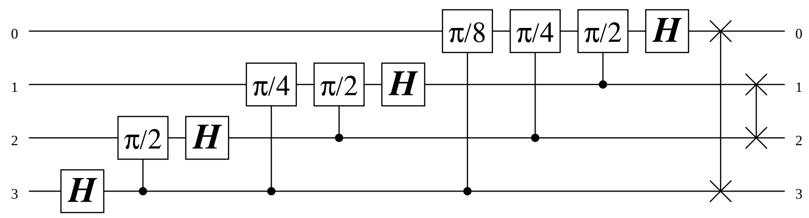


Figure 2.7: *Quantum Fourier Transform for a 4-qubit register*

```
qufunct flip(qureg q) {        // Invert order of qubits
  int i;
  for i=0 to #q/2-1 {
    Swap(q[i],q[#q-i-1]);      // swap 2 opposite qubits
  }
}
```

The implementation of an operator may also contain calls to itself, which results in a recursive definition of the resulting circuit. The QCL operator below is a recursive implementation of the phase transformation

$$P_\phi(\mathbf{s}) : |k\rangle_\mathbf{s} \to \mathrm{e}^{\mathrm{i}\phi k}|k\rangle_\mathbf{s}. \tag{2.41}$$

using the conditional phase gate $V(\phi) = C^n[\mathrm{e}^{\mathrm{i}\phi}]$, which for $n = 1$ is equivalent to $R_z(\phi) = \mathrm{e}^{-\mathrm{i}\phi/2}|0\rangle\langle 0| + \mathrm{e}^{\mathrm{i}\phi/2}|1\rangle\langle 1|$.

```
operator P(qureg q,real phi) {      // Phase transformation
  V(phi,q[0]);                      // rotate LSB
  if #q>1 {                         // if there are higher qubits
    P(q[1..#q-1],2*phi);            // call P with phase 2*phi
  }
}
```

Fig. 2.8 shows how the resulting circuit is constructed recursively for a 4-qubit register $\mathbf{q} = \mathbf{q}_0 \circ \mathbf{q}_1 \circ \mathbf{q}_2 \circ \mathbf{q}_3$.

2.5.2.2 Parameter Types

Besides general registers (QCL type `qureg`) which allow arbitrary quantum circuits, more restrictive quantum data types (see 2.4.1.3) can be used as parameter registers to indicate that an operator U belongs to a certain class of unitary transformations.

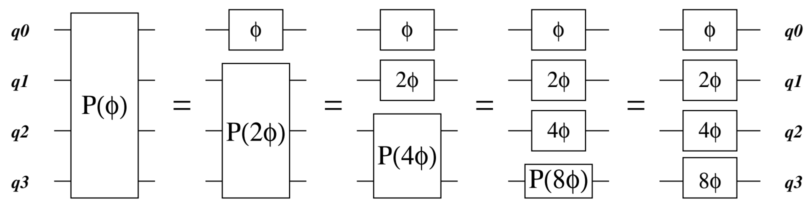


Figure 2.8: *Recursive phase transformation*

Constant Parameters A *constant register* **c** (QCL type `quconst`) indicates that **c** is invariant to the operator, i.e. that U is a *selection operator* of the form

$$U = \sum_k (|k\rangle_{\mathbf{c}}\langle k|_{\mathbf{c}}) \otimes \sum_{ij} u_{kij}|i\rangle\langle j|. \tag{2.42}$$

Definition 38 *An $(n+m)$-qubit operator U of the form*

$$U = \operatorname{diag}(U_0, U_1 \ldots U_{2^m-1}) \tag{2.43}$$

with $U_k \in SU(2^n)$ and $m > 0$ is called selection operator.

To enforce that restriction, within the implementation of U, **c** may only be used as constant argument to any suboperators. So the following implementation of the Z-gate is illegal, despite the fact that

$$Z = HXH = |0\rangle_{\mathbf{c}}\langle 0|_{\mathbf{c}} - |1\rangle_{\mathbf{c}}\langle 1|_{\mathbf{c}} \tag{2.44}$$

is of the form (2.42).

```
qcl> operator z(quconst c) { H(c); X(c); H(c); }
! in operator z(quconst c) at "H(c); ":
! parameter mismatch: quconst used as non-const argument to H
```

In order to allow the declaration of a constant parameter register **c** even when **c** is used as general register argument to suboperators, in QCL, the above restriction can be circumvented by redeclaring **c** as `qureg`. It is then in the responsibility of the programmer to ensure proper `quconst`-semantics (see also 2.5.3.2).

```
operator z(quconst c) {        // Correct implementation of the Z-gate
  qureg q = c;                 // redeclare c as qureg
  H(q);                        // transform into dual basis
  X(q);                        // X-rotation in dual basis
  H(q);                        // transform back into
}                              //   computational basis
```

Target Parameters A *target register* $\mathbf{t}$ (QCL type `quvoid`) is expected to be empty when the un-inverted operator is called, so $U\,|k\rangle_{\mathbf{t}}|\psi\rangle$ is undefined for $k \neq 0$.[18] Therefore, two operators $U^{(1)}$ and $U^{(2)}$ with a target register $\mathbf{t}$ are considered to be equivalent iff

$$U^{(1)}|0\rangle_{\mathbf{t}}|\psi\rangle_{\bar{\mathbf{t}}} = U^{(2)}|0\rangle_{\mathbf{t}}|\psi\rangle_{\bar{\mathbf{t}}} \quad \text{for all} \quad |\psi\rangle \in \mathcal{H}_{\bar{\mathbf{t}}}. \tag{2.45}$$

While target registers are usually used as result registers for quantum functions (see 2.5.3.3), they can also be used for general operators:

```
operator prepare(quvoid t) {        // Prepare test state for empty t
  H(t);                             // produce even superposition
  P(t,2*pi/2^#t);                   // phase transformation
}
```

The QCL operator above expects an empty register $\mathbf{t}$ to prepare a test state of the form

$$|\Psi\rangle = \left(2^{-|\mathbf{t}|/2} \sum_{k=0}^{2^{|\mathbf{t}|}-1} \mathrm{e}^{2^{1-|\mathbf{t}|}\mathrm{i}\pi k}|k\rangle_{\mathbf{t}}\right) \otimes |\psi\rangle_{\bar{\mathbf{t}}} \tag{2.46}$$

using the phase transformation $P(\varphi)$ (2.41) from 2.5.2.1.

```
qcl> qureg q[5];                    // allocate empty 5-qubit register
qcl> prepare(q);                    // prepare test state
[5/32] 0.17678 |0> + ... + (0.17338-0.034487i) |31> (32 terms)
qcl> plot;                          // plot simulated state
```

Fig. 2.9 shows the 5-qubit test state

$$|\Psi\rangle = \sum_{k=0}^{31} c_k|k\rangle \quad \text{with} \quad c_k = \frac{1}{\sqrt{32}}\,\mathrm{e}^{\mathrm{i}\pi k/16} \tag{2.47}$$

as prepared above. Dots ("•") and crosses ("×") denote the real and imaginary parts ($\Re(c_k)$ and $\Im(c_k)$) and the length of the vertical lines the absolute value ($|c_k|$) of the complex amplitudes c_k.

[18] Of course, $U(\mathbf{t},\mathbf{s})$ *does* have a deterministic effect for $\rho_{\mathbf{t}} \neq |0\rangle\langle 0|$, just that this effect is not part of the operator's declared semantics and is considered to be an implementation detail.

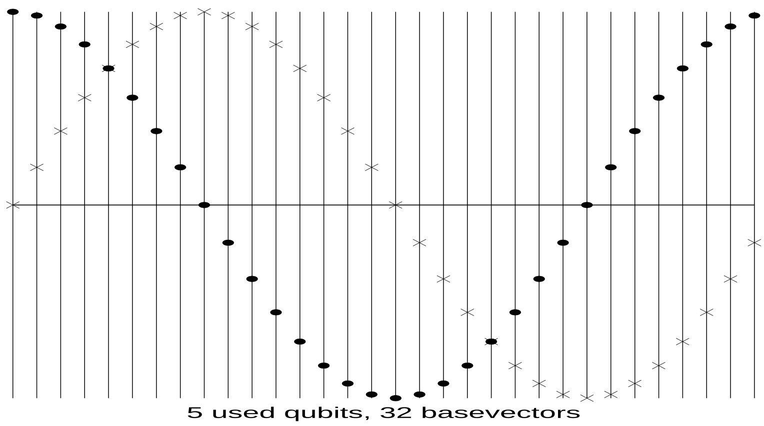


Figure 2.9: *5-qubit test state*

Scratch Parameters A *scratch register* **s** (QCL type `quscratch`) is expected to be empty before and after the operator is called, so U has has to be of the form

$$U = (|0\rangle_{\mathbf{s}}\langle 0|_{\mathbf{s}}) \otimes \sum_{ij} u_{ij}|i\rangle\langle j| + \sum_{k,l=1}^{2^{|\mathbf{s}|}-1} \sum_{ij} u_{klij}|k\rangle_{\mathbf{s}}|i\rangle\langle l|_{\mathbf{s}}\langle j| \qquad (2.48)$$

Scratch parameters (also called *explicit* scratch registers), have the same semantics as unmanaged (see 2.5.2.4) local scratch variables (QCL-type `qureg`), except that they are provided as parameters instead of getting allocated from the quantum heap (see 2.4.1.2). Explicit scratch parameters can be useful for suboperators as they allow to temporarily use target registers, or other registers that are known to be empty at some point during a computation, as scratch to save quantum memory.

2.5.2.3 Inverse Operators

Unlike procedures, operator calls can be inverted. In QCL, this is done by preceding the operator name with the adjungation prefix "`!`". The adjoint operator of a composition of unitary operators is[19]

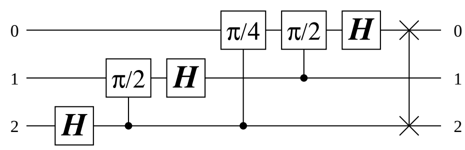
$$\left(\prod_{i=1}^{n} U_i\right)^{\dagger} = \prod_{i=1}^{n} U_{n+1-i}^{\dagger} \qquad (2.49)$$

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

In the circuit model, the inverse operator is simply the execution of the circuit from right to left, whereby gates are replaced by their adjoint gates.

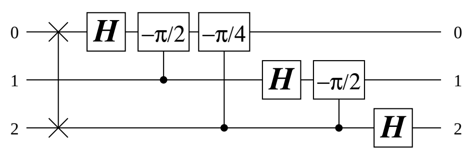
```
qcl> qureg q[3];                        // allocate 3-qubit register
qcl> H(q[1]);                           // prepare test state
[3/32] 0.70711 |0> + 0.70711 |2>
qcl> set log 1;                         // show gate sequence
qcl> dft(q);                            // quantum Fourier transform
```

```
@ H(qureg q=<2>)
@ V(real phi=1.5708,quconst q=<1,2>)
@ H(qureg q=<1>)
@ V(real phi=0.785398,quconst q=<0,2>)
@ V(real phi=1.5708,quconst q=<0,1>)
@ H(qureg q=<0>)
@ Swap(qureg a=<0>,qureg b=<2>)
```




```
[3/32] 0.5 |0> + (0.25+0.25i) |1> + (0.25-0.25i) |3> +
       0.5 |4> + (0.25+0.25i) |5> + (0.25-0.25i) |7>
qcl> !dft(q);                           // inverse transform
```

```
@ !Swap(qureg a=<0>,qureg b=<2>)
@ !H(qureg q=<0>)
@ !V(real phi=1.5708,quconst q=<0,1>)
@ !V(real phi=0.785398,quconst q=<0,2>)
@ !H(qureg q=<1>)
@ !V(real phi=1.5708,quconst q=<1,2>)
@ !H(qureg q=<2>)
```




```
[3/32] 0.70711 |0> + 0.70711 |2>
```

Delayed Execution Since the sequence of applied suboperators is specified using a procedural language which cannot be executed in reverse, the adjungation is is achieved by the *delayed execution* of suboperator calls.

Whenever a suboperator call is encountered during the execution of an inverted operator, the name of the suboperator and its evaluated arguments are pushed on the *execution stack*. Afterwards, the stacked suboperator calls are processed in reverse order. Thereby, normal calls are replaced by inverted calls and vice-versa.

2.5.2.4 Scratch Registers

Let U be an arbitrary k-qubit unitary operator on $\mathcal{B}^{\otimes k}$. While any universal set of gates $G = \{G_1, G_2 \ldots\}$ allows the direct implementation of U as

$$U^{(1)} = \prod_i G_{n_i}(\mathbf{s}_i) = U \quad \text{with} \quad \mathbf{s}_i \in R^k, \tag{2.50}$$

the implementation of a $(k+s)$-qubit operator

$$U^{(2)} = \prod_j G_{n_j}(\mathbf{s}_j) \quad \text{with} \quad \mathbf{s}_j \in R^{k+s} \tag{2.51}$$

such that

$$U^{(2)}\,|\psi\rangle|0\rangle = (U|\psi\rangle)\,|0\rangle \quad \text{for all} \quad |\psi\rangle \in \mathcal{B}^{\otimes k} \tag{2.52}$$

may be considerably more efficient.

In quantum programming, we consider $U^{(1)}$ and $U^{(2)}$ as equivalent and refer to $U^{(2)}$ as an implementation of U with *s scratch qubits*.

Formally, a scratch register is a local quantum variable defined within the body of an operator definition. As all other quantum variables, local registers are empty on allocation (see 2.4.1.2). While, in contrast to quantum functions (see 2.5.3.3), for general operators there is no way to automatically "clean-up" scratch registers, the implementation itself has to assure that a local register $\mathbf{s}$ is empty (i.e. in the state $\rho_{\mathbf{s}} = |0\rangle\langle 0|$) after the call *(unmanaged scratch space)*.

In QCL, unmanaged scratch registers are local variables of type `qureg`. The operator below uses a scratch qubit to implement the conditional phase gate

$$V(\varphi) : |k\rangle_{\mathbf{q}} \to \begin{cases} \mathrm{e}^{\mathrm{i}\phi}|k\rangle_{\mathbf{q}} & \text{if } k = 2^{|\mathbf{q}|}-1 \\ |k\rangle_{\mathbf{q}} & \text{otherwise} \end{cases} \tag{2.53}$$

using the single qubit Z-rotation R_z (see 1.4.2.2) and the generalized `CNot`-gate

$$\texttt{CNot}(\mathbf{q},\mathbf{p}) = \texttt{Not}_{[[\mathbf{p}]]}(\mathbf{q}) = C^{|\mathbf{p}|}[X^{\otimes|\mathbf{q}|}](\mathbf{q},\mathbf{p}) \tag{2.54}$$

```
operator cphase(real phi,quconst q) {   // Conditional phase gate
  qureg s[1];                           // single scratch qubit
  CNot(s,q);                            // s=1 if q=111...
  RotZ(phi,s);                          // add phase if s=1
  CNot(s,q);                            // restore scratch qubit
}
```

For an n-qubit argument register `cphase`(φ) can be written as the $(n+1)$-qubit matrix

$$\texttt{cphase}(\varphi) = \begin{pmatrix} \mathrm{e}^{-\mathrm{i}\frac{\varphi}{2}}\,C^n[\mathrm{e}^{\mathrm{i}\varphi}] & 0 \\ 0 & \mathrm{e}^{\mathrm{i}\frac{\varphi}{2}}\,C^n[\mathrm{e}^{-\mathrm{i}\varphi}] \end{pmatrix} \tag{2.55}$$

on $\mathcal{B}^{\otimes n+1}$ and implements $V(\varphi,\mathbf{q})$ up to an irrelevant global phase as

$$\texttt{cphase}(\varphi)\,|\psi\rangle_{\mathbf{q}}|0\rangle_{\mathbf{s}} = \left(\mathrm{e}^{-\mathrm{i}\varphi/2}\,V(\varphi)|\psi\rangle_{\mathbf{q}}\right)|0\rangle_{\mathbf{s}}. \tag{2.56}$$

```
qcl> qureg q[2];                        // allocate 2 qubits
qcl> H(q);                              // prepare test state
[2/32] 0.5 |0> + 0.5 |1> + 0.5 |2> + 0.5 |3>
qcl> cphase(pi,q);                      // rotate phase for |3>
[2/32] -0.5i |0> - 0.5i |1> - 0.5i |2> + 0.5i |3>
```

2.5.3 Basis Permutations

Because of the linearity of unitary transformations, an operator applied to a superposition state $|\Psi\rangle$ is simultaneously applied to all basis vectors that constitute $|\Psi\rangle$ since

$$U \sum_i c_i|i\rangle = \sum_i c_i(U\,|i\rangle). \tag{2.57}$$

This feature is called *quantum parallelism* and is exploited in most non-classical algorithms (see 1.5.2.4).

Often U implements a reversible boolean, or, equivalently, a bijective integer function, by treating the basis vectors merely as bitstrings or binary numbers.

Definition 39 *An n-qubit basis permutation is a unitary operator of the form $F : |k\rangle \to |f(k)\rangle$ with f being a bijective function (i.e. a permutation) on $\mathbf{Z}_{2^n}$ (or $\mathbf{B}^n$).*

2.5.3.1 Reversible Boolean Networks

While general operators implement quantum circuits, basis permutations are procedural descriptions of reversible boolean networks operating on qubits. This allows us to discuss computations on qubits analogously to classical bits, so we can e.g. describe the effect of the controlled-not operation $C[X](\mathbf{a}, \mathbf{b})$ as "if qubit $\mathbf{b}$ is set then invert qubit $\mathbf{a}$".

The set $L = \{X, C[X], C^2[X]\}$ is universal for basis permutations.[20] The gates in L can be generalized to the `CNot`-gate (2.54) which operates on argument and target registers of arbitrary sizes.

In QCL, basis permutations are represented by the subroutine type `qufunct`. Fig. 2.10 shows the 4-qubit circuit generated by the operator

```
qufunct inc(qureg x) {                  // Increment register
  int i;
  for i = #x-1 to 0 step -1 {
    CNot(x[i],x[0::i]);                 // apply controlled-not
  }                                     //   from MSB to LSB
}
```

which implements the basis vector incrementation

$$\texttt{inc} : |k\rangle_{\mathbf{x}} \to |(k+1) \bmod 2^{|\mathbf{x}|}\rangle_{\mathbf{x}}. \tag{2.58}$$

[20] In fact, the Toffoli gate $T = C^2[X]$ alone would be universal if two additional constant qubits in the state $|1\rangle$ are available.

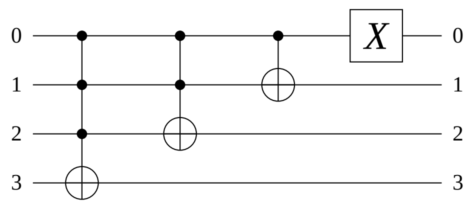


Figure 2.10: *Increment operator*

```
qcl> qureg q[8];                          // allocate quantum byte
qcl> H(q[2]&q[5]); CNot(q[0],q[2]);     // prepare test state
[6/32] 0.5 |0> + 0.5 |5> + 0.5 |32> + 0.5 |37>
qcl> inc(q);                              // increment basis vectors
[6/32] 0.5 |1> + 0.5 |6> + 0.5 |33> + 0.5 |38>
qcl> inc(q);
[6/32] 0.5 |2> + 0.5 |7> + 0.5 |34> + 0.5 |39>
qcl> !inc(q);                             // decrement basis vectors
[6/32] 0.5 |1> + 0.5 |6> + 0.5 |33> + 0.5 |38>
```

2.5.3.2 Non-Boolean Factorizations

According to the hierarchy of subroutines (see 2.5.1.1), any routine can only call subroutines of the same or a more restricted type. This especially means that basis permutations may not call general operators or gates within their definition. However, there are universal sets of gates G where $L \not\subseteq G$. One example would be the *standard set* (see 1.4.2.6) $\{H, S, T, C[X]\}$ which lacks the `Not`- and the Toffoli-gate.

A procedural QPL therefore has to provide a way to circumvent the hierarchy of subroutines in order to define non-boolean implementations for elementary basis permutations. In QCL this can be achieved by a *double declaration*, e.g.

```
qufunct operator Not(qureg q) {    // Standard set implementation
  int i;                           // of the generalized Not-gate
  for i=0 to #q-1 {                // for all qubits:
    H(q[i]);                       //   transform into dual basis
    S(q[i]); S(q[i]);              //   S^2 = Z = |0><0|-|1><1|
    H(q[i]);                       //   transform back into
  }                                //   computational basis
}
```

2.5.3.3 Quantum Functions

One obvious problem in quantum computing is its restriction to reversible computations. Consider a simple operation like counting the number of set qubits in a register $\mathbf{q}$

$$\texttt{bitcount}' : |k\rangle_{\mathbf{q}} \to |b(k)\rangle_{\mathbf{q}} \quad \text{with} \tag{2.59}$$

$$b(n) = n \bmod 2 + b(\lfloor n/2 \rfloor) \quad \text{and} \quad b(0) = 0. \tag{2.60}$$

Clearly, this operation is non-reversible since $\texttt{bitcount}'|2^n\rangle = 1$ for all $n < |\mathbf{q}|$, so $\texttt{bitcount}'$ is not an unitary operator. However, if we use an additional register $\mathbf{p}$ with $2^{|\mathbf{p}|} > |\mathbf{q}|$, then we can always find a unitary operator `bitcount` such that

$$\texttt{bitcount} : |k\rangle_{\mathbf{q}}|0\rangle_{\mathbf{p}} \to |k\rangle_{\mathbf{q}}|b(k)\rangle_{\mathbf{p}} \tag{2.61}$$

Definition 40 *Let $\mathbf{c} \in R_n$ and $\mathbf{t} \in R_m$ be disjoint registers and f be an arbitrary function $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^m}$ (or $f : \mathbf{B}^n \to \mathbf{B}^m$). A unitary operator F of the form*

$$F : |x\rangle_{\mathbf{c}}|y\rangle_{\mathbf{t}} \to |x\rangle_{\mathbf{c}}|g(f(x), y)\rangle_{\mathbf{t}} \quad \text{with} \quad g(z, 0) = z \tag{2.62}$$

is referred to as quantum function implementing f with the accumulation function g.

Since F is required to be unitary, each $h_z : y \to g(z, y)$ must be a 2^m-permutation with $h_z(0) = z$. So for any given function $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^m}$ there exist $2^n(2^m - 1)!$ different quantum functions $F^{(k)}$ which implement f.

In procedural quantum programming, two quantum functions $F^{(1)}$ and $F^{(2)}$ are considered equivalent when they implement the same function, so $F|x\rangle_{\mathbf{t}}|y\rangle_{\mathbf{t}}$ is undefined for $y \neq 0$ which makes the actual definition of g an irrelevant implementation detail.[21]

The above notion of equivalence implies that $\mathbf{t}$ is a *target register* (see 2.5.2.2). Also, from the form of (2.62) it follows that $\mathbf{c}$ is a *constant register.* Using the QCL register types `quconst` and `quvoid`, we can implement `bitcount` as the quantum function

```
qufunct bitcount(quconst q,quvoid p) {  // count set bits in q
  int i; int j;
  if #q>=2^#p {                         // make sure that p is wide
    exit "target register too small";   // enough
  }
  for i=0 to #q-1 {                     // iterate over qubits in q
    for j=#p-1 to 0 step -1 {           //   increment p if q[i]
      CNot(p[j],p[0::j] & q[i]);        //   is set
    }
  }
}
```

[21] This abstraction is necessary as the use of scratch registers (see 2.5.3.5) can affect g.

with the accumulation function $g(z, y) = (z + y) \bmod 2^{|\mathbf{p}|}$, so

$$\texttt{bitcount} : |x\rangle_{\mathbf{q}}|y\rangle_{\mathbf{p}} \to |x\rangle_{\mathbf{q}}|(y + b(k)) \bmod 2^{|\mathbf{p}|}\rangle_{\mathbf{p}}. \tag{2.63}$$

```
qcl> qureg q[3]; qureg p[2];  // allocate argument and target reg.
qcl> H(q);                    // prepare superposition
[5/32] 0.3535 |0,0> + 0.3535 |1,0> + 0.3535 |2,0> + 0.3535 |3,0> +
       0.3535 |4,0> + 0.3535 |5,0> + 0.3535 |6,0> + 0.3535 |7,0>
qcl> bitcount(q,p);           // count set qubits
[5/32] 0.3535 |0,0> + 0.3535 |1,1> + 0.3535 |2,1> + 0.3535 |4,1> +
       0.3535 |3,2> + 0.3535 |5,2> + 0.3535 |6,2> + 0.3535 |7,3>
```

2.5.3.4 Junk Registers

While quantum functions can be used to work around the reversible nature of quantum computing, the necessity to keep a copy of the argument is a problem, as longer computations will leave registers filled with intermediate results.

Suppose we want to compare the number of set qubits in two registers **a** and **b**, i.e. find a quantum function to implement the predicate

$$c(x, y) = \begin{cases} 1 & \text{if } b(x) = b(y) \\ 0 & \text{otherwise .} \end{cases} \tag{2.64}$$

Using an auxiliary register **s** and $B = \texttt{bitcount}$ from (2.63), the operator

$$\texttt{bitcmp0}(\mathbf{a}, \mathbf{b}, \mathbf{t}, \mathbf{s}) = \texttt{CNot}(\mathbf{t}, \mathbf{s})\, \texttt{Not}(\mathbf{s})\, B^\dagger(\mathbf{b}, \mathbf{s})\, B(\mathbf{a}, \mathbf{s}) \tag{2.65}$$

implements c but leaves the register **s** in a "dirty" state:

$$|x, y, 0, 0\rangle \xrightarrow{B(\mathbf{a},\mathbf{s})} |x, y, b(x), 0\rangle \xrightarrow{B^\dagger(\mathbf{b},\mathbf{s})} |x, y, b(x) - b(y), 0\rangle \xrightarrow{\texttt{Not}(\mathbf{s})} \tag{2.66}$$

$$|x, y, 11...1 \oplus (b(x) - b(y)), 0\rangle \xrightarrow{\texttt{CNot}(\mathbf{t},\mathbf{s})} |x, y, 11...1 \oplus (b(x) - b(y)), c(x, y)\rangle$$

```
qufunct bitcmp0(quconst a,quconst b,quvoid t,quvoid s) {
  bitcount(a,s);         // write #bits(a) to s
  !bitcount(b,s);        // subtract b(b) from s
  Not(s);                // invert s
  CNot(t,s);             // t=1 if #bits(a)-#bits(b)=0
}
```

Since **s** does not contain useful information and is usually entangled (i.e. $\mathrm{tr}(\rho_{\mathbf{s}}^2) < 1$) so that it cannot be reset by measurement (see 2.4.3.3) without affecting other registers, **s** is called a *junk register*. A quantum function

$$F : |x, 0\rangle|0\rangle_{\mathbf{s}} \to |x, f(x)\rangle|j(x)\rangle_{\mathbf{s}} \tag{2.67}$$

with a junk parameter **s** is called a *dirty* implementation of f.

```
qcl> qureg a[3];qureg b[3];          // allocate registers a,b,s,t
qcl> qureg s[2];qureg t[1];
qcl> H(a[0]); Not(a[2]);             // prepare test state for a
[9/32] 0.70711 |4,0,0,0> + 0.70711 |5,0,0,0>
qcl> H(b[1]); Not(b[0]);             // prepare test state for b
[9/32] 0.5 |4,1,0,0> + 0.5 |5,1,0,0> + 0.5 |4,3,0,0> + 0.5 |5,3,0,0>
qcl> bitcmp(a,b,t,s);                // compare number of bits
[9/32] 0.5 |4,3,0,0> + 0.5 |5,1,2,0> + 0.5 |4,1,3,1> + 0.5 |5,3,3,1>
```

2.5.3.5 Scratch Space Management

Let F be a a dirty implementation of a classical function f with the argument register $\mathbf{x}$, the target register $\mathbf{y}$ and the junk register $\mathbf{s}$

$$F : |k\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \to |k\rangle_{\mathbf{x}}|f(k)\rangle_{\mathbf{y}}|j(k)\rangle_{\mathbf{s}} \tag{2.68}$$

While computing $f(k)$, F also fills $\mathbf{s}$ with the temporary junk bits $j(k)$. To reclaim $\mathbf{s}$, Bennett proposed the following method [7], which is known as *uncomputing*:

1. Allocate an (empty) auxiliary register $\mathbf{t}$ of the same size as $\mathbf{y}$.
2. Replace $F(\mathbf{x},\mathbf{y},\mathbf{s})$ by the operator

$$F'(\mathbf{x},\mathbf{y},\mathbf{s},\mathbf{t}) = F^\dagger(\mathbf{x},\mathbf{t},\mathbf{s})\,\mathtt{Fanout}(\mathbf{t},\mathbf{y})\,F(\mathbf{x},\mathbf{t},\mathbf{s}). \tag{2.69}$$

The above procedure restores both the junk and the auxiliary register so $\mathbf{s}$ and $\mathbf{t}$ are scratch parameters (see 2.5.2.2) of F' so F' is a clean implementation of f with $|\mathbf{s}|+|\mathbf{t}|$ scratch qubits (see 2.5.2.4):

$$\begin{aligned} |k,0,0,0\rangle &\xrightarrow{F(\mathbf{x},\mathbf{t},\mathbf{s})} |k,0,j(k),f(k)\rangle \xrightarrow{\mathtt{Fanout}(\mathbf{t},\mathbf{y})} \\ |k,f(k),j(k),f(k)\rangle &\xrightarrow{F^\dagger(\mathbf{x},\mathbf{t},\mathbf{s})} |k,f(k),0,0\rangle \end{aligned} \tag{2.70}$$

The Fanout Operator The `Fanout` operator is a quantum function implementing the identity i.e.

$$\mathtt{Fanout} : |x\rangle_{\mathbf{a}}|0\rangle_{\mathbf{b}} \to |x\rangle_{\mathbf{a}}|x\rangle_{\mathbf{b}} \quad \text{with} \quad |\mathbf{a}| = |\mathbf{b}| \tag{2.71}$$

and is usually realized using $|\mathbf{x}|$ controlled-not gates with the accumulation function $g_F(x,y) = x \oplus y$.

```
cond 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]); }
}
```

Let F be a dirty implementation of f with the accumulation function g and a junk register $\mathbf{s}$, so

$$F : |x\rangle_{\mathbf{x}}|y\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \to |x\rangle_{\mathbf{x}}|g(f(x), y)\rangle_{\mathbf{y}}|j(x, y)\rangle_{\mathbf{s}}. \tag{2.72}$$

The clean version F' according to (2.69) with the scratch registers $\mathbf{s}$ and $\mathbf{t}$ is given as

$$F' : |x\rangle_{\mathbf{x}}|y\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}}|0\rangle_{\mathbf{t}} \to |x\rangle_{\mathbf{x}}|g_F(f(x), y)\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}}|0\rangle_{\mathbf{t}}. \tag{2.73}$$

So in F' the original accumulation function g is replaced by the accumulation function g_F of the `Fanout` operator.

Scratch Registers In procedural quantum programming, the method of uncomputing (2.69) allows the automatically reclaiming of local registers which are left in a non-empty state after the body of the operator has been executed *(managed scratch registers).* In QCL managed scratch registers are local variables of type `quscratch`.

Since the method of uncomputing only works for quantum functions, managed scratch registers are restricted to basis permutations with constant and target registers as quantum parameters.

For `bitcmp0` from (see 2.5.3.4), a clean version can be constructed by making $\mathbf{s}$ a managed local scratch register

```
qufunct bitcmp(quconst a,quconst b,quvoid t) {
  quscratch s[ceil(log(max(#a,#b)+0.5,2))];
  bitcmp0 (a,b,t,s);
}
```

which is not only equivalent to but also implements the same unitary transformation[22] as the following operator using two unmanaged scratch registers (see 2.5.2.4).

```
qufunct bitcmp(quconst a,quconst b,quvoid t) {
  qureg s[ceil(log(max(#a,#b)+0.5,2))];
  qureg u[#t];
  bitcmp0 (a,b,u,s);
  CNot(t,u);
  !bitcmp0 (a,b,u,s);
}
```

[22]Note that for single qubits, every possible implementation of the `Fanout` operator is identical to a `CNot` gate.

2.5.3.6 Calling Semantics

The potential use of managed scratch registers extends the calling semantics of basis permutations as compared to general operators. Consider the quantum functions

```
qufunct U1(quconst x,quvoid y,quvoid s) {  // s is junk register
  A(x,y,s);
  B(x,y,s)
}

qufunct U2(quconst x,quvoid y) {           // s is scratch register
  quscratch s;
  A(x,y,s);
  B(x,y,s)
}
```

While the calls `U1(x,y,s)` and `!U1(x,y,s)` are analogous to general operator calls i.e.

$$U_1(\mathbf{x},\mathbf{y},\mathbf{s}) = B(\mathbf{x},\mathbf{y},\mathbf{s})\,A(\mathbf{x},\mathbf{y},\mathbf{s}) \quad \text{and} \quad U_1^\dagger(\mathbf{x},\mathbf{y},\mathbf{s}) = A^\dagger(\mathbf{x},\mathbf{y},\mathbf{s})\,B^\dagger(\mathbf{x},\mathbf{y},\mathbf{s}) \tag{2.74}$$

the calls `U2(x,y,s)` and `!U2(x,y,s)` cause the transparent allocation of a register $\mathbf{t}$, with $|\mathbf{t}| = |\mathbf{y}|$ and results in the transformations

$$U_2(\mathbf{x},\mathbf{y},\mathbf{s},\mathbf{t}) = U_1^\dagger(\mathbf{x},\mathbf{t},\mathbf{s})\,\mathtt{Fanout}(\mathbf{t},\mathbf{y})\,U_1(\mathbf{x},\mathbf{t},\mathbf{s}) \quad \text{and} \tag{2.75}$$

$$U_2^\dagger(\mathbf{x},\mathbf{y},\mathbf{s},\mathbf{t}) = U_1^\dagger(\mathbf{x},\mathbf{t},\mathbf{s})\,\mathtt{Fanout}^\dagger(\mathbf{t},\mathbf{y})\,U_1(\mathbf{x},\mathbf{t},\mathbf{s}). \tag{2.76}$$

Note that U_2 and $U_2^\dagger$ are almost identical and that $U_2 = U_2^\dagger$ if a Hermitian implementation of the `Fanout` operator is used.

Double Execution When a general operator or basis permutation without managed scratch registers is executed, calls of suboperators are either processed immediately (linear execution) or pushed on the execution stack (delayed execution), depending on whether the normal or the inverse operator has been called (see 2.5.2.3).

In the case of a quantum function with managed scratch registers, after the initial remapping of the target registers, suboperator calls always get executed *and* pushed on the execution stack *(double execution)*. After the body of the subroutine has been processed, the appropriate `Fanout` (normal call) or $\mathtt{Fanout}^\dagger$ (inverted call) operation is executed and then the stacked calls get executed again in reverse order with their adjungation flags inverted.

2.6 Quantum Flow Control

All classical programming languages, in one way or another, allow the conditional execution of code depending on a boolean variable or logical expression *(conditional branching)*. In structured languages conditional branching is realized by if-statements with well defined entry and exit points.

While the unobservability of qubits forbids a direct implementation, we will show how conditional operators can be used to emulate conditional branching on qubits *(quantum if-statement)* as well as (bounded) conditional loops with quantum termination conditions to the effect that, with regard to flow-control, qubits can be treated almost equivalently to classical bits *(structured quantum programming)*.

We will further demonstrate how qubits and registers can be combined to arbitrary boolean expressions and complex predicates *(quantum conditions)*.

2.6.1 Conditional Operators

As already mentioned in 2.4.2.3, for a unitary register operator $U(\mathbf{s})$ and an enable (or control) register $\mathbf{e} \in R$ disjoint to $\mathbf{s}$, the conditional operator $U_{[[\mathbf{e}]]}(\mathbf{s})$ is defined as

$$U_{[[\mathbf{e}]]}(\mathbf{s}) : |k\rangle_{\mathbf{s}}|c\rangle_{\mathbf{e}} \to \begin{cases} (U\,|k\rangle_{\mathbf{s}})|c\rangle_{\mathbf{e}} & \text{if } c = 111\ldots \\ |k\rangle_{\mathbf{s}}|c\rangle_{\mathbf{e}} & \text{otherwise .} \end{cases} \tag{2.77}$$

If $\mathbf{e}$ is a qubit so $|\mathbf{e}| = 1$, then informally, we can describe the effect of $U_{[[\mathbf{e}]]}(\mathbf{s})$ as "if $\mathbf{e}$ is set then apply $U(\mathbf{s})$" which relates to the fact that

$$U_{[[\mathbf{e}]]} \sum_{k=0}^{2^{|\mathbf{s}|}-1} \sum_{b \in \mathbf{B}} c_{kb}|k\rangle_{\mathbf{s}}|b\rangle_{\mathbf{e}} = \sum_{k=0}^{2^{|\mathbf{s}|}-1} c_{kb}|k\rangle_{\mathbf{s}}|0\rangle_{\mathbf{e}} + \sum_{k=0}^{2^{|\mathbf{s}|}-1} c_{kb}\,(U\,|k\rangle_{\mathbf{s}})\,|1\rangle_{\mathbf{e}}. \tag{2.78}$$

2.6.1.1 Properties of Conditional Operators

Orthogonal Enable Registers Let $U, V \in SU(2^n)$ be n-qubit unitary operators, $\mathbf{s} \in R_n$ a n-qubit register and $\mathbf{q} \in R_1$ a qubit disjoint to $\mathbf{s}$ (i.e. $\mathbf{q} \subseteq \bar{\mathbf{s}}$), then

$$\left[U_{[[\mathbf{q}]]}(\mathbf{s})\,X(\mathbf{q}), V_{[[\mathbf{q}]]}(\mathbf{s})\,X(\mathbf{q})\right] = 0. \tag{2.79}$$

For arbitrary register operators $U(\mathbf{x})$ and $V(\mathbf{y})$, (2.79) can be generalized to

$$\left[U_{[[\mathbf{e}]]}(\mathbf{x})\,X^n(\mathbf{c}), V_{[[\mathbf{e}]]}(\mathbf{y})\,X^n(\mathbf{c})\right] = 0 \tag{2.80}$$

with $\mathbf{c} \in R_n$ being an non-empty subregister $\mathbf{c} \subseteq \mathbf{e}$ and $\mathbf{x}, \mathbf{y} \subseteq \bar{\mathbf{e}}$.

Conditional Decomposition Its easy to verify that for an arbitrary $U(\mathbf{s})$ and a qubit $\mathbf{q} \in R_1, \mathbf{q} \subseteq \bar{\mathbf{s}}$

$$U(\mathbf{s}) = U_{[[\mathbf{q}]]}(\mathbf{s})\, X(\mathbf{q})\, U_{[[\mathbf{q}]]}(\mathbf{s})\, X(\mathbf{q}). \tag{2.81}$$

Definition 41 *An n-qubit basis permutation $P : |k\rangle \to |p(k)\rangle$ is called cyclic iff it generates the computational basis B such that for any $|k\rangle \in B$*

$$\bigcup_{i=0}^{2^n-1} P^i |k\rangle = \{|j\rangle \mid j \in \mathbf{Z}_{2^n}\} = B \tag{2.82}$$

p is therefore a permutation on $\mathbf{Z}_{2^n}$ with the cycle length 2^n. If P is cyclic, then $A^\dagger\, P\, A$ is also cyclic for an arbitrary basis permutations A. An example for a cyclic basis permutation is the operator `inc` (2.58) from 2.5.3.1.

We can now generalize the identity (2.81) for enable registers $\mathbf{e} \in R_n, \mathbf{e} \subseteq \bar{\mathbf{s}}$ to

$$U(\mathbf{s}) = \left(U_{[[\mathbf{e}]]}(\mathbf{s})\, P(\mathbf{e})\right)^{2^n} \tag{2.83}$$

with P being an arbitrary n-qubit cyclic basis permutation.

Conditional Composition Let $V, W \in SU(2^n)$ be n-qubit unitary operators, then the selection operator

$$U = \begin{pmatrix} V & 0 \\ 0 & W \end{pmatrix} \tag{2.84}$$

can be implemented as

$$U(\mathbf{s}, \mathbf{c}) = X(\mathbf{c})\, V_{[[\mathbf{c}]]}(\mathbf{s})\, X(\mathbf{c})\, W_{[[\mathbf{c}]]}(\mathbf{s}) \tag{2.85}$$

Likewise, for any $(n+m)$-qubit selection operator

$$U = \operatorname{diag}(U_0, U_1 \ldots U_{2^m-1}) \tag{2.86}$$

there exists an cyclic m-qubit basis permutation P such that

$$U(\mathbf{s}, \mathbf{c}) = \prod_{i=0}^{2^m-1} U_{i\,[[\mathbf{c}]]}(\mathbf{s})\, P(\mathbf{c}) \tag{2.87}$$

2.6.1.2 Conditional Subroutines

As conditional operators are just a special case of selection operators (see 2.5.2.2), they can be implemented as quantum subroutines with a constant enable register.

The operator below implements a conditional version `cinc` of the register increment operator (2.58) defined as

$$\texttt{cinc} : |k\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} \to \begin{cases} |(k+1) \bmod 2^{|\mathbf{x}|}\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} & \text{if } c = 11...1 \\ |k\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{2.88}$$

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

The above operator is a conditional version of (2.58)

$$\texttt{cinc} : |k\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} \to \begin{cases} |(k+1) \bmod 2^{|\mathbf{x}|}\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} & \text{if } c = 11...1 \\ |(k\rangle_{\mathbf{x}}|c\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{2.89}$$

Structured QPLs directly support conditional execution, i.e. the automatic construction of $U_{[[\mathbf{e}]]}$ from a given implementation of U, by allowing the explicit declaration of *conditional subroutines*.

In QCL, conditional operators can be defined by prefixing the subroutine declaration with the keyword `cond`.

```
cond qufunct inc(qureg x) {        // Conditional increment register
  int i;                           // as conditional
  for i = #x-1 to 0 step -1 {      // subroutine
    CNot(x[i],x[0::i]);
  }
}
```

Conditional Calls The enable register **e** is an implicit constant parameter and not part of the parameter declaration of a conditional subroutine. Instead, **e** is set by a quantum if-statement (see 2.6.2.1) and transparently passed on to all suboperators, which therefore are also required to be either conditional operators or conditional elementary gates (see 2.4.2.3).

```
qcl> qureg q[4];qureg e[1];   // allocate counting and control reg.
qcl> H(q[3] & e);             // prepare test state
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |0,1> + 0.5 |8,1>
qcl> cinc(q,e);               // conditional increment
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |1,1> + 0.5 |9,1>
qcl> if e { inc(q); }         // equivalent to cinc(q,e)
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |2,1> + 0.5 |10,1>
qcl> !cinc(q,e);              // conditional decrement
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |1,1> + 0.5 |9,1>
qcl> if e { !inc(q); }        // equivalent to cinc(q,e);
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |0,1> + 0.5 |8,1>
```

In the example above, the if-statement "`if e { inc(q); }`" causes the control qubit **e** to be appended to the enable registers of the `CNot`-gates generated by the call `inc(q)`, so "`if e { inc(q); }`" and `cinc(q,e)` describe the same quantum circuit (fig. 2.11).

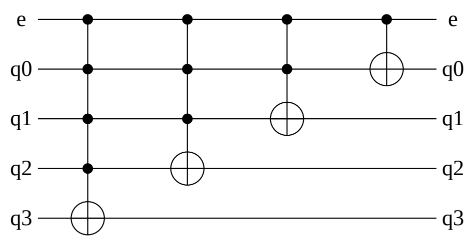


Figure 2.11: *Conditional increment operator*

Unconditional Calls If a conditional subroutine is called outside of a quantum if-statement[23] then **e** is empty *(unconditional call)* and call semantics are identical to unconditional subroutines.

```
qcl> inc(q);                  // unconditional increment
[4/32] 0.5 |1,0> + 0.5 |9,0> + 0.5 |1,1> + 0.5 |9,1>
```

Since the declaration of an operator as conditional does not incur any overhead on unconditional calls, it is reasonable to always declare routines as conditional if all required suboperators are available as conditional subroutines or gates.

[23] In the case of a suboperator, this also includes all parent operators

2.6.1.3 Call Graph of Subroutines

The hierarchy of subroutines (see 2.5.1.1) implies that routines may only call subroutines of the same or a more restricted type. If, in addition to that, we also take into account that conditional operators cannot call unconditional suboperators, the hierarchy becomes a lattice, which can be represented by the call graph depicted in fig. 2.12.

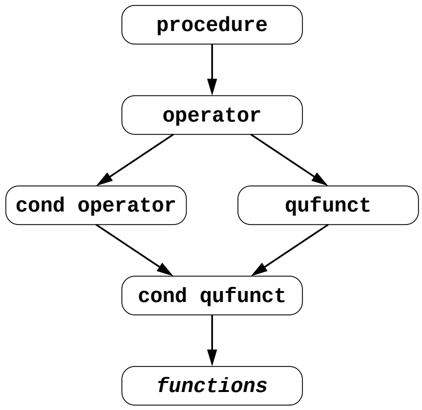


Figure 2.12: *Call graph of QCL subroutines*

2.6.1.4 Explicit Enable Registers

Conditional operators, by definition, must be able to handle enable registers of arbitrary sizes. As structured quantum programming is supposed to be a hardware independent formalism, it cannot be assumed that the quantum oracle provides a universal set of conditional gates. In fact, most standard gates do not operate on more than three qubits (see 1.4.2.4).

A structured QPL therefore has to allow for the implementation of basic conditional operators using unconditional suboperators and gates and consequently has to provide a means to make the enable register available as a symbolic quantum constant *(explicit enable register)*.

In QCL this is achieved by redeclaring the enable register as local register of type `quconst`. The example below is the QCL default implementation of the `Not`-gate:

```
extern qufunct NOT(qureg q);                  // unconditional Not
extern qufunct CNOT(qureg q,quconst c);       // unconditional CNot

cond qufunct Not(qureg q) {                   // conditional Not
  quconst e = cond;                           // e=enable register
  if #e>0 { CNOT(q,e); } else { NOT(q); }     //
}
```

2.6.2 Conditional Branching

2.6.2.1 Quantum If-Statements

In classical structured programming languages, the conditional execution of a statement sequence (block) σ depending on the value of a boolean variable p is expressed by an if-statement S of the form

$$\texttt{if } p \texttt{ then } \sigma_1, \sigma_2 \dots \sigma_n \texttt{ endif}$$

or, if one of two blocks σ and τ should be executed,

$$\texttt{if } p \texttt{ then } \sigma_1, \sigma_2 \dots \sigma_n \texttt{ else } \tau_1, \tau_2 \dots \tau_m \texttt{ endif}$$

If the instead of a classical boolean variable, the if-condition is a qubit **p**

$$\texttt{if } \mathbf{p} \texttt{ then } \sigma_1, \sigma_2 \dots \sigma_n \texttt{ else } \tau_1, \tau_2 \dots \tau_m \texttt{ endif}$$

then S is referred to as *quantum if-statement (QIS)*. S is called

(i) *invalid* iff σ or τ contain[24] procedure calls, measurements, reset or input commands or use random numbers.

(ii) *nested* iff σ or τ contain other quantum if-statements.

(iii) *dirty* iff σ or τ contain assignments or `break`-statements.

(iv) *clean* iff S is valid and not dirty.

(v) *simple* iff S is clean and $\tau = ()$ i.e. no else-branch is defined.

Block and Tail Operators Like the body of an operator subroutine, the if- and else-branches σ and τ are basically procedural descriptions of quantum circuits, or more precisely, of sequences of suboperators which get generated by linear execution. However, unlike a subroutine, σ (τ) has no declared interface, so its "parameters" are all classical and quantum variables that are visible in the current scope. Also, σ (τ) is not guaranteed to have mathematical semantics (dirty QIS).

[24] This also includes any sub-statements in control structures.

We will write $\bar{\sigma}$ ($\bar{\tau}$) to denote the composition of suboperators (*blockoperator*) which would be generated if σ (τ) is executed in place of S[25] and $\hat{\sigma}$ ($\hat{\tau}$) to denote the circuit that is generated when the execution is continued until the end of the current subroutine *(tail operator)*.[26] In global scope $\hat{\sigma}$ and $\hat{\tau}$ are undefined.

In the following example $\bar{\sigma}, \bar{\tau}, \hat{\sigma}$ and $\hat{\tau}$ would evaluate to

$$\bar{\sigma} = B(\mathbf{x}),\ \bar{\tau} = C(\mathbf{x}),\ \hat{\sigma} = D(\mathbf{x}, \mathbf{y})\, B(\mathbf{x}) \quad \text{and} \quad \hat{\tau} = D(\mathbf{x}, \mathbf{y})\, C(\mathbf{x}). \tag{2.90}$$

```
operator U(qureg x,qureg y) {
  A(x);
  if y { B(x); } else { C(x); }    // quantum if-statement
  D(x,y);
}
```

2.6.2.2 Conditional Execution

A *simple* QIS S is of the form

  if p then $\sigma_1, \sigma_2 \dots \sigma_n$ **endif**

and implements the transformation $U_S = \bar{\sigma}_{[[\mathbf{p}]]}$. So simple QISs can be used to set the enable registers of conditional operators (conditional calls, see 2.6.1.2).

Since $\mathbf{p}$ is already passed as an implicit parameter $\bar{\sigma}$ must not operate on $\mathbf{p}$ i.e. $\bar{\sigma} = \bar{\sigma}(\bar{\mathbf{p}})$, so the QIS below is not a miraculous implementation of a unitary erase operation, but simply triggers an error

```
qcl> if q { Not(q); }
! at "Not(q); ":
! runtime error: arguments overlap with quantum condition
```

Global Control Register Internally, the passing of enable registers to conditional subroutines is handled by a *global control register* $\mathbf{g}$.

Whenever a simple QIS is encountered, the following happens:

1. The appropriate enable register $\mathbf{p}$ is appended to $\mathbf{g}$.
2. The if-branch σ is executed.
3. $\mathbf{p}$ is removed from $\mathbf{g}$.

Whenever a call to a conditional gate is encountered, the current value of $\mathbf{g}$ is passed as enable register to the quantum oracle.

[25] i.e. the circuit that would be generated if S were a classical if-statement with $p = \texttt{true}$ ($p = \texttt{false}$)

[26] Note that therefore, even if no else-branch is defined and consequently $\tau = ()$, this does *not* imply that $\hat{\tau} = I$.

2.6.2.3 Quantum Selection

A clean QIS S with $\tau \neq ()$ is of the form

$$\texttt{if } \mathbf{p} \texttt{ then } \sigma_1, \sigma_2 \dots \sigma_n \texttt{ else } \tau_1, \tau_2 \dots \tau_m \texttt{ endif}$$

The semantics of S (i.e. the corresponding unitary transformation) are defined as the selection operator

$$U_S : |k\rangle|c\rangle_{\mathbf{p}} \to \begin{cases} (\bar{\sigma}|k\rangle)|c\rangle_{\mathbf{p}} & \text{if } c = 1 \\ (\bar{\tau}|k\rangle)|c\rangle_{\mathbf{p}} & \text{if } c = 0. \end{cases} \tag{2.91}$$

As U_S contains $\bar{\sigma}$ and $\bar{\tau}$ as suboperators, this means that, unlike a classical if-statement, *both* branches of S get executed. As we assumed that neither σ nor τ contain assignments or other statements which would change the program state, both branches cannot exhibit mutual side effects and this can be done without being inconsistent with classical semantics.[27]

U_S is realized as the conditional composition (2.85)

$$U_S = \texttt{Not}(\mathbf{p})\, \bar{\tau}_{[[\mathbf{p}]]}\, \texttt{Not}(\mathbf{p})\, \bar{\sigma}_{[[\mathbf{p}]]} = \begin{pmatrix} \bar{\tau} & 0 \\ 0 & \bar{\sigma} \end{pmatrix} \tag{2.92}$$

so the QIS on the left corresponds to the right sequence (see 2.6.1.2)

```
if p {
  inc(x);          //  cinc(x,p);   conditional increment
} else {           //  Not(p);      invert enable qubit
  !inc(x);         //  !cinc(x,p);  conditional decrement
}                  //  Not(p);      restore enable qubit
```

2.6.2.4 Quantum Forking

If the body of a classical if-statement S contains changes to the program state (e.g. assignments to local variables), then subsequent operator calls may differ, depending on whether the if- or the else-branch has been executed. If however, S is a (dirty) QIS, then both paths need to be executed in order to determine the corresponding conditional operator U_S, which allows for non-classical side-effects and can lead to classically inconsistent states.

To illustrate the problem, consider the operator

[27] An example for inconsistent code would be e.g.

$$\texttt{boolean } v := \texttt{true}$$
$$\texttt{if } \mathbf{p} \texttt{ then } v := \neg v \texttt{ else } v := \neg v \texttt{ endif}$$
$$\texttt{if } v \texttt{ then error } \textit{“classically unreachable code”} \texttt{ endif}$$

```
operator U(qureg s,quconst p) {
  int n;
  if p { n=1; } else { n=0; }        // dirty QIS
  V(n,s);                            // n=0 or n=1?
}
```

At first glance it seems impossible to come up with semantics $U(\mathbf{s},\mathbf{p})$ that are both, classically consistent and also provide a reasonable interpretation as quantum control statement because there is no reason to prefer one of the "obvious" candidates $V_0(\mathbf{s})$ and $V_1(\mathbf{s})$ over the other.

As a quantum analogy, we might say that the QIS brought $\mathbf{p}$ and the classical variable n into a correlated state, which would suggest the interpretation that $V_0(\mathbf{s})$ is applied to all basis vectors $|k\rangle_{\mathbf{s}}|0\rangle_{\mathbf{e}}$ and $V_1(\mathbf{s})$ to all basis vectors $|k\rangle_{\mathbf{s}}|1\rangle_{\mathbf{e}}$ i.e.

$$U : |k\rangle_{\mathbf{s}}|n\rangle_{\mathbf{p}} \to (V_n|k\rangle_{\mathbf{s}})|n\rangle_{\mathbf{p}} \tag{2.93}$$

While these semantics are symmetric[28] and have a reasonable quantum interpretation, they are not classically consistent as they leave the binding of n undefined. But here, the mathematical semantics of operators (see 2.5.1.2) come to the rescue, as they state that operators are defined by their unitary effect and may not change the program state. Since n is a local variable about to leave scope, we might as well leave it undefined. However, these semantics also imply that dirty QISs cannot be used in global scope or within procedures.

The above interpretation could also be described as extending the QIS to include the remainder of the operator, so `U` is considered to be equivalent to

```
operator U(qureg s,quconst p) {
  int n;
  if p { n=1; V(n,s); } else { n=0; V(n,s); }
}
```

As this can only be done explicitly if the QIS in question is not part of a control statement (like, e.g. a while-loop), a more general notion of what exactly constitutes the "remainder of a subroutine" is necessary. This is provided by the the tail operators $\hat{\sigma}$ and $\hat{\tau}$ (see 2.6.2.1).

Using $\hat{\sigma}$ and $\hat{\tau}$, the above method can be formalized so that for a dirty QIS S of the form

$$\texttt{if } \mathbf{p} \texttt{ then } \sigma_1, \sigma_2 \ldots \sigma_n \texttt{ else } \tau_1, \tau_2 \ldots \tau_m \texttt{ endif}$$

the transformation U_S implemented by S can be defined as

$$U_S : |k\rangle|c\rangle_{\mathbf{p}} \to \begin{cases} (\hat{\sigma}|k\rangle)|c\rangle_{\mathbf{p}} & \text{if } c = 11..1 \\ (\hat{\tau}|k\rangle)|c\rangle_{\mathbf{p}} & \text{otherwise .} \end{cases} \tag{2.94}$$

[28]i.e. treat both branches of the QIS equally.

and realized as

$$U_S = \mathtt{Not}(\mathbf{p})\,\hat{\tau}_{[[\mathbf{p}]]}\,\mathtt{Not}(\mathbf{p})\,\hat{\sigma}_{[[\mathbf{p}]]} = \begin{pmatrix} \hat{\tau} & 0 \\ 0 & \hat{\sigma} \end{pmatrix} \tag{2.95}$$

Threaded Execution Since the evaluation of $\hat{\sigma}$ and $\hat{\tau}$ involves not only the execution of the corresponding block of S but of the complete subroutine, a dirty QIS splits the classical flow of control into two separate threads *(quantum forking)*. Each thread is executed independently until the end of the subroutine is reached *(threaded execution)*. Because of (2.80), the generated quantum circuits $\hat{\sigma}$ and $\hat{\tau}$ can then be recombined to

$$U_S = \mathtt{Not}(\mathbf{p})\,\hat{\tau}_{[[\mathbf{p}]]}\,\mathtt{Not}(\mathbf{p})\,\hat{\sigma}_{[[\mathbf{p}]]} = \begin{pmatrix} \hat{\tau} & 0 \\ 0 & \hat{\sigma} \end{pmatrix} \tag{2.96}$$

Restrictions While the concept of threaded execution allows classical code to be conditionally executed depending on qubits, dirty QISs also impose several restrictions:

- Since different threads will generally result in different program states, dirty QISs may only appear in subroutines with mathematical semantics and are therefore restricted to operators.
- As the effect of a dirty QIS extends over the remainder of the routine, all reachable suboperators have to be conditional, even when the routine is not declared conditional and they are not part of a QIS.
- When a dirty QIS is part of a loop, then the enable registers have to be disjoint for each iteration.

2.6.2.5 Pseudo Measurements

A possible use of QISs is to formally accumulate the content of (unobservable) quantum registers into a classical variable *(pseudo measurements)*.

The QCL operator below implements the selection operator

$$\mathtt{mux} : |k\rangle_{\mathbf{q}}|i\rangle_{\mathbf{s}} \to \Big(\mathtt{Not}(\mathbf{q}_i)|k\rangle_{\mathbf{q}}\Big)\,|i\rangle_{\mathbf{s}}. \tag{2.97}$$

by conditional composition (2.87).

```
cond qufunct mux(qureg q,quconst s) {  // Quantum Multiplexer
  int i;
  int n = 0;
  for i=0 to #s-1 {             // accumulate content of
    if s[i] { n=n+2^i; }        //   selection register in a
  }                             //   classical variable
  Not(q[n]);                    // flip selected output qubit
}
```

Figure 2.13 shows the quantum circuit generated by `mux(s,q)` in the case of a 2-qubit selection register `s`.

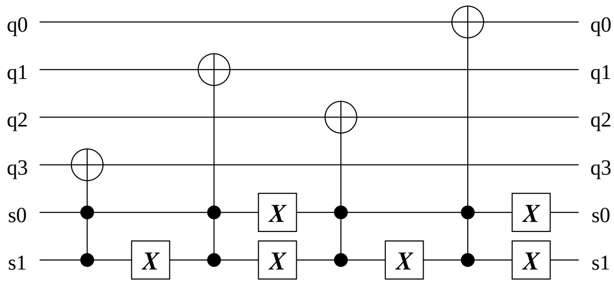


Figure 2.13: *A quantum multiplexer*

2.6.2.6 Conditional Loops

By conditionally executing a `break` statement, QISs can be used to implement (bounded) conditional loops with quantum termination conditions.

The QCL operator below implements the phase transformation

$$\texttt{pdlog}(\varphi) = \mathrm{diag}(1, \delta, \delta^2, \delta^2, \delta^3, \delta^3, \delta^3, \delta^3, \delta^4 \ldots) \quad \text{with} \quad \delta = \mathrm{e}^{\mathrm{i}\varphi} \tag{2.98}$$

which rotates basis vectors according to the highest set qubit.

```
operator pdlog(real phi,qureg q) {
  int i;
  for i=#q-1 to 0 step -1 {     // iterate from MSB to LSB
    if q[i] { break; }          //   exit if qubit is set
    Phase(-phi);                //   rotate by -phi
  }
}
```

Figure 2.14 shows the 4-qubit quantum circuit generated by `pdlog`.

2.6.3 Quantum Conditions

Definition 42 (Boolean Formulas) *Let S be a set of symbols, then $B_0(S) = S \cup \{\texttt{true}, \texttt{false}\}$ is the set of atomic boolean formulas on S. The set $B(S)$ of boolean formulas on S is recursively constructed in the following way*

(i) *if $a \in B_0(S)$ then $a, inB(S)$*

(ii) *if $a \in B(S)$ then $\neg a \in B(S)$*

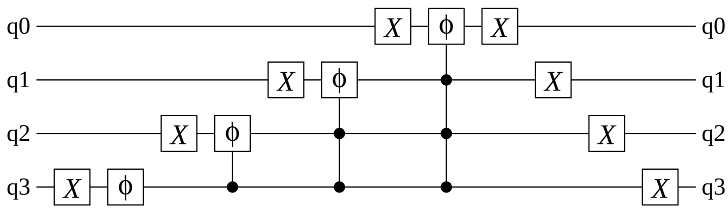


Figure 2.14: *Dual log phase transform*

(iii) *if* $a, b \in B(S)$ *then* $a \wedge b, a \vee b, a \oplus b \in B(S)$

Definition 43 (Quantum Condition) *A boolean formula* $\mathbf{C} \in B(R_1)$ *of qubits is called quantum condition. A quantum register* $\mathbf{e} = \mathbf{e}_0 \circ \ldots \circ \mathbf{e}_{n-1} \in R_n$ *corresponds to the quantum condition* $\mathbf{e}_0 \wedge \ldots \wedge \mathbf{e}_{n-1}$*. Two registers* $\mathbf{p}$ *and* $\mathbf{q}$ *are equivalent (*$\mathbf{p} \equiv \mathbf{q}$*) if they correspond to the same quantum condition (i.e. consist of the same qubits)*

2.6.3.1 Exclusive Disjunctive Normal Form

Definition 44 *Let* S *be a set of symbols. A boolean formula* $f \in B(S)$ *of the form*

$$f = \bigoplus_{i=1}^{n} \bigwedge_{j=1}^{m_i} p_{ij} \quad \text{with} \quad p_{ij} \in S \quad \text{and} \quad \{p_{kj}\} = \{p_{lj}\} \Leftrightarrow k = l \tag{2.99}$$

is in exclusive disjunctive normal form (XNF). Also we declare

$$\texttt{false} \equiv \bigoplus_{i=1}^{0} \quad \text{and} \quad \texttt{true} \equiv \bigwedge_{j=1}^{0} . \tag{2.100}$$

Any boolean formula f can be transformed into the XFN by recursively applying the following rules.

(i) $a \oplus a \to \texttt{false}$

(ii) $a \wedge a \to a$

(iii) $\neg a \to \texttt{true} \oplus a$

(iv) $a \vee b \to a \oplus a \wedge b \oplus b$

(v) $a \wedge \bigoplus_i b_i \to \bigoplus_i a \wedge b_i$

From now on, we will assume that quantum conditions are provided in XNF. Further, we will use the notation

$$\mathbf{C} = \{\mathbf{p}_i\} \iff \mathbf{C} = \bigoplus_i \bigwedge_{j=0}^{|\mathbf{p}_i|-1} \mathbf{p}_{ij}. \tag{2.101}$$

Definition 45 *Let* $\mathbf{p}, \mathbf{q} \in R$ *be registers and* $P = \{\mathbf{p}_i\}, Q = \{\mathbf{q}_i\}$ *be the set of qubits in* $\mathbf{p}, \mathbf{q}$*. The registers* $\mathbf{p} \cup \mathbf{q}$ *and* $\mathbf{p} \cap \mathbf{q}$ *are defined as*

$$\mathbf{p} \cup \mathbf{q} \quad = \mathbf{r}_0 \circ \ldots \circ \mathbf{r}_{k-1}, \quad \mathbf{r}_i \in P \cup Q, \;\; \mathbf{r}_i < \mathbf{r}_j \Leftrightarrow i < j \tag{2.102}$$

$$\mathbf{p} \cap \mathbf{q} \quad = \mathbf{s}_0 \circ \ldots \circ \mathbf{s}_{l-1}, \quad \mathbf{r}_i \in P \cap Q, \;\; \mathbf{r}_i < \mathbf{r}_j \Leftrightarrow i < j \tag{2.103}$$

Definition 46 *Let* $\mathbf{C} = \{\mathbf{p}_i\}$ *be a quantum condition. The register*

$$\mathbf{c} = \operatorname{reg} \mathbf{C} = \bigcup_i \mathbf{p}_i \tag{2.104}$$

is called condition register of $\mathbf{C}$*.*

2.6.3.2 Quantum Predicates

Definition 47 *Let* $\mathbf{C} = \{\mathbf{p}_i\}$ *be a quantum condition with the condition register* $\mathbf{c} = \operatorname{reg} \mathbf{C}$ *and* $\mathbf{t} \in R_1$*. The operator*

$$P_{\mathbf{C}}(\mathbf{c}, \mathbf{t}) = X_{[[\mathbf{C}]]}(\mathbf{t}) = \prod_i \mathtt{CNot}(\mathbf{t}, \mathbf{p}_i) \tag{2.105}$$

is called quantum predicate of $\mathbf{C}$ *with the target register* $\mathbf{t}$*.*

Definition 48 *Let* $C = \{\mathbf{p}_i\}$ *be a quantum condition with the condition register* $\mathbf{c} = \operatorname{reg} \mathbf{C}$*,* $c : \mathbf{B}^{|\mathbf{c}|} \to \mathbf{B}$ *such that* $P_{\mathbf{C}}|x\rangle_{\mathbf{c}}|0\rangle = |x\rangle_{\mathbf{c}}|c(x)\rangle$ *and* U *be a unitary operator. The (extended) conditional operator* $U_{[[\mathbf{C}]]}$ *is defined as*

$$U_{[[\mathbf{C}]]} : |k\rangle|x\rangle_{\mathbf{c}} \to \left\{ \begin{array}{cl} (U\,|k\rangle)|x\rangle_{\mathbf{c}} & \text{if } c(k) \\ |k\rangle|x\rangle_{\mathbf{c}} & \text{if } \neg c(k) \end{array} \right. . \tag{2.106}$$

By using an empty scratch qubit $\mathbf{s}$, $U_{[[\mathbf{C}]]}$ can be implemented as

$$U_{[[\mathbf{C}]]} = P_{\mathbf{C}}(\mathbf{c}, \mathbf{s})\, U_{[[\mathbf{s}]]}\, P_{\mathbf{C}}(\mathbf{c}, \mathbf{s}) \tag{2.107}$$

Boolean Operations Let $\mathbf{p}, \mathbf{q}$ be qubits, $\mathbf{s}$ be an empty scratch qubit and U a unitary operator. The extended conditional operators for the boolean operations `not`, `and`, `xor` and `or` can be implemented as

$$U_{[[\neg\mathbf{p}]]} \;=\; X(\mathbf{p})\, U_{[[\mathbf{p}]]}\, X(\mathbf{p}) \tag{2.108}$$

$$U_{[[\mathbf{p}\wedge\mathbf{q}]]} \;=\; U_{[[\mathbf{p}\circ\mathbf{q}]]} \tag{2.109}$$

$$U_{[[\mathbf{p}\oplus\mathbf{q}]]} \;=\; X_{[[\mathbf{p}]]}(\mathbf{s})\, X_{[[\mathbf{q}]]}(\mathbf{s})\, U_{[[\mathbf{s}]]}\, X_{[[\mathbf{q}]]}(\mathbf{s})\, X_{[[\mathbf{p}]]}(\mathbf{s}) \tag{2.110}$$

$$U_{[[\mathbf{p}\vee\mathbf{q}]]} \;=\; X_{[[\mathbf{p}\circ\mathbf{q}]]}(\mathbf{s})\, U_{[[\mathbf{p}\oplus\mathbf{q}]]}\, X_{[[\mathbf{p}\circ\mathbf{q}]]}(\mathbf{s}) \tag{2.111}$$

2.6.3.3 Language Representation of Quantum Conditions

In structured quantum programming, quantum conditions are represented as a special datatype (QCL type `qucond`) that represents a list of constant quantum registers. Boolean operators can be used to combine registers, quantum conditions and even classical boolean expressions:

```
qcl> qureg a[1]; qureg b[1];          // allocate 2 qubits
qcl> print a and b, a or b, a xor b;  // basic boolean operators
: <0,1> <0; 1; 0,1> <0; 1>
qcl> qucond c;                        // qucond variable
qcl> c=not (a or b);                  // qucond assignment
qcl> print c, #c, c[3];               // print c, the number of
: <*; 0; 1; 0,1> 4 <0,1>              //   clauses, the last register
qcl> print c xor true, c and (1==2);  // mixed quantum/boolean
: <0; 1; 0,1> <>                      //   expressions
qcl> c=(pi > 3);                      // boolean expressions get
qcl> print c;                         //   cast to qucond if
: <*>                                 //   necessary
```

QCL even defines a comparison operator to compare registers to other registers or integers.

```
qcl> qureg q[4];
qcl> print q==15,q==7;
: <0,1,2,3> <0,1,2; 0,1,2,3>
```

2.6.3.4 Quantum Conditions and If-Statements

The main use of quantum conditions is as arguments to quantum if-statements.

`if` $\mathbf{C}$ `then` $\sigma_1, \sigma_2 \ldots \sigma_n$ `else` $\tau_1, \tau_2 \ldots \tau_m$ `endif`

Depending on $\mathbf{C}$, there are 5 possible cases:

1. If $\mathbf{C} = \{\} =$ `false`, the else-branch is executed.
2. If $\mathbf{C} = \{\mathbf{o}\} =$ `true`, the if-branch is executed.
3. If $\mathbf{C} = \{\mathbf{p}\}$ and $|\mathbf{p}| = 1$ then the QIS is executed normally.
4. If $\mathbf{C} = \{\mathbf{q}\}$ and $|\mathbf{q}| > 1$ and the QIS is simple, then the QIS is executed normally.
5. Otherwise,

   (a) an empty scratch qubit $\mathbf{s}$ is allocated,

(b) the quantum predicate $P_{\mathbf{C}}(\mathbf{c}, \mathbf{s})$ is applied to $\mathbf{s}$,

(c) $\mathbf{C}$ is substituted by $\mathbf{s}$ and the QIS is executed,

(d) $\mathbf{s}$ is uncomputed by $P_{\mathbf{C}}(\mathbf{c}, \mathbf{s})$ and deallocated.

```
qcl> qureg q[3];
qcl> qureg a[1]; qureg b[1];
qcl> H(a & b);
[5/32] 0.5 |0,0,0> + 0.5 |0,1,0> + 0.5 |0,0,1> + 0.5 |0,1,1>
qcl> if a { inc(q); }
[5/32] 0.5 |0,0,0> + 0.5 |1,1,0> + 0.5 |0,0,1> + 0.5 |1,1,1>
qcl> if a and b { inc(q); }
[5/32] 0.5 |0,0,0> + 0.5 |1,1,0> + 0.5 |0,0,1> + 0.5 |2,1,1>
qcl> if a or b { inc(q); }
[5/32] 0.5 |0,0,0> + 0.5 |2,1,0> + 0.5 |1,0,1> + 0.5 |3,1,1>
qcl> if not a or b { inc(q); }
[5/32] 0.5 |1,0,0> + 0.5 |2,1,0> + 0.5 |2,0,1> + 0.5 |4,1,1>
```

2.6.3.5 Quantum Condition Functions

Auxiliary functions can be used to compute complex conditions. The following function computes a quantum condition to test for primality:

```
qucond isprime(quconst q) {   // Primality test for register q
  int i;
  qucond c;                   // qucond variable c={}=false
  for i = 0 to 2^#q-1 {       // iterate of possible numbers
    if testprime(i) {         // if prime then add qucond(q==i)
      c = c or q==i;          // to c
    }
  }
  return c;
}
```

Fig. 2.15 shows the quantum circuit of the quantum predicate $P_{\texttt{isprime}}(\mathbf{q}, \mathbf{p})$ for $|\mathbf{q}| = 4$.

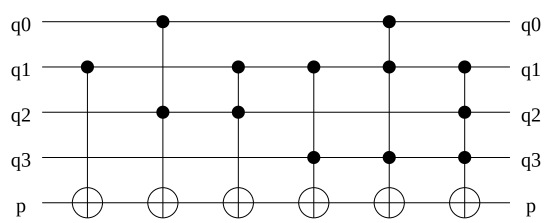


Figure 2.15: *4-qubit XNF primality test*

```
qcl> qureg q[4];
qcl> H(q);
[4/32] 0.25 |0> + 0.25 |1> + 0.25 |2> + 0.25 |3> + 0.25 |4> +
  0.25 |5> + 0.25 |6> + 0.25 |7> + 0.25 |8> + 0.25 |9> + 0.25 |10> +
  0.25 |11> + 0.25 |12> + 0.25 |13> + 0.25 |14> + 0.25 |15>
qcl> if isprime(q) { Phase(pi); }       // swap sign if prime
[4/32] 0.25 |0> + 0.25 |1> - 0.25 |2> - 0.25 |3> + 0.25 |4> -
  0.25 |5> + 0.25 |6> - 0.25 |7> + 0.25 |8> + 0.25 |9> + 0.25 |10> -
  0.25 |11> + 0.25 |12> - 0.25 |13> + 0.25 |14> + 0.25 |15>
```

Appendix A

QCL Quick Reference

A.1 Syntax

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

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

Within syntax definitions, the following conventions apply

- **Keywords** and other literal text is set in `courier`
- **Subexpressions** are set in *italic*
- **Optional** expressions are put in [square brackets]. 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 $alt_1 | \, alt_2 | \ldots$ 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* ← decimal digit from `0` to `9`.
- *letter* ← alphabetic letter form `a` to `z` or `A` to `Z`. Case is significant.
- *char* ← printable character except ‘`"`’.

A QCL Program is a sequence of *statements* and *definitions* so

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

Comments Like 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.

A.1.1 Expressions

$$
\begin{array}{rcl}
\textit{complex-coord} & \leftarrow & [\, \texttt{+} \,|\, \texttt{-} \,]\ \textit{digit}\ \{\, \textit{digit} \,\}\ [\ \texttt{.}\ \{\, \textit{digit} \,\}] \\
\textit{const} & \leftarrow & \textit{digit}\ \{\, \textit{digit} \,\}\ [\ \texttt{.}\ \{\, \textit{digit} \,\}] \\
 & \leftarrow & \texttt{(}\ \textit{complex-coord}\ \texttt{,}\ \textit{complex-coord}\ \texttt{)} \\
 & \leftarrow & \texttt{true} \,|\, \texttt{false} \\
 & \leftarrow & \texttt{"}\ \{\, \textit{char} \,\}\ \texttt{"} \\
\textit{subscript} & \leftarrow & \textit{identifier}\ \texttt{[}\ \textit{expr}\ \{\ \texttt{,}\ \textit{expr} \,\}\ \texttt{]} \\
\textit{expr} & \leftarrow & \textit{const} \,|\, \textit{subscript} \\
 & \leftarrow & \textit{identifier}\ [\ \texttt{[}\ \textit{expr}\ [(\ \texttt{:} \,|\, \texttt{..}\ )\ \textit{expr}]\ \texttt{]}\ ] \\
 & \leftarrow & \textit{identifier}\ \texttt{(}\ [\, \textit{expr}\ \{\ \texttt{,}\ \textit{expr} \,\}]\ \texttt{)} \\
 & \leftarrow & \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & \leftarrow & \texttt{\#}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{\^{}}\ \textit{expr} \\
 & \leftarrow & \texttt{-}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ (\ \texttt{*} \,|\, \texttt{/}\ )\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{mod}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ (\ \texttt{+} \,|\, \texttt{-} \,|\, \texttt{\&}\ )\ \textit{expr}
\end{array}
$$

$$
\begin{array}{rcl}
 & \leftarrow & \mathit{expr}\,(\,\texttt{==}\mid\texttt{!=}\mid\texttt{<}\mid\texttt{<=}\mid\texttt{>}\mid\texttt{>=}\,)\,\mathit{expr} \\
 & \leftarrow & \texttt{not}\ \mathit{expr} \\
 & \leftarrow & \mathit{expr}\ \texttt{and}\ \mathit{expr} \\
 & \leftarrow & \mathit{expr}\,(\,\texttt{or}\mid\texttt{xor}\,)\,\mathit{expr}
\end{array}
$$

A.1.2 Statements

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

A.1.3 Definitions

$$
\begin{array}{rcl}
\mathit{scalartype} & \leftarrow & \texttt{int}\mid\texttt{real}\mid\texttt{complex} \\
\mathit{quantumtype} & \leftarrow & \texttt{qureg}\mid\texttt{quvoid}\mid\texttt{quconst}\mid\texttt{quscratch} \\
\mathit{tensortype} & \leftarrow & (\,\texttt{vector}\mid\texttt{matrix}\mid\texttt{tensor}\ \mathit{digit}\,)\ \mathit{scalartype}
\end{array}
$$

othertype	←	(`qucond` | `boolean` | `string`
type	←	*scalartype* | *tensortype* | *quantumtype* | `othertype`
const-def	←	`const` *identifier* `=` *expr* ;
var-def	←	*type identifier* `[` *expr* `]` ;
	←	*type identifier* [`=` *expr*] ;
arg-def	←	*type identifier*
arg-list	←	`(` [*arg-def* { `,` *arg-def* }] `)`
body	←	`{` { *const-def* | *var-def* } { *stmt* } `}`
def	←	*const-def* | *var-def*
	←	*type identifier arg-list body*
	←	`procedure` *identifier arg-list body*
	←	[`cond`] `operator` *identifier arg-list body*
	←	[`cond`] `qufunct` [`operator`] *identifier arg-list body*
	←	`extern operator` *identifier arg-list* ;
	←	`extern qufunct` *identifier arg-list* ;

A.2 Expressions

A.2.1 Data Types

A.2.1.1 Classical Scalar Types

The classical scalar data-types of QCL are the arithmetic types `int`, `real` and `complex`, as well as `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`	logical value	`true, false`
`string`	character string	`"hello world", ""`

A.2.1.2 Tensors

Since v0.5, QCL supports vectors, (square) matrices and higher tensors up to order 9.

Type	Description	Examples
`vector`	vector	`vector(0,0.5,0.866)`
`matrix`	square matrix	`matrix(0,(0,-1),(0,1),0)`
`tensor`*n*	tensor of order n	`tensor3(1,0,0,0,0,0,0,1)`

Tensors can be defined for the arithmetic scalar types `int`, `real` and `complex`. A tensor variable *v* of order *n* and dimension *dim* is declared by the syntax

```
scalartype tensorn v[dim];
```

For tensors of order 1 and 2 the keywords `vector` and `matrix` can be used.

The subscript operator `v[coord]` is used to access tensor elements with *coord* being a comma separated list of *n* zero-based integer indices.

There are no tensor literals; instead a tensor object of order *n* and dimension d is created by the constructor function `tensorn(elem)`, with *elem* being a comma separated list of d^n scalar expressions ordered in ascending order of their indices with the leftmost index being the most significant.

Tensors of equal order and dimension can be added, subtracted and assigned to. Tensors can be multiplied by scalars or by tensors of equal dimension. In the latter case, multiplication is defined as generalized dot-product, i.e. contraction by summing over the innermost indices

$$a_{i_1 \ldots i_{n-1} i_n} * b_{j_1 j_2 \ldots j_m} = \sum_k a_{i_1 \ldots i_{n-1} k} b_{k j_2 \ldots j_m}$$

A.2.1.3 Register Types

For local variables or parameters of the types `quvoid` and `quscratch` let U be the corresponding unitary operator.

Type	Function	Restriction
`qureg`	general register	none
`quconst`	quantum constant	invariant to all operators
`quvoid`	target register	empty when U is called
`quscratch`	scratch register	empty when U or $U^\dagger$ are called

A.2.1.4 Quantum Conditions

Boolean expressions of qubits are represented by the QCL type `qucond`. Let *C* be a variable of type `qucond` and

$$\mathbf{C} = \{\mathbf{p}_i\} = \bigoplus_{i=0}^{n-1} \bigwedge_{j=0}^{|\mathbf{p}_i|-1} \mathbf{p}_{ij}$$

the XNF of `C`. The size operator `#C` returns the number n of clauses in $\mathbf{C}$ and `C[k]` gives the `quconst` register $\mathbf{p}_k$.

A.2.2 Operators

The following table lists all QCL operators ordered by precedence. *`arith`* stands for all arithmetic scalar types, *`tensor`* for all tensor types and *`quantum`* for all register types.

Op	Description	Argument type
`[ ]`	qubit subregister	*`quantum`* variable
	XNF clause	`qucond` variable
	`vector` subscript	*`vector`* variable
`[ .. ]`	subregister by range	*`quantum`* variable
`[ :: ]`	subregister by offset and length	*`quantum`* variable
`[ , ]`	tensor subscript	*`tensor`* variable
`#`	register size	*`quantum`*
	number of XNF clauses	`qucond`
	dimension	*`tensor`*
`^`	power	*`arith`*
	integer power	`int`
`-`	unary minus	*`arith`*, *`tensor`*
`*`	multiplication	*`arith`*, *`tensor`*
`/`	division	*`arith`*
	integer division	`int`
`mod`	integer modulus	`int`
`+`	addition	*`arith`*, *`tensor`*
`-`	subtraction	*`arith`*, *`tensor`*
`&`	concatenation	`string`, *`quantum`*
`==`	equal	*`arith`*, `string`
`!=`	unequal	*`arith`*, `string`
`<`	less	`int`, `real`
`<=`	less or equal	`int`, `real`
`>`	greater	`int`, `real`
`>=`	greater or equal	`int`, `real`
`not`	logical not	`boolean`, `qucond`
`and`	logical and	`boolean`, `qucond`
`or`	logical inclusive or	`boolean`, `qucond`
`xor`	logical exclusive or	`boolean`, `qucond`

A.2.3 Elementary Functions

Elementary functions are part of QCL and need not be declared. Unlike function subroutines, elementary functions

- do not need to have a fixed number of arguments
- may take arguments of different types and
- may have varying return types depending on the types of the arguments

Function	**Description**	**Arg.**
`sin`, `cos`, `tan`, `cot`	trigonometric functions	*arith*
`sinh`, `cosh`, `tanh`, `coth`	hyperbolic functions	*arith*
`exp(x)`	e raised to the power of x	*arith*
`log(x)`	natural logarithm of x	*arith*
`log(x,n)`	base-n logarithm of x	*arith*
`sqrt(x)`	square root of x	*arith*
`abs(x)`	absolute value of x	*arith*
`Re`, `Im`	real and imaginary part	`complex`
`conj(z)`	complex conjugate of z	`complex`
`floor`, `ceil`	next higher and lower integer	`real`
`gcd(n,...)`	greatest common divisor	`int`
`lcm(n,...)`	least common multiple	`int`
`min`, `max`	minimum and maximum	*arith*
`not`, `and`, `or`, `xor`	binary functions	`int`
`bit(n,k)`	logical value of the k^{th} bit of n	`int`
`int`, `real`, `complex`, `string`	explicit typecasts	*scalar*
`vector`, `matrix`, `tensor`*n*	tensor constructors	*arith*
`random()`	random value from $[0,1)$	*none*

A.3 Statements

Unless otherwise stated, parameters in *`slanted courier`* denote expressions (see A.1.1).

A.3.1 Simple Statements

A.3.1.1 Assignments

`lvalue` `=` *`rvalue`*`;`

`lvalue` can either be a variable or a *subscript* expression. Quantum variables (i.e. symbolic registers) cannot be assigned to.

Implicit typecasting is performed from `int` and `real` to `real` or `complex` and from `boolean` and all register types to `qucond`. In all other cases, `lvalue` and `rvalue` have to be of the same type.

A.3.1.2 Subroutine Calls

[!]`sub(args);`

`args` is a comma separated list of arguments and can be empty. The number of expressions in `args` has to match the declaration of the subroutine `sub` which can be a `procedure`, `operator` or `qufunct`.

If `sub` is a quantum subroutine, then the inverse operator can be called by prefixing the name with "`!`". Procedures cannot be inverted.

A.3.1.3 Input and Output

`input` [`prompt ,`] `var;`

reads classical input from the user as assigns it to the scalar variable `var`. An optional `prompt` of type `string` can be specified.

`print list;`

prints a comma separated `list` of expressions.

A.3.1.4 Measurement and Initialization

`measure q`[`,var`]`;`

measures the register `q` and assigns the measured value to the integer variable `var` if specified.

`reset;`

initializes the machine state.

A.3.2 Flow Control

A.3.2.1 Loops

The body of a loop is a list of statements. Braces are obligatory even if `body` is a single statement. The `break` statement

`break;`

can be used within the body to immediately exit the innermost loop.

Conditional Loops

```
while cond { body }
{ body } until cond;
```

cond is a boolean expression. The body of an `until`-loop is executed at least once.

Counting Loops

```
for i = a to b [ step s ] { body }
```

The range *a*, *b* and the optional step size *s* are integer expressions. The loop variable *i* is also of type `int` and is redeclared as symbolic constant within the body.

A.3.2.2 If-statement

```
if cond { sigma } [ else { tau }]
```

The if-branch *sigma* and the optional else-branch *tau* are list of statements; braces are obligatory even for single statements.

The if-condition *cond* is either a boolean expression *(classical if-statement)* or a quantum condition *(quantum if-statement)*.

A.3.2.3 Abnormal Termination

```
exit msg;
```

prints out the error message *msg* and terminates the QCL program.

A.3.3 Interactive Commands

The QCL interpreter `qcl` defines additional commands which are mainly intended for interactive use and debugging and not considered to be core-parts of QCL.

A.3.3.1 Simulation Commands

The following commands are only available if QCL is used together with a numerical simulator.

```
dump [ q ];
```

prints the probability spectrum of the register *q*. If no register is specified, the current machine state is printed.

```
plot [ q ];
```

plots the probability spectrum of the register *q*. If no register is specified, the current machine state is plotted.

```
plot q, p ;
```

plots the probability spectrum of $q \circ p$ as two-dimensional density graph.

```
save file;
```

saves the current machine state to *file*.

```
load file;
```

loads the current machine state from *file*.

A.3.3.2 Other Commands

```
set opt val;
```

sets the interpreter option *opt* (without the leading "--") to the value *val*. Please refer to appendix A.4 for a complete list of options.

```
list [sym];
```

lists the definition of the symbol *sym* in the current scope. If no symbol is specified, all currently defined symbols are listed.

```
shell;
```

opens a subshell which can be closed again with the `exit` command. Subshells can also be opened in response to keyboard interrupts and runtime errors (see A.4). Symbols defined in a subshell are local and leave scope when the shell is left again.

```
exit;
```

closes the current shell. Closing the top-level shell terminates the session.

A.4 Interpreter Options

The QCL interpreter `qcl` has the following options:

Startup Options Those can only be set on the command line.

-h, --help	*display this message*
-V, --version	*display version information*
-i, --interactive	*force interactive mode*
-x, --exec<commands>	*execute ¡commands¿ on startup*
-q, --quiet	*skip startup message*
--color	*color xterm interface*
--texmacs	*TeXmacs interface (experimental)*
-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 Those can be set on the command line or via the `set` command (see above). Default values are given in brackets.

-s, --seed=<seed-value>	*set random seed value* (*system time*)
-I, --include-path=<path>	*QCL include path* (/usr/local/lib/qcl)
--library=<y|n>	*ignore redefinitions of existing symbols* (n)
-d, --dump-file=<file>	*send output of dump-command to file* (*none*)
-p, --plot-file=<file>	*Postscript file created by plot-command* (*none*)
-f, --dump-format=x,d,b	*list base vectors as hex, decimal or binary* (d)
-r, --show-regs=<y|n>	*show global registers in dumped states* (y)
-D, --dump-precision=<d>	*shown d digits in dumped states* (5)
-P, --precision=<digits>	*shown digits for real and complex values* (6)
-Z, --trunc-zeros=<y|n>	*truncate zeros for real and complex values* (y)
-T, --truc-states=<y|n>	*truncate non-allocated qubits* (y)
--plot-paper=<format>	*Set paper-format for Postscript output* (b5)
--plot-size=<pixel>	*Set maximum window size for X11 plots* (600)
-Q, --qureg-mask=<y|n>	*list registers as masks instead of lists* (n)
-g, --debug=<y|n>	*open debug-shell on error* (n)
-a, --auto-dump=<max>	*dump states up to max terms in shell mode* (8)
-l, --log==<y|n>	*log external operator calls* (n)
-L, --log-state==<y|n>	*log state after each transformation* (n)
-c, --check==<y|n>	*check consistency of quantum heap* (n)
--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 program, ignore quantum operations* (n)
-e, --shell-escape=<y|n>	*honor shell-escapes* (y)
--irq=<y|n>	*allow user interrupts if supported* (y)

Bibliography

[1] Anthony A. Aaby: Introduction to Programming Languages (1996). `http://cs.wwc.edu/~aabyan/221_2/PLBOOK/`

[2] G. Baker: Qgol (1996). *project-thesis, Macquarie University,* `http://www.ifost.org.au/~gregb/q-gol/`

[3] A. Barenco et al.: Elementary Gates for Quantum Computation (1995). *Phys. Rev. A 52, pp. 3457-3467*

[4] D. Beckman et al.: Efficient networks for quantum factoring (1996). *Phys. Rev. A 54, pp. 1034-1063*

[5] P.A. Benioff: The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines (1980). *J. Stat. Phys., 22(5), pp. 563-591*

[6] P.A. Benioff: Models of Quantum Turing Machines (1997). *LANL* `quant-ph/9708054`

[7] C.H. Bennet: Logical Reversibility of Computation (1973). *IBM J. Res. Develop. 17, 525*

[8] C.H. Bennet: Time-space trade-offs for reversible computation (1989). *SIAM J. Comput. 18, p. 766*

[9] E. Bernstein, U. Vazirani: Quantum complexity theory (1997). *SIAM J. Comput., 26(5), pp. 1411-1473*

[10] S. Blaha: Quantum Computers and Quantum Computer Languages: Quantum Assembly Language and Quantum C Language (2002). *LANL* `quant-ph/0201082`

[11] C. Böhm and G. Jacopini: Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules (1966). *Communications of the ACM, Vol. 9, No. 5, pp. 336-371*

[12] M. Boyer, G. Brassard, P. Hoyer, A. Tapp: Tight bounds on quantum searching (1998). *Fortschritte der Physik, vol. 46(4-5), pp. 493-505*

[13] P.O, Boykin et al.: On universal and fault-tolerant quantum computing (1999). *LANL* `quant-ph/9906054`

[14] J. Branson: Quantum Physics 130A (2001). *lecture notes,* `http://heppc16.ucsd.edu/ph130a/130a_notes/node15.html`

[15] J. Buchmann: Faktorisierung großer Zahlen (1996). *Spektrum der Wissenschaft 9/96, pp. 80-88*

[16] S.L. Braunstein: Quantum computation: a tutorial (1995). `http://www.informatics.bangor.ac.uk/~schmuel/comp/comp.html`

[17] D.C. Cassidy: Exhibit on Werner Heisenberg (1998). *American Institute of Physics* `http://www.aip.org/history/heisenberg/p09_text.htm`

[18] I.L. Chuang et al.: NMR quantum computing: Realizing Shor's algorithm (2001). *Nature 414, pp. 883-887*

[19] J.I. Cirac, P. Zoller: Quantum Computations with Cold trapped Ions (1995). *Phys. Rev. Lett. 74, p. 4091*

[20] R. Cleve, A.Ekert, C. Macchiavello, M. Mosca: Quantum algorithms revisited (1998). *Proc. R. Soc., London A 454, pp. 339-354*

[21] B.J. Copeland: The Church-Turing Thesis (1996). *Stanford Encyclopedia of Philosophy ISSN, pp. 1095-5054*

[22] D. Coppersmith: An Approximate Fourier Transform Useful in Quantum Factoring (1994). *IBM Research Report No. RC19642*

[23] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: Structured Programming (1972). *Academic Press, London, UK*

[24] D. Deutsch,: Quantum theory, the Church-Turing principle and the universal quantum computer (1985). *Proc. R. Soc., London, A 400, pp. 97-117*

[25] D. Deutsch: Quantum computational networks (1989). *Proc. R. Soc., London, A 439, pp. 553-558*

[26] D. Deutsch and R. Jozsa: Rapid solution of problems by quantum computer (1992). *Proc. Roy. Soc. London, Ser. A, vol. 439, pp. 553-558*

[27] E.W. Dijkstra: Structured Programming (1969). *Software Engineering Techniques, Buxton, J. N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee*

[28] D.P. DiVincenzo: Two bit gates are universal for quantum computation (1995). *Phys. Rev. A, 51(2), pp. 1015-1022*

[29] A. Einstein, B. Podolsky, N. Rosen: Can quantum-mechanical description of physical reality be considered complete? (1935). *Physical Review 47, pp. 777-780*

[30] A. Ekert, R. Jozsa: Shor's Quantum Algorithm for Factoring Numbers (1996). *Rev. Modern Physics 68 (3), pp. 733-753*

[31] R.P. Feynman: Quantum mechanical computers (1985). *Optics News 11, pp 11-20*

[32] Lov K. Grover: A fast quantum mechanical algorithm for database search (1996). *Proceeding of the 28th Annual ACM Symposium on Theory of Computing*

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

[34] G.H. Hardy, E.M. Wright: An Introduction to the Theory of Numbers Oxford (1938). *U.K.: Oxford Univ. Press*

[35] A. Hermann: Lexikon - Geschichte der Physik A-Z (1978). *Aulis-Verlag Deubner & Co KG*

[36] R.W. Keyes: Miniaturization of electronics and its limits (1988). *IBM J. Res. Develop. 32(1), pp. 24-28*

[37] D. Kielpinski, C. Monroe and D.J. Wineland: Architecture for a large-scale ion-trap quantum computer (2002). *Nature 417, pp. 709-711*

[38] W. Kummer and R. Trausmuth: Quantentheorie (1988). *Skriptum zur Vorlesung 131.869*

[39] A. Leitsch: Skriptum zur Vorlesung "Algorithmen-, Rekursions und Komplexitätstheorie" (1994). *Technical University Vienna*

[40] M. Agrawal, N. Kayal, N. Saxena: PRIMES is in P (2002). *preprint,* `http://www.cse.iitk.ac.in/users/manindra/primality.ps`

[41] Z. Meglicki: Introduction to Quantum Computing (M743) (2002). `http://beige.ucs.indiana.edu/B679/M743.html`

[42] F.D. Murnaghan: The Unitary and Rotation Groups (1962). *Spartan Books, Washington*

[43] M.A. Nielsen and I.L. Chuang: Quantum Computation and Quantum Information (2000). *Cambridge University Press*

[44] B. Ömer: Simulation of Quantum Computers (1996). `http://www.itp.tuwien.ac.at/~oemer/papers.html`

[45] B. Ömer: A Procedural Formalism for Quantum Computing (1998). *master-thesis, Technical University of Vienna*

[46] B. Ömer: Quantum Programming in QCL (2000). *master-thesis, Technical University of Vienna*

[47] B. Ömer: Procedural Quantum Programming (2002). *Proc. of the 5th Int. Conference CASYS 2001, D. M. Dubois (Ed.), AIP Conference Proceedings 627, pp. 276-285*

[48] B. Ömer: Classical Concepts in Quantum Programming (2003). *accepted for publication in the QS2002 Conference Proceedings, LANL* `quant-ph/0211100`

[49] B. Ömer: QCL homepage (2003). `http://www.itp.tuwien.ac.at/~oemer/qcl.html`

[50] E.L. Post: Finite Combinatory Processes - Formulation 1 (1936). *Journal of Symbolic Logic, 1, pp. 103-105*

[51] H. Putnam: A philosopher looks at quantum mechanics (1965). *R. Colodny (ed.), Beyond the Edge of Certainty: Essays in Contemporary Science and Philosophy, Englewood Cliffs, New Jersey: Prentice Hall, pp. 130-158.*

[52] M. Reck, A. Zeilinger, H.J. Bernstein, P. Bertani: Experimental realization of any discrete unitary operator (1994). *Phys. Rev. Lett., 73(1), pp. 58-61*

[53] M.A. Rowe et al.: Transport of quantum states and separation of ions in a dual rf ion trap (2002). *Quantum Information and Computation 2, pp. 257-271*

[54] P.W. Shor.: Algorithms for quantum computation: Discrete logarithms and factoring (1994). *Proc. 35th Annual Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA*

[55] R. Solovay and V. Strassen: A Fast Monte-Carlo Test for Primality (1977). *SIAM Journal on Computing, 1977, pp. 84-85*

[56] K. Svozil: Quantum algorithmic information theory (1996). *Journal of Universal Computer Science 2, pp. 311-346*

[57] T. Toffoli: Bicontinuous extension of reversible combinatorial functions (1981). *Math. Syst. Theory 14, pp. 13-23*

[58] A.M. Turing: On computable numbers, with an application to the Entscheidungsproblem (1936). *Proc. of the London Mathematical Society, ser. 2. vol. 42, pp. 230-265,*
`http://www.abelard.org/turpap2/tp2-ie.asp`

[59] A.M. Turing: Intelligent Machinery (1948). *National Physical Laboratory Report. In Meltzer, B., Michie, D. (eds) 1969. Machine Intelligence 5. Edinburgh: Edinburgh University Press., 7*

[60] B. VanHoy: Structured Programming, Control Structures, if-else Statements, Pseudocode (1998). *APCS-C++, Lesson 8, ICT*
`http://www.mvhs.fuhsd.org/bob_vanhoy/pdfs/lesson08.pdf`

[61] J. Wallace: Quantum Computer Simulators (2001). *Partial Proc. of the 4th Int. Conference CASYS 2000, D. M. Dubois (Ed.), International Journal of Computing Anticipatory Systems, volume 10, 2001, pp. 230-245*

[62] J. Wallace: Quantum Computer Simulators (2002). *online survey,*
`http://www.dcs.ex.ac.uk/~jwallace/simtable.htm`

[63] P. Zuliani: Quantum Programming (2001). *PhD thesis, University of Oxford,*
`http://web.comlab.ox.ac.uk/oucl/work/paolo.zuliani/pzthesis.ps.gz`

List of Figures

List of Tables