Praca doktorska

Probabilistyczne aspekty programowania kwantowego

Jarosaw Adam Miszczak

Promotor: Prof. dr hab in. Jerzy Klamka

Instytut Informatyki Teoretycznej i Stosowanej
Polskiej Akademii Nauk

Marzec 2008

Doctoral thesis

Probabilistic aspects of quantum programming

Jarosaw Adam Miszczak

Supervisor: Prof. dr hab in. Jerzy Klamka

The Institute of Theoretical and Applied Informatics
Polish Academy of Sciences

March 2008

Contents

Abstract

Quantum information theory allows us to use quantum mechanics resources for processing and sending information. Since the rules of quantum mechanics are in many aspects counterintuitive, the development of new quantum algorithms and protocols is complicated. At the moment only a few algorithms exist which can be executed on a quantum computer with significant improvement in speed or memory usage.

The main goal of this thesis is to describe the methods of using quantum programming languages for developing new quantum algorithms. We present the original implementation of a classical game where quantum programming was used to develop and analyse quantisation of the game. We also show how quantum data types, ie. data types allowing for storing superposition of values, can be used to develop new quantum algorithms.

Acknowledgements

I would like to thank my supervisor prof. Jerzy Klamka for his help during my work on this thesis.

I would like to thank prof. Vladimír Bužek from the Slovak Academy of Sciences for giving me an opportunity to work in his group and prof. Karol yczkowski from the Jagiellonian University for interesting and fruitful discussions.

I also would like to express my gratitude to Ravinda Chhajlany, Piotr Gawron, Josef Košík and Zbyszek Puchaa for interesting discussions and valuable comments. I am also grateful to Ryszard Winiarczyk for his support and guidance.

Finally, I would like to thank my wife Iza for her help, support and comments.

This work was realized under financial support of Polish Ministry of Science and Higher Education grants N519 012 31/1957 and N206 013 31/2258.

Chapter 1

Introduction

In this chapter we give short introduction to quantum information theory and provide the background for presented research. We briefly describe recent results in the field of quantum information, related to the topic of this thesis. We focus on quantum programming languages and novel approaches to quantum algorithms development. Among these we list quantum games and quantum walks. We also explain the main motivation for presented research and provide the outline of results presented in this thesis.

As we do not aim to repeat too many elementary facts the references to literature are provided. The selected mathematical tools used in quantum information theory are presented in Appendix B.

1.1 Quantum information theory

Quantum information theory is a new, fascinating field of research which aims to use quantum mechanical description of the systems to perform computational tasks. It is based on quantum physics and classical computer science, and its goal is to use the laws of quantum mechanics to develop more powerful algorithms and protocols.

According to the Moore's Law [84] the number of transistors on a given chip is doubled every two years (see Figure 1.1). Since classical computation has its natural limitations in terms of the size of computing devices, it is natural to investigate the behaviour of objects in micro scale.

Quantum effects cannot be neglected in microscale and thus they must be taken into account when designing future computers. Quantum computation aims not only take them into account, but also develop methods for controlling them. Quantum algorithms and protocols are recipes how one should control quantum system to achieve better efficiency.

Information processing on quantum computer was first mentioned in

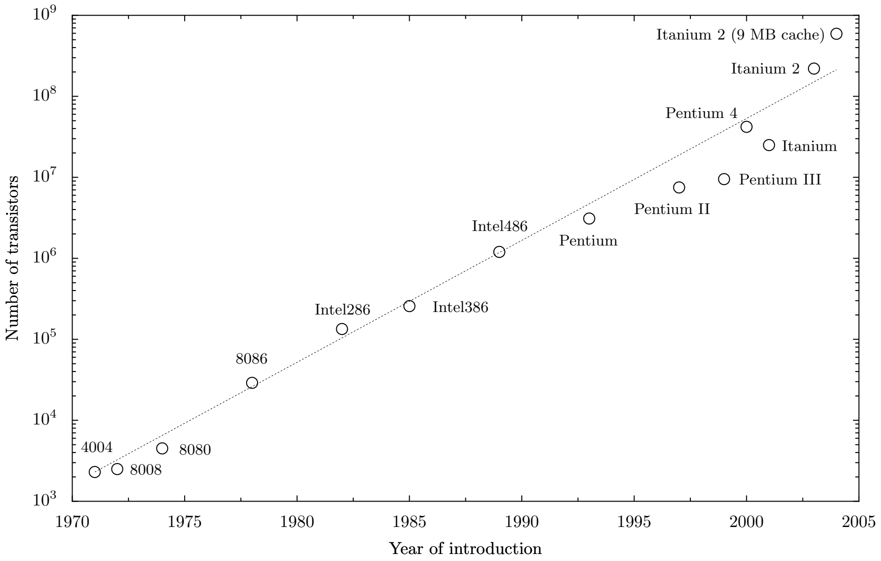


Figure 1.1: Illustration of Moore's hypothesis. Number of transistors which can be put on single chip grow expotentially. The circles represent microprocessors introduced by Intel Corportation [57]. Dashed line illustrates the rate of growth, with the number of transistors doubling every two years.

1982 by Feynman [37]. This seminal work was motivated by the fact that simulation of a quantum system on the classical machine requires exponential resources. Thus, if we could control physical system at the quantum level we should be able to simulate other quantum systems using such machines.

The first quantum protocol was proposed two years later by Bennett and Brassard [9]. It gave the first example of the new effects which can be obtained by using the rules of quantum theory for processing information. In 1991 Ekert described the protocol [35] showing the usage of quantum entanglement [32] in communication theory.

Today we know that thanks to the quantum nature of photons it is possible to create unconditionally secure communication links [16] or send information with efficiency unachievable using classical carriers. During the last years cryptographic systems have been implemented in real-world systems. Quantum key distribution is the most promising application of quantum information theory, if one takes practical applications [114] into account.

On the other hand we know that the quantum mechanical laws of nature allow us to improve the solution of some problems [107, 46], construct games [34] and random walks [66] with new properties.

Nevertheless, the most spectacular achievements in quantum information theory to the moment are the quantum algorithm for factoring numbers[1] and

[1] Detailed description of factorisation algorithm can be found in [73].

calculating discrete logarithms over finite field proposed in the late ninethies by Shor [106]. The quantum algorithm solves the factorisation problem in polynomial time, while the best known probabilistic classical algorithm runs in time exponential with respect to the size of input number. Shor's factorisation algorithm is one of the strongest arguments for the conjecture that quantum computers can be used to solve in polynomial time problems which cannot be solved classically in reasonable (ie. polynomial) time.

Concerning research efforts focused on discovering new quantum algorithms it is suprising that during the last ten years no similar results have been obtained. One should note that there is no proof that quantum computers can actually solve **NP**-complete problems in polynomial time [41]. This proof could be given by quantum algorithms solving in polynomial time problems known to be **NP**-complete such as k-colorability. The complexity of quantum computation remains poorly understood. We do not have much evidence how useful quantum computers can be. Still much is to be discovered about the relations between quantum complexity classes such as **BQP** and classical complexity classes like **NP**.

1.2 Recent results

To give the necessary background for this thesis, we review the recent results in the fields of quantum programming and quantum algorithms. We also review new methods proposed during the last few years to study and develop quantum algorthims, namely quantum games and quantum walks.

1.2.1 Progress in quantum programming

Several languages and formal models were proposed for the description of quantum computation process. The most popular of them is quantum circuit model [26], which is tightly connected to the physical operations implemented in laboratory. On the other hand the model of quatum Turing machine is used for analysing the complexity of quantum algorithms [10].

Another model used to describe quantum computers is Quantum Random Access Machine (QRAM). In this model we have strictly distinguised the quantum part performing computation and classical part, which is used to control computation. This model is used as a basis for most quantum programming languages [43]. Among high-level programming languages designed for quantum computers we can distinguish imperative and functional languages.

At the moment of writing this thesis the most advanced imperative quantum programming languages is Quantum Computation Language (QCL) designed and implemented by Ömer [93, 91, 92]. QCL is based on the syntax of the C programming language and provides many elements known from classical programming languages. The interpreter is implemented using simulation library for executing quantum programmes on classical computer, but it can be in principle used as a code generator for classical machine controlling a quantum circuit. Translation of QCL source code into Quantum Markup Language [75], used to describe quantum circuits, was presented in [123].

Along with QCL few other imperative quantum progamming languages were proposed. Notably Q Language developed by Betteli [11, 12] and libquantum [119] have the ability to simulate noisy environment. Thus, they can be used to study decoherence and analyse the impact of imperfections in quantum systems on the accuracy of quantum algorithms.

Q Language is implemented as a class library for C++ programming language and libquantum is implemented as a C programming language library. They also share some limitation with QCL, since it is possible to operate on single qubits or quantum registers (ie. arrays of qubits) only. Thus, they are similar to packages for computer algebra systems used to simulate quantum computation [77].

Concerning problems with physical implementations of quantum computers, it became clear that one needs to take quantum errors into account when modelling quantum computational process. Also quantum communication has become very promising application of quantum information theory over the last few years. Both facts are reflected in the design of new quantum programming languages.

LanQ developed by Mlnařík was defined in [83]. It provides syntax based on C programming language. LanQ provides several mechanisms such as the creation of a new process by forking and interprocess communication. Thus, it supports the implementation of multiparty protocols. Moreover, operational semantics of LanQ has been defined. Thus, it can be used for the formal reasoning about quantum algorithms.

It is also worth to mention new quantum programming languages based on functional paradigm. QPL [103] was the first functional quantum programming language. This language is staticaly typed and allows to detect errors at compile-time rather than run-time.

Its more mature version is cQPL — communication capable QPL . cQPL was created to facilitate the development of new quantum communication protocols. Its interpreter uses QCL as a backend language so the cQPL programmes are translated in C++ code using QCL simulation library.

In Chapter 3 we compare the selected quantum programming languages and descibe their main advantages and limitations.

1.2.2 Limitations of quantum programming

In QCL quantum memory can be accessed using only qreg data type, which represents the array of qubits. In the syntax of cQPL data type qint has been introduced, but it is only synonymous for the array of 16 qubits. Similar situation exists in LanQ [83], where quantum data types are introduced using q$n$it keyword, where n represents a dimension of elementary unit (eg. for qubits $n = 2$, for qutrits $n = 3$). However, only unitary evolution and measurement can be performed on variables defined using one of these types.

In this thesis quantum data type is defined as a data type which allows for superpositions of values in its range. In Chapter 5 we argue that thanks to quantum data types many common tasks can be performed using compact syntax. We also propose small programming language designed to test developed algorithms.

Experimental quantum programming language kulka introduces qint data type for storing integer numbers. It is possible to initialise variable to store superposition of integers (see Appendix A), eg.

```
qint v1 = (5—10—15);
```

Naturally, this operation must be represented using quantum gates — this is one of the main requirements for any quantum programming language. Like in the case of quantum conditions introduced in QCL [93], initialisation to superposition provides only syntactic sugar for operation of quantum memory. Nevertheless it can be also used as the basis for more elaborated quantum algorithms. The example of such algorithm is the quantum radix sorting algorithm described in Chapter 5.

1.2.3 New methods for developing quantum algorithms

Due to the lack of progress in discovering new quantum algorithms novel methods for studying impact of quantum mechanics on algorithmic problems were proposed.

The first of these methods aims at applying the rules of quantum mechanics to game theory [61]. Classical games are used to model the situation of conflict between competing agents [98]. The simplest application of quantum games is presented in the form of quantum prisoners dilemma [34]. In this case one can analyse the impact of quantum information processing on classical sceniarios. On the other hand quantum games can be also used to analyse typical quantum situations like state estimation and clonning [69]. Among interesting results of this approach we can also point out quantum Parrondo's paradox [78], which was applied to analyse quantum algorithms [68] and quantum decoherence [70].

Quantum walks provide the second promising method for developing new quantum algorithms. Quantum walks are the counterparts of classical random walks [66]. In [4] the quantum algorithm for element distinctness using this method was proposed. It requires $O(n^{2/3})$ queries to determine if the input $\{x_1, \ldots, x_n\}$ consisting of n elements contains two equal numbers. Generalization of this algorithm, with applications to the problem of subset finding, was described in [19]. In [3] the survey of quantum algorithms based on quantum walks is presented.

In this thesis we show how quantum programming language can be used to develop and analyse quantum games. The example of such application is described in Chapter 4.

1.3 Motivation and goals

At the moment only few quantum algorithms exist. Taking into account intensive work in this field it is natural to ask about the roots of this situation [108, 109].

One possible explanation of this situation is that quantum mechanical information processing can be used to solve the narrow class of problems. It is hard to find new algorithms since it is difficult to find problems which can be solved on quantum computers better than on classical ones.

The second, more optimistic, explanation lies in the fact that quantum mechanical description of the computational process differs significantly from the description we are used to in classical physics. We know that quantum mechanical objects behave in the way that is far from common intuition.

Following this argument one can conclude that it is necessary to develop new methods for dealing with quantum mechanical systems, which allow us to develop new algorithms and protocols. This was the main motivation for starting research in the field of quantum programming languages.

Any classical computation can be described using only bits and operations on them, but it is very complicated to write programmes processing only bits. Similar situation occurs in quantum computing where one needs to process not only single bits, but also the superposition of them. Of course any quantum programme written in high-level programming language needs to be decomposed into the set of elementary operations. But like in the classical case this can be done automatically using the compiler of a high-level language.

This is also the motivation for research described in this thesis. Our aim is to show that quantum programming languages and quantum data types can be used to facilitate construction of new quantum algorithms.

1.4 Thesis

The main goal of this research is to show how quantum programming allows for developing new quantum algorithms. In particular we show how it can be used to develop quantum versions of classical results. We present two applications where quantum programming languages were used for developing the generalisation of classical results.

The first example is the application of quantum computing paradigm to game theory. We present the original quantum version of Parrondo's paradoxical game [48, 49].[2] The development and analysis of the presented quantum Parrondo's game were conducted using a quantum programming language. In spite of the fact that existing quantum programming languages do not provide many high-level elements they can be used to facilitate the development and analysis of quantum algorithms.

The second application is related to experimental quantum programming language kulka designed during the work on this thesis. We present the algorithm for state initialisation used in kulka interpreter. It was used to implement quantum data type for handling integer numbers. We also present the quantum version of radix sorting algorithm. It has been developed using data types allowing for the superposition of values from their range.

By analyzing existing quantum programming languages and examples from experimental programming language we aim to prove following thesis concerning quantum programming languages and quantum algorithms:

❏ Existing quantum programming languages can be used to develop new quantum algorithms.

❏ Introduction of quantum data types provides the way for handling common computing operations, in particular primitive operations used in quantum information processing.

1.5 Organization of this thesis

This thesis is organized as follows.

In Chapter 2 we describe formal models used to describe quantum computation. This includes quantum circuits model, quantum Turing machine and quantum random access machine. In analogy to the situation in classical computer science the model based on random access machine is used as the theoretical background for programming languages.

Chapter 3 presents the state of art in the field of quantum programming languages. We describe the most advanced existing languages designed to operate on quantum memory. Some examples are provided to allow for better insight into syntax of presented languages. We argue that quantum programming is necessary for developing new quantum algorithms and protocols. We also point out the drawbacks of existing languages and why it is necessary to develop new languages.

In Chapter 4 the application of quantum programming to quantum games is discussed. We provide the example of classical game implemented in high-level quantum programming language. We show how procedural quantum programming language can be used to create and analyse the quantum version of classical game.

[2] It shuld be noted that for a given classical game there is no unique method for constructing its quantum version.

Chapter 5 contains the description of the quantum procedures developed during the work on this thesis. We present the original algorithm for preparing superposition of quantum registers, which has been used in the interpreter of high level quantum programming language. We also present the quantum algorithm for sorting, which has been developed using the methods based on quantum data types. We compare the time and space complexity of the presented algorithm with the complexity of existing algorithms. Appendix A contains the description of experimental programming language kulka which has been developed to test algorithms presented in this chapter.

Chapter 6 contains the summary, conclusion and considerations about the possible directions of further work.

Chapter 2

Models of quantum computation

Computational process must be studied using the fixed model of computational device. This chapter introduces the basic models of computation used in quantum information theory. We show how these models are defined by extending classical models.

We start by introducing basic facts about classical and quantum Turing machines. This model helps to understand how useful quantum computing can be. It can be also used to discuss the difference between quantum and classical computation. For the sake of completeness we also give brief introduction to the main results of quantum complexity theory. Next we introduce Boolean circuits and describe the most widely used model of quantum computation, namely quantum circuits. We focus on this model since many presented facts about quantum circuits are used in the following chapters. Finally we introduce another model which is more suited for defining programming languages operating on quantum memory — quantum random access machine (QRAM).

Note that we will not discuss problems related to the physical realization of described models. We also do not cover the area of quantum error correcting codes, which aims to provide methods for dealing with decoherence in quantum systems. For an introduction to those problems and recent progress in this area see for example [17] and [5].

2.1 Computability

Classically computation can be described using various models. The choice of the model used depends on the particular purpose or problem. Among the most important models of computation we can point:

- ❏ Turing Machine introduced in 1936 by Turing [52] and used as the main model in complexity theory [94].
- ❏ Random Access Machine [22, 105] which is the example of register machines; this model captures the main features of modern computers and provides theoretical model for programming languages.
- ❏ Boolean circuits defined in terms of logical gates and used to compute Boolean functions $f : \{0,1\}^m \to \{0,1\}^n$; they are used in complexity theory to study circuit complexity.
- ❏ Lambda calculus defined by Church [20] and used as the basis for many functional programming languages [2].
- ❏ Universal programming languages which are probably the most widely used model of computation [80].

It can be shown that all these models are equivalent [94, 52]. In other words the function which is computable using one of these models can be computed using any other model. It is quite suprising since Turing machine is a very simple model, especially when compared with RAM or programming languages.

In particular the model of multitape Turing machine is regarded as a canonical model. This fact is captured by the Church-Turing hypothesis.

Hypothesis 2.1 (Church-Turing) Every function which would be naturally regarded as computable can be computed by a universal Turing machine.

Although stated as a hypothesis this thesis is one of the fundamental axioms of modern computer science.

Universal Turing machine is a machine which is able to simulate any other machine. The simplest method for constructing such device is to use the model of Turing machine with two tapes [65, 94].

Research in quantum information processing is motivated by the extended version of Church-Turing thesis formulated by Deutsch [25].

Hypothesis 2.2 (Church-Turing-Deutsch) Every physical process can be simulated by a universal computing device.

In other words this thesis states that if the laws of physics are used to construct Turing machine, such model might provide greater computational power when compared with the classical model. Since the basic laws of physics are formulated as quantum mechanics, such improved version of Turing machine should be governed by the laws of quantum physics.

In this chapter we review some of these computational models focusing of their quantum counterparts. The discussion of quantum programming languages, which are based on the quantum random access machines (QRAM), is presented in Chapter 3.

We start be recalling the basic facts concerning Turing machine. This model allows to establish clear notion of computational resources like time and space used during computation. It is also used to precisely define other models introduced in this chapter.

On the other hand for practical purposes the notion of Turing machine is clumsy. Even for simple algorithms it requires quite complex description of transition rules. Thus we use more sophisticated methods like Boolean circuits and programming languages to describe the results presented in this thesis.

2.2 Turing machine

The model of Turing machine is widely used in classical and quantum complexity theory. Despite of its simplicity it captures the notion of computability.

In what follows by alphabet $A = \{a_1, \dots, a_n\}$ we mean any finite set of characters or digits. Elements of A are called letters. Set A^k contains all strings of length k composed from elements of A. Elements of A^k are called words and the length of the word w is denoted by $|w|$. The set of all words over A is denoted by A^*. Symbol ϵ is used to denote an empty word. The complement of language $L \subset A^*$ is denoted by $\bar{L}$ and it is the language defined as $\bar{L} = A^* - L$.

2.2.1 Classical Turing machine

Turing machine can operate only using one data structure – string of symbols. Despite its simplicity, this model can simulate any algorithm with inconsequential loss of efficiency [94]. Classical Turing machine consists of

- ❏ an infinitely long tape containing symbols from the finite alphabet A,
- ❏ a head, which is able to read symbols from the tape and write them on the tape,
- ❏ memory for storing programme for the machine.

The programme for the Turing machine is given in terms of transition function δ. The schematic illustration of Turing machine is presented in Figure 2.1.

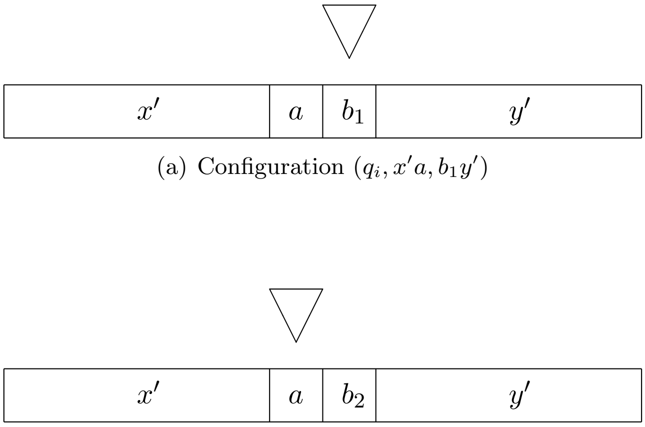
Formally, the classical deterministic Turing machine is defined as follows.

Definition 2.1 (Deterministic Turing machine) A deterministic Turing machine M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where

- ❏ Q is the set of internal control states,

- $q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,
- $\delta : Q \times A \to Q \times A \times \{-1, 0, 1\}$ is a transition function ie. the programme of a machine.

By the configuration of machine M we understand a triple (q_i, x, y), $q_i \in Q$, $x, y \in A^*$. This the describes the situation where the machine is in the state q_i, the tape contains the word xy and the machine starts to scan the word y. If $x = x'$ and $y = b_1 y'$ we can illustrate this situation as in Figure 2.1.



(a) Configuration $(q_i, x'a, b_1 y')$

(b) Configuration $(q_j, x', ab_2 y')$

Figure 2.1: Computational step of the Turing machine. Configuration $(q_i, x'a, b_1 y')$ is presented in (a). If the transition function is defined such that $\delta(q_i, b_1) = (q_2, b_2, -1)$ that computational step leads to configuration $(q_j, x', ab_2 y')$ (see (b)).

The transition from the configuration c_1 to the configuration c_2 is called a computational step. We write $c \vdash c'$ if δ defines the transition from c to c'. In this case c' is called the successor of c.

Turing machine can be used to compute values of functions or to decide about input words. The computation of a machine with input $w \in A^*$ is defined as the sequences of configurations $c_0, c_1, c_2, \ldots$, such that $c_0 = (q_i, \epsilon, w)$ and $c_i \vdash c_{i+1}$. We say that computation halts if some c_i has no successor or for configuration c_i, the state of the machine is q_a (machine accepts input) or q_r (machine rejects input).

The computational power of the Turing machine has its limits. Let us define two important classes of languages.

Definition 2.2 A set of words $L \in A^*$ is a recursively enumerable language if there exists a Turing machine accepting input w iff $w \in L$.

Definition 2.3 A set of words $L \in A^*$ is a recursive language if there exists a Turing machine M such that

❏ M accepts w iff $e \in L$,

❏ M halts for any input.

The computational power of the Turing machine is limited by the following theorem.

Theorem 2.1 There exists a language H which is recursively enumerable but not recursive.

Language H used in the above theorem is defined in halting problem [94]. It consists of all words composed of words encoding Turing machines and input words for those machines, such that a particular machine halts on a given word. Universal Turing machine can simulate any machine, thus for a given input word encoding machine and input for this machine we can easily perform the required computation.

Deterministic Turing machine is used to measure time complexity of algorithms. Note that if for some language there exists a Turing machine accepting it, we can use this machine as an algorithm for solving this problem. Thus we can measure the running time of the algorithm by counting the number of computational steps required for Turing machine to output the result.

The time complexity of algorithms can be described using following definition.

Definition 2.4 Class $\mathbf{TIME}(f(n))$ consists of all languages L such that there exists a deterministic Turing machine running in time $f(n)$ accepting input w iff $w \in L$.

In particular complexity class $\mathbf{P}$ defined as

$$\mathbf{P} = \bigcup_k \mathbf{TIME}(n^k). \tag{2.1}$$

captures the intuitive class of problems which can be solved easily on a Turing machine.

2.2.2 Nondeterministic and probabilistic computation

Since one of the main features of quantum computers is their ability to operate on the superposition of states we can easily extend classical model of probabilistic Turing machine and use it to describe quantum computation. Since in general many results in the area of algorithms complexity are stated in the terms of a nondeterministic Turing machine we start by introducting this model.

Definition 2.5 (Nondeterministic Turing machine) A nondeterministic Turing machine M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where

❑ Q is the set of internal controll states,

❑ $q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,

❑ $\delta \subset Q \times A \times Q \times A \times \{-1, 0, 1\}$ is a relation.

The last condition in the definition of a nondeterministic machine is the reason for its power. It also requires to change the definition of accepting by the machine.

We say that a nondeterministic Turing machine accepts input w if for some initial configuration (q_i, ϵ, w) computation leads to configuration (q_a, a_1, a_2) for some words a_1 and a_2. Thus a nondeterministic machine accepts input if there exists some computational path defined by transition relation δ leading to an accepting state q_a.

The model of a nondeterministic Turing machine is used to define complexity classes **NTIME**.

Definition 2.6 Class $\mathbf{NTIME}(f(n))$ consists of all languages L such that there exists a nondeterministic Turing machine running in time $f(n)$ accepting input w iff $w \in L$.

The most prominent example of such complexity classes is **NP**, which is the union of all $\mathbf{NTIME}(n^k)$

$$\mathbf{NP} = \bigcup_k \mathbf{NTIME}(n^k). \tag{2.2}$$

Nondeterministic computation Turing machine is used as a theoretical model in complexity theory. However, it is hard to imagine how such device operates [65]. One can illustrate the computational path of nondeterministic machine as in Figure 2.2.2.

Since our aim is to provide the model of physical device we restrict ourselves to more realistic model. We can do that by assigning to each element of relation a number representing probability. In this case we obtain model of probabilistic Turing machine.

Definition 2.7 (Probabilistic Turing machine) A probabilistic Turing machine M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where

❑ Q is the set of internal control states,

❑ $q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,

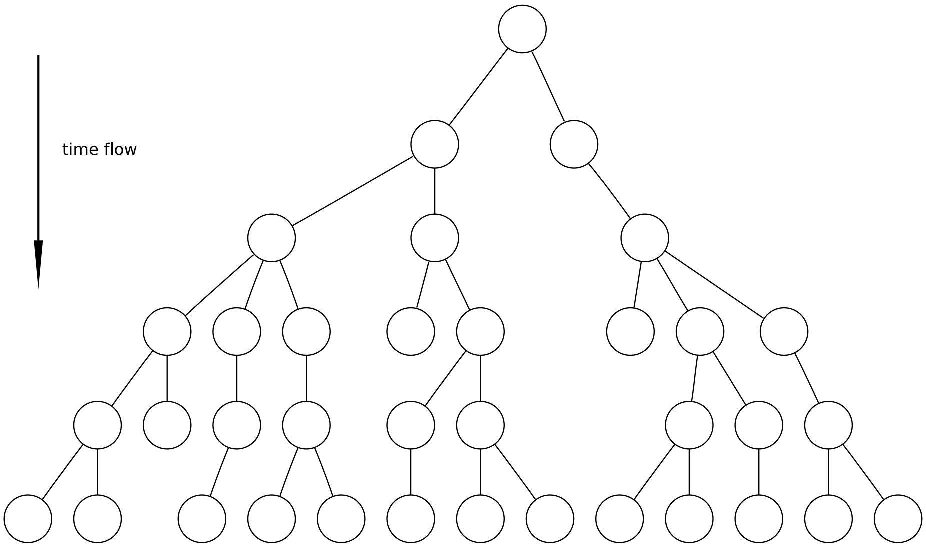


Figure 2.2: Schematic illustration of the computational paths of nondeterministic Turing machine [94]. Each circle represents the configuration of the machine. The machine can be in many configurations simultanously.

- $\delta : Q \times A \times Q \times A \times \{-1, 0, 1\} \to [0, 1]$ is a transition probability function ie.

$$\sum_{(q_2,a_2,d)\in Q\times A\times\{-1,0,1\}} \delta(q_1, a_1, q_2, a_2, d) = 1. \tag{2.3}$$

For a moment we can assume that probabilities of transition used by a probabilistic Turing machine can be represented only by rational numbers. we do this to avoid problems with machines operating on arbitrary real numbers. We will address this problem when extending the above definition to the quantum case.

The time complexity of computation can be measured in terms of the number of computational steps of the Turing machine required to execute a programme. Among important complexity classes we have chosen to point out:

- **P** – the class of languages for each there exists a deterministic Turing machine running in polynomial time,

- **NP** – the class of languages for each there exists a nondeterministic Turing machine running in polynomial time,

- **RP** – the class of languages L for each there exists a probabilistic Turing machine M such that: M accepts input w with probability at least $\frac{1}{2}$ if $w \in L$ and always rejects w if $w \notin L$,
- **coRP** – the class of languages L for each $\bar{L}$ is in **RP**,
- **ZPP** – **RP** $\cap$ **coRP**.

2.2.3 Quantum Turing machine

Quantum Turing machine was introduced by Deutsch in [25]. This model is equivalent to quantum circuit model [124, 90]. However, it is very inconvenient for describing quantum algorithms since the state of a head and the state of a tape are described by state vectors.

Quantum Turing machine consists of

- Processor: M 2-state observables $\{n_i | i \in_{\mathbb{Z}_M}\}$.
- Memory: infinite sequence of 2-state observables $\{m_i | i \in \mathbb{Z}\}$.
- Observable x, which represents the address of the current head position.

The state of the machine is described by the vector $|\psi(t)\rangle = |x; n_0, n_1, \ldots; m\rangle$ in the Hilbert space $\mathcal{H}$ associated with the machine.

At the moment $t = 0$ the state of the machine is described by the vectors $|\psi(0)\rangle = \sum_m a_m |0; 0, \ldots, 0; \ldots, 0, 0, 0, \ldots\rangle$ such that

$$\sum_i |a_i|^2 = 1. \tag{2.4}$$

The evolution of the quantum Turing machine is described by the unitary operator U acting on $\mathcal{H}$.

Classical probabilistic (or nondeterministic) Turing machine can be described as a quantum Turing machine such that at each step of its evolution the state of the machine is represented by the base vector.

The formal definition of the quantum Turing machine was introduced in [10].

It is common to use real numbers as amplitudes when describing the state of quantum systems during quantum computation. To avoid problems with an arbitrary real number we introduce the class of numbers which can be used as amplitudes for amplitude transition functions of the quantum Turing machine.

Let us denote by $\widetilde{\mathbb{C}}$ the set of complex numbers $c \in \mathbb{C}$, such that there exists a deterministic Turing machine, which allows to calculate $\mathrm{Re}\,(c)$ and $\mathrm{Im}\,(c)$ with accuracy $\frac{1}{2^n}$ in time polynomial in n.

Definition 2.8 (Quantum Turing Machine) A quantum Turing machine (QTM) M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where

❏ Q is the set of internal control states,

❏ $q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,

❏ $\delta : Q \times A \times Q \times A \times \{-1, 0, 1\} \to \widetilde{\mathbb{C}}$ is a transition amplitude function ie.

$$\sum_{(q_2,a_2,d)\in Q\times A\times\{-1,0,1\}} |\delta(q_1, a_1, q_2, a_2, d)|^2 = 1. \tag{2.5}$$

Reversible classical Turing machines (ie.Turing machines with reversible transition function) can be viewed as particular examples of quantum machines. Since any classical algorithm can be transformed into reversible form, it is possible to simulate classical Turing machine using quantum Turing machine.

2.2.4 Quantum complexity

Quantum Turing machine allows for rigorous analysis of algorithms. This is important since the main goal of quantum information theory is to provide gain in terms of speed or memory with respect to classical algorithms. It should be stressed that at the moment no formal proof has been given that quantum Turing machine is more powerful than classical Turing machine.

In this section we give some results concerning quantum complexity theory.

In analogy to classical case it is possible to define complexity classes for the quantum Turing machine. The most important complexity class is this case is **BQP**.

Definition 2.9 Complexity class **BQP** contains languages L for which there exists quantum Turing machine running in polynomial time such that, for any input word x this word is accepted with probability at least $\frac{3}{4}$ if $x \in L$ and is rejected with probability at least $\frac{3}{4}$ if $x \notin L$.

Class **BQP** is a quantum counterpart of the classical class **BPP**.

Definition 2.10 Complexity class **BPP** contains languages L for which there exists nondeterministic Turing machine running in polynomial time such that, for any input word x this word is accepted with probability at least $\frac{3}{4}$ if $x \in L$ and is rejected with probability at least $\frac{3}{4}$ if $x \notin L$.

Since many results in complexity theory are stated in terms of oracles, we define an oracle as follows.

Definition 2.11 Oracle or black box is an imaginary machine which can decide certain problems in a single operation.

We use notation $\mathbf{A}^{\mathbf{B}}$ to describe the class of problems solvable by an algorithm in class **A** with an oracle for the language **B**.

It was shown in [10] that

Theorem 2.2 Complexity classes fulfil the following inequality

$$\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{P}^{\#\mathbf{P}}. \tag{2.6}$$

Complexity class #**P** consists of problems of the form compute f(x), where f is the number of accepting paths of an **NP** machine. For example problem #**SAT** formulated below is in #**P**.

Problem 2.1 (#**SAT**) For a given Boolean formula, compute how many satisfying true assignments it has.

Complexity class $\mathbf{P}^{\#\mathbf{P}}$ consists of all problems solvable by a machine running in polynomial time which can use oracle solving problems in #**P**.

2.3 Quantum computational networks

After presenting the basic facts about Turing machines we are ready to introduce more usable models of computing devices. We start be defining Boolean circuits and extending this model to quantum case. Quantum circuits defined in this section will be used to describe algorithms in Chapters 4 and 5.

2.3.1 Boolean circuits

Boolean circuits are used to compute functions of the form

$$f : \{0,1\}^m \to \{0,1\}^n. \tag{2.7}$$

Basic gates (functions) which can be used to define such circuits are

- $\wedge : \{0,1\}^2 \to \{0,1\}$, $\wedge(x,y) = 1 \Leftrightarrow x = y = 1$ (logical and),
- $\vee : \{0,1\}^2 \to \{0,1\}$, $\vee(x,y) = 0 \Leftrightarrow x = y = 0$ (logical or),
- $\sim : \{0,1\} \to \{0,1\}$, $\sim(x) = 1 - x$ (logical not).

The set of gates is called universal if all functions $\{0,1\}^n \to \{0,1\}$ can be constructed using the gates from this set. It is easy to show that the set of functions composed of the $\sim$, $\vee$ and $\wedge$ is universal. Thus it is possible to computer any functions $\{0,1\}^n \to \{0,1\}^m$ using only these functions. The full characteristic of universal sets of functions was given by Post in 1949 [127].

Using above set of functions a Boolean circuit is defined as follows.

Definition 2.12 (Boolean circuit) A Boolean circuit is an acyclic direct graph with nodes labelled by input variable, output variables or logical gates $\vee$, $\wedge$ or $\sim$.

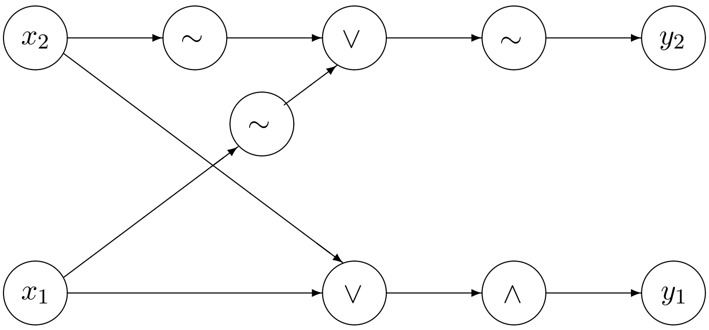


Figure 2.3: The example of a Boolean circuit computing the sum of bits x_1 and x_2 [50]. Nodes labelled x_1 and x_2 represent input variables and nodes labelled y_1 and y_2 represent output variables.

Input variable node has no incoming arrow while output variable node has no out-coming arrows. The example of a Boolean circuit computing the sum of bits x_1 and x_2 is given in Figure 2.3.

Note that in general it is possible to define Boolean circuit using different sets of elementary functions. Since functions $\vee$, $\wedge$ and $\sim$ provide universal set of gates we defined Boolean circuit using these particular functions.

Function $f : \{0,1\}^m \to \{0,1\}$ is defined on the binary string of arbitrary length. Let $f_n : \{0,1\}^m \to \{0,1\}^n$ be a restriction of f to $\{0,1\}^n$. For each such restriction there is a Boolean circuit C_n computing f_n. We say that $C_0, C_1, C_2, \ldots$ is a family of Boolean circuits computing f.

Note that any binary language $L \subset \{0,1\}^*$ can be accepted by some family of circuits. But since we need to know the value of f_n to construct a circuit C_n such family is not an algorithmic device at all. We can state that there a exists family accepting language but we don't know how to build it [94].

To show how Boolean circuits are related to Turing machine we introduce uniformly generated circuits.

Definition 2.13 We say that language $L \in A^*$ has uniformly polynomial circuits if there exists a Turing machine M that an input $\underbrace{1\ldots1}_{n}$ outputs the graph of circuit C_n using space $O(\log n)$, and the family $C_0, C_1, \ldots$ accepts L.

The following theorem provides a link between uniformly generated circuits and Turing machines.

Theorem 2.3 A language L has uniformly polynomial circuit iff $L \in \mathbf{P}$.

Quantum circuits model is an analogous to uniformly polynomial circuits. They can be introduced as the straightforward generalisation of reversible circuits.

2.3.2 Reversible circuits

The evolution of isolated quantum systems is described by a unitary operator U (see Appendix B). The main difference with respect to classical evolution is that this type of evolution is reversible.

Before introducting quantum circuit we define reversible Boolean circuit

Definition 2.14 (Reversible gate) A classical reversible function (gate) $\{0,1\}^m \to \{0,1\}^m$ is a permutation.

Definition 2.15 A reversible Boolean circuit is a Boolean circuit composed of reversible gates.

The important fact expressed by the following theorem allows us to simulate any classical computation no a quantum machine described using a reversible circuit

Theorem 2.4 All Boolean circuits can be simulated using reversible Boolean circuits.

Like in the case of nonreversible circuit one can introduce the universal set of functions for reversible circuits.

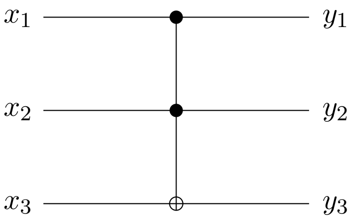


Figure 2.4: Classical Toffoli gate is universal for reversible circuits. It was also used in [26] to provide the universal set of quantum gates.

The important example of a gate universal for reversible Boolean circuits is a Toffoli gate. The graphical representation of this gate is presented in Figure 2.4. The following theorem was proved by Toffoli [112].

Theorem 2.5 A Toffoli gate is a universal reversible gate.

As we will see in the following section it is possible to introduce two-bit quantum gates which are universal for quantum circuits. This is impossible in classical case and one needs at least a three-bit gate to construct the universal set of reversible gates.

In particular, any reversible circuit is automatically a quantum circuit. However quantum circuits offer much more diversity in terms of number of allowed operations.

2.3.3 Quantum circuits

The computational process of the quantum Turing machine is complicated since data as well as control variables can be in a superposition of base states. To provide more convenient method of describing quantum algorithms one can use quantum circuits model.[1]

Quantum circuits model was first introduced by Deutsch in [26] and it is the most commonly used notation for quantum algorithms. It is much easier to imagine than the quantum Turing machine since the control variables (executed steps and their number) are classical. The only data in a quantum circuit are quantum.

Quantum circuit consists of the following elements (see Table 2.2):

- the finite sequence of wires representing qubits or sequences of qubits (quantum registers),
- quantum gates representing elementary operations from the particular set of operations implemented on a quantum machine,
- measurement gates representing measurement operation, which is usually executed as the final step of quantum algorithm. It is commonly assumed that is is possible to perform measurement on each qubit in canonical basis $\{|0\rangle, |1\rangle\}$ which corresponds to the measurement of the S_z observable.

The concept of a quantum circuit is the natural generalisation of acyclic logic circuits studied in classical computer science. Quantum gates have the same number of inputs as outputs. Each n qubit quantum gate represents the 2^n-dimensional unitary operation of the group $U(2^n)$, ie. generalised rotation in a complex Hilbert space.

The main adventage of this model is it's simplicity. It also provides very convenient representation of physical evolution in quantum systems.

Elements of quantum circuit model

From the mathematical point of view quantum gates are unitary matrices acting on n-dimensional Hilbert space. They represent the evolution of an isolated quantum system [89].

[1] This models is sometimes called quantum gate arrays model.

The problem of constructing new quantum algorithms requires more careful study of operations used in quantum circuit model. In particular we are interested in efficient decomposition of quantum gates into elementary operations.

We start by providing basic charactersistic of unitary matrices [7, 89]

Theorem 2.6 Every unitary 2×2 matrix $G \in U(2)$ can be decomposed using elementary rotations as

$$G = R_z(\beta)R_y(\theta)R_z(\alpha)\Phi(\delta) \tag{2.8}$$

where

$$\Phi(\xi) = \left(\begin{array}{cc} e^{i\xi} & 0 \\ 0 & e^{i\xi} \end{array}\right),\ R_y(\xi) = \left(\begin{array}{cc} \cos(\xi/2) & \sin(\xi/2) \\ -\sin(\xi/2) & \cos(\xi/2) \end{array}\right),$$

and

$$R_z(\xi) = \left(\begin{array}{cc} e^{i\frac{\xi}{2}} & 0 \\ 0 & e^{-i\frac{\xi}{2}} \end{array}\right).$$

We introduce the definition of quantum gates as stated in [50].

Definition 2.16 A quantum gate U acting on m qubits is a unitary mapping (see Appendix B) on $\mathbb{C}^{2^m} \equiv \underbrace{\mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2}_{m \text{ times}}$

$$U : \mathbb{C}^{2^m} \to \mathbb{C}^{2^m}, \tag{2.9}$$

which operates on the fixed number of qubits.

Formally quantum circuit is defined as the unitary mapping which can be decompose into the sequence of elementary gates.

Definition 2.17 A quantum circuit on m qubits is a unitary mapping on $\mathbb{C}^{2^m}$, which can be represented as a concatenation of a finite set of quantum gates.

Examples

Any reversible classical gate is also a quantum gate. In particular logical gate $\sim$ (negation) is represented by quantum gate NOT, which is realized by σ_x Pauli matrix.

As we know any Boolean circuit can be simulated by a reversible circuit and thus any function computed by a Boolean circuit can be computed using a quantum circuit. Since quantum circuit operates on a vector in complex Hilbert space it allows for new operations typical for this model.

The first example of quantum gate which has no classical counterpart is $\sqrt{NOT}$ gate. It has property

$$\sqrt{NOT}\sqrt{NOT} = NOT, \tag{2.10}$$

which cannot be fulfilled by any classical Boolean function $\{0,1\} \to \{0,1\}$. Gate $\sqrt{N}$ is represented by the unitary matrix

$$\sqrt{NOT} = \frac{1}{2}\left(\begin{array}{cc} 1+i & 1-i \\ 1-i & 1+i \end{array} \right). \tag{2.11}$$

Another example is Hadamard gate H. This gate is used to introduce the superposition of base states. It acts on the base state as

$$H|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right), H|1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right). \tag{2.12}$$

If the gate G is a quantum gate acting on one qubit it is possible to construct the family of operators acting on many qubits. Particularly important class of multiqubit operations is the class of controlled operations.

Definition 2.18 (Controlled gate) Let G be a 2×2 unitary matrix representing a quantum gate. Operator

$$|1\rangle\langle 1| \otimes G + |0\rangle\langle 0| \otimes \mathbb{I} \tag{2.13}$$

acting on two qubits, is called a controlled-G gate.

Here $A \otimes B$ denotes the tensor product of gates (unitary operator) A and B and $\mathbb{I}$ is an identity matrix (see Appendix B). If in the above definition we take $G = NOT$ we get

$$|1\rangle\langle 1| \otimes \sigma_x + |0\rangle\langle 0| \otimes \mathbb{I} = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array} \right), \tag{2.14}$$

which is the definition of $CNOT$ (controlled-NOT) gate. This gate can be used to construct the universal set of quantum gates. Also this gate allows to introduce entangled states during computation

$$CNOT(H \otimes \mathbb{I})|00\rangle = CNOT\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \otimes |0\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) \tag{2.15}$$

The classical counterpart of $CNOT$ gate is XOR gate.

Other examples of single qubit and two qubit quantum gates are presented in Table 2.2. In Figure 2.5 a quantum circuit for quantum Fourier transform on three qubits is presented.

One can extend definition 2.18 and introduce quantum gates with many controlled qubits.

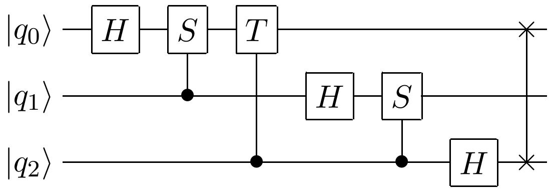


Figure 2.5: Quantum circuit representing quantum Fourier transform for three qubits. Elementary gates used in this circuit are described in Table 2.2.

Definition 2.19 Let G be a 2×2 unitary matrix. Quantum gate defined as

$$|\underbrace{1\dots1}_{n-1}\rangle\langle\underbrace{1\dots1}_{n-1}| \otimes G + \sum_{l \neq \underbrace{1\dots1}_{n-1}} |l\rangle\langle l| \otimes \mathbb{I} \tag{2.16}$$

is called $(n-1)$-controlled G gate. We denote this gate by $\wedge_{n-1}(G)$.

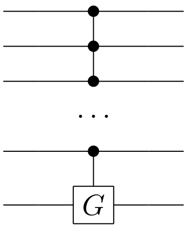


Figure 2.6: Generalised quantum Toffoli gate acting on n qubits. Gate G is controlled by the state of $n-1$ qubits according to definition 2.19.

This gate $\wedge_{n-1}(G)$ is sometimes referred to as a generalised Toffoli gate or a Toffoli gate with m controlled qubits. Graphical representation of this gate is presented in Figure 2.3.3. We will use this construction in Chapter 5.

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Logical values for XOR gate. Quantum CNOT gate computes value of x_1 XOR x_2 in the first register and stores values of x_2 in the second register.

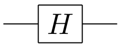
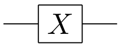
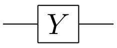
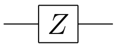
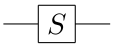
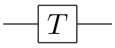
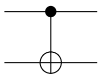
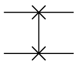
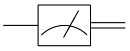
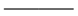
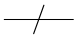
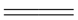
The name of the gate	Graphical representation	Mathematical form
Hadamard		$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Pauli X		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli Y		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli Z		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Phase		$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
$\pi/8$		$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$
CNOT		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
SWAP		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Measurement		$\left\{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right\}$
qubit		wire ≡ single qubit
n qubits		wire representing n qubits
classical bit		double wire ≡ single bit

Table 2.2: Basic gates used in quantum circuits with their graphical representation and mathematical form. Note that measurement gate is represented in Kraus form (see Section B.1.1), since it is the example of non-unitary quantum evolution.

Decomposition of quantum gates

The important feature of quantum circuits is expressed by the following universality property [7].

Theorem 2.7 The set of gates consisting of all one-qubit gates $U(2)$ and one two-qubit CNOT gate is universal in the sense that any n qubit operation can be expressed as the composition of these gates.

Note that in contrast to classical case, where one needs at least three-bit gates to construct a universal set, quantum circuits can be simulated using one two-qubit universal gate.

In order to implement a quantum algorithm one has to decompose many qubit quantum gates into elementary gates. It has been shown that almost any n qubit quantum gate ($n \geq 2$) can be used to the build universal set of gates [27] in the sense that any unitary operation on the arbitrary number of qubits can be expressed as the composition of gates from this set. In fact the set consisting of two-qubit exclusive-or (XOR) quantum gate and all single-qubit gates is also universal [7].

Let us assume that we have the set of gates containing only CNOT and one-qubit gates. In [104] theoretical lower bound for the number of gates required to simulate a circuit using these gates was derived. The efficient method of elementary gates sequence synthesis for an arbitrary unitary gate was presented in [86].

Theorem 2.8 (Shende-Markov-Bullock) Almost all n-qubit operators cannot be simulated by a circuit with fewer than $\lceil \frac{1}{4}[4^n - 3n - 1] \rceil$ CNOT gates.

In [116] the construction providing the efficient way of implementing arbitrary quantum gates was described. The resulting circuit has complexity $O(4^n)$ which coincides with lower bound from Theorem 2.8.

Since in Chapter 5 we use gates with many controlled and one target qubits, it is useful to provide more details about this special case. The following results were proved in [7].

Theorem 2.9 For any single-qubit gate U the gate $\wedge_{n-1}(U)$ can be simulated in terms of $\Theta(n^2)$ basic operations.

In many situation it is useful to construct a circuit which approximates the required circuit. We say that quantum circuits approximate other circuits with accuracy ε if the distance (in terms of Euclidean norm) between unitary transformations associated with these circuits is at most ε [7].

Theorem 2.10 For any single-qubit gate U and $\varepsilon > 0$ gate $\wedge_{n-1}(U)$ can be approximated with accuracy ε using $\Theta(n \log \frac{1}{\varepsilon})$ basic operations.

Note that the efficient decomposition of quantum circuit is crucial in physical implementation of quantum information processing. In particular case decomposition can be optimised using the set of elementary gates specific for target architecture. CNOT gates are of big importance since they allow to introduce entangled states during computation. It is also hard to physically realise CNOT gate since one needs to control physical interaction between qubits.

2.4 Random access machines

Quantum circuit model does not provide a mechanism for controlling with classical machine the operations on quantum memory. Usually quantum algorithms are described using mathematical representation, quantum circuits and classical algorithms [63]. The model of quantum random access machine is built on an assumption that the quantum computer has to be controlled by a classical device [93]. Schematic presentation of such architecture is provided in Figure 2.7.

Quantum random access machine is interesting for us since it provides convenient model for developing quantum programming languages. However, these languages are our main area of interest. We see no point in providing the detailed description of this model as it is given in [93] together with the description of hybrid architecture used in quantum programming.

2.4.1 Classical RAM model

The classical model of random access machine (RAM) is the example of more general register machines [22, 94, 105].

The random access machine consists of an unbounded sequence of memory registers and finite number of arithmetic registers. Each register may hold an arbitrary integer number. The programme for the RAM is a finite sequence of instructions $\Pi = (\pi_1, \ldots, \pi_n)$. At each step of execution register i holds an integer r_i and the machine executes instruction π_κ, where κ is the value of the programme counter. Arithmetic operations are allowed to compute the address of a memory register.

Despite the difference in the construction between Turing machine and RAM, it can be easily shown that Turing machine can simulate any RAM machine with polynomial slow-down only [94].

2.4.2 Quantum RAM model

Quantum random access machine (QRAM) model is the extension of the classical RAM, which can exploit quantum resources and at the same time can be used to perform any kind of classical computation. QRAM allows us to control operations performed on quantum registers and provides the set of instructions for defining them.

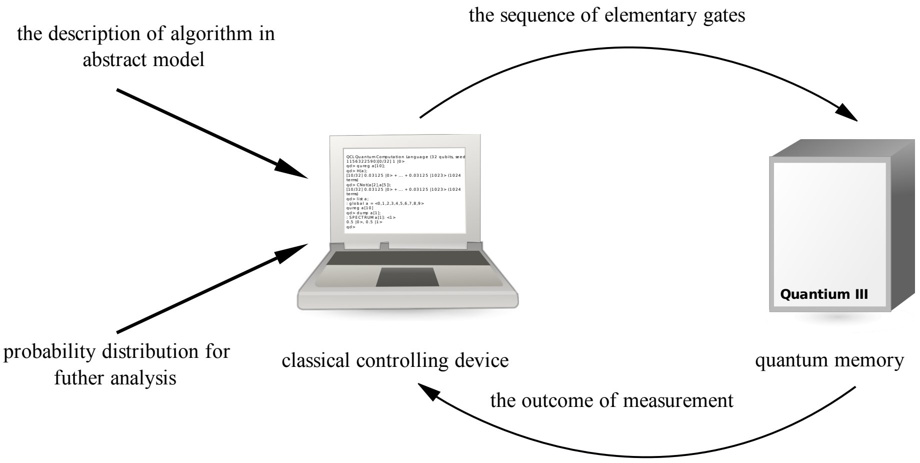

the description of algorithm in abstract model
the sequence of elementary gates
Quantium III
probability distribution for futher analysis
classical controlling device
quantum memory
the outcome of measurement

Figure 2.7: The model of classically controlled quantum machine [93]. Classical computer is responsible for performing unitary operations on quantum memory. The results of quantum computation are received in the form of measurement results.

The quantum part of QRAM model is used to generate probability distribution. This is achieved by performing measurement on quantum registers. The obtained probability distribution has to be analysed using classical computer.

2.4.3 Quantum pseudocode

Quantum algorithms are in most of the cases described using the mixture of quantum gates, mathematical formulas and classical algorithms. The first attempt to provide a uniform method of describing quantum algorithms was made in [21], where the author introduces a high-level notation based on the notation known from computer science textbooks [23, 52].

In [63] the first formalised language for description of quantum algorithms was introduced. Moreover, it was tightly connected with the model of quantum machine called quantum random access machine (QRAM).

Quantum pseudocode proposed by Knill [63] is based on conventions for classical pseudocode proposed in [23, Chapter 1]. Classical pseudocode was designed to be readable by professional programmers, as well as people who had done a little programming. Quantum pseudocode introduces operations on quantum registers. It also allows to distinguish between classical and quantum registers.

Quantum registers are distinguished by underlining them. They can be introduced by applying quantum operations to classical registers or by calling a subroutine which returns a quantum state. In order to convert a quantum register into a classical register measurement operation has to be perfomed.

The example of quantum pseudocode is presented in Listing 2.1. It shows the main advantage of QRAM model over quantum circuits model – the ability to incorporate classical control into the description of quantum algorithm.

Procedure: Fourier($\underline{a}, d$)
Input: A quantum register $\underline{a}$ with d qubits. Qubits arenumbered form 0 to $d-1$.
Output: The amplitudes of $\underline{a}$ are Fourier transformed over $\mathbb{Z}_{2^d}$.

```
C: assign value to classical variable
$\omega \leftarrow e^{i2\pi/2^d}$
C: perform sequence of gates
for $i = d-1$ to $i = 0$
  for $j = d-1$ to $j = i+1$
  if $\underline{a_j}$ then $\mathcal{R}_{\omega^{2^{d-i-1+j}}}(\underline{a_i})$
    C: number of loops executing phase depends on
    C: the required accuracy of the procedure
  $\mathcal{H}(\underline{a_i})$

C: change the order of qubits
for $j = 0$ to $j = \frac{d}{2} - 1$
  $\mathcal{SWAP}(\underline{a_j}, \underline{a_{d-a-j}})$
```

Listing 2.1: Quantum pseudoceode for quantum Fourier transform on d qubits. Quantum circuit for this operation with $d = 3$ is presented in Figure 2.5.

Operation $\mathcal{H}(\underline{a_i})$ executes a quantum Hadamard gate on a quantum register $\underline{a_i}$ and $\mathcal{SWAP}(\underline{a_i}, \underline{a_j})$ performs SWAP gate between $\underline{a_i}$ and $\underline{a_j}$. Operation $\mathcal{R}_\phi(\underline{a_i})$ executes a quantum gate $R(\phi)$ is defined as

$$R(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}, \tag{2.17}$$

on the quantum register $\underline{a_i}$. Using conditional contruction

if $\underline{a_j}$ then $\mathcal{R}_\phi(\underline{a_i})$

it is easy to define controlled phase shift gate (see Definition 2.19). Similar construction exists in QCL quantum programming language described in Chapter 3.

The measurement of a quantum register can be indicated using an assignement.

$a_j \leftarrow \underline{a_j}$

2.4.4 Quantum programming environment

Since the main aim of this thesis is to present the advantages of high-level programming languages, we need to explain how such languages are related to quantum random access machine. Thus as the summary of this chapter we present the overview of an architecture for quantum programming, which is based on QRAM model.

The architecture proposed in [110, 111] is designed for transforming a high-level quantum programming language to the technology-specific implementation set of operations. This architecture is composed of four layers:

- High level programming language providing high-level mechanisms for performing useful quantum computation; this language should be independent from particular physical implementation of quantum computing.
- Compiler of this language providing architecture independent optimisation; also compilation phase can be used to handle quantum error correction required to perform useful quantum computation.
- Quantum assembly language (QASM) – assembly language extended by the set of instructions used in the quantum circuit model.
- Quantum physical operations language (QCPOL), which describes the execution of quantum programme in a hardware-dependent way; it includes physical operations and it operates on the universal set of gates optimal for a given physical implementation.

The authors of [110, 111] do not define a specific high-level quantum programming language. However, they point out that existing languages, mostly based on Dirac notation, do not provide the sufficient level of abstraction. They also stress, following [11], that it should have the basic set of features. We will discuss these basic requirements in detail in Chapter 3. At the moment quantum assembly language (QASM) is the most interesting part of this architecture, since it is tightly connected to the QRAM model.

QASM should be powerful enough for representing high level quantum programming language and it should allow for describing any quantum circuit. At the same time it must be implementation-independent so that it could be used to optimise execution of the programme with respect to different architectures.

QASM uses qubits and cbits (classical bit) as basic units of information. Quantum operations consist of unitary operations and measurement. Moreover each unitary operator is expressed in terms of single qubit gates and CNOT gates.

In the architecture proposed in [111] each single-qubit operation is stored as the triple of rationals. Each rational multiplied by π represents one of three Euler-angles, which are sufficient to specify one-qubit operation.

2.5 Further reading

The methods for efficient decomposing arbitrary quantum gates into the sequence of elementary operations were described in [86, 87]. In [125] necessary condition for the optimal construction of a two-qubit unitary operation is discussed.

Recently a new model of sequential quantum random machine (SQRAM) has been proposed. Instruction set for this model and compilation of high-level languages is discussed in [88]. However, it is very similar to QRAM model.

Complexity ZOO [1] contains the description of complexity classes and many famous problems from complexity theory. Complete introduction to the complexity theory can be found in [94]. Theory of **NP**-completeness with many examples of problems from this class is presented in [42].

Programming languages can be defined without using RAM model. Interesting programming language for Turing machine, providing the minimal set of instructions, was introduced in [15].

Many important results and basic definitions concerning quantum complexity theory can be found in [10]. The proof of equivalence between quantum circuit and quantum Turing machine was given in [124]. Interesting discussion of quantum complexity classes and relation of **BQP** class to classical classes can be found in [41].

Chapter 3

Quantum programming languages

In this chapter we briefly describe selected examples of the existing quantum programming languages.

We start by formulating the requirements which must be fulfilled by any universal quantum programming language. Next we describe languages based on imperative paradigm – QCL (Quantum Computation Language) and LanQ. We focus on QCL since it is used as the implementation language for quantum Parrondo's game described in Chapter 4.

We also describe recent research efforts focused on implementing languages based on functional paradigm and discuss adventages of a language based on this paradigm. As the example of functional quantum programming language we present cQPL.

We introduce syntax and discuss features of presented languages. We also point out their weaknesses. For the sake of completeness a few examples of quantum algorithms and protocols are presented. We use these examples to introduce the main features of presented languages.

3.1 Introduction

During the last few years many quantum programming languages were proposed [43].

Recently developed languages focus on quantum communication and thus provide syntactic elements which can be used to facilitate simulation of quantum protocols. Recently it also became clear that models of quantum computation must reflect the situation in real-world quantum systems where decoherence and errors are unavoidable.[1] Thus new quantum programming languages aim to incorporate the model of mixed states.

[1] For the basic introduction to quantum channels and errors see Appendix B.

Table 3.1 contains the comparison of few quantum programming languages. It includes the most important features of existing languages. In particular we list the underlying mathematical model (ie. pure or mixed states) and the support for quantum communication.

	QCL	Q Language	QPL	cQPL	LanQ	kulka
reference	[91]	[13]	[103]	[71]	[83]	
implemented	y	y	y	y	y	y
formal semantics	n	n	y	y	y	n
communication	n	n	n	y	y	n
universal	y	y	y	y	y	y
mixed states	n	n	y	y	y	n

Table 3.1: Comparison of quantum programming languages with information about implementation and basic features. Based on information in [83] and [71]. Experimental quantum programming language kulka is introduced in Appendix A.

All languages listed in Table 3.1 are universal and thus they can be used to compute any function computable on quantum Turing machnie. Consequently, all these language provide the model of quantum computation which is equivalent to the model of quantum Turing machine.

In this chapter we introduce the basic syntax of three of the languages listed in Table 3.1 – QCL, LanQ and cQPL. This is motivated by the fact that these languages have a working interpreter and can be used to perform simulations of quantum algorithms. In particular the results described in Chapter 4 were obtained using QCL interpreter. We introduce the elements of QCL required to understand the implementation of Parrondo's paradox. We also compare the main features of presented languages.

The syntax of experimental programming language kulka is described in Appendix A. The implementation of the interpreter for this language motivated the results presented in Chapter 5.

3.2 Requirements for quantum programming language

Taking into account QRAM model described in Chapter 2 we can formulate basic requirements which have to be fulfilled by any quantum programming language [12].

❏ Completeness: Language must allow to express any quantum circuit and thus enable the programmer to code every valid quantum programme written as a quantum circuit.

❏ Extensibility: Language must include, as its subset, the language implementing some high level classical computing paradigm. This isimportant since some parts of quantum algorithms (for example Shors algorithm) require nontrivial classical computation.

- Separability: Quantum and classical parts of the language should be separated. This allows to execute any classical computation on purely classical machine without using any quantum resources.
- Expressivity: Language has to provide high level elements for facilitating the quantum algorithms coding.
- Independence: The language must be independent from any particular physical implementation of a quantum machine. It should be possible to compile a given programme for different architectures without introducing any changes in its source code.

As we will see, the languages presented in this chapter fulfill most of above requirements. The main problem is the expressivity requirement. Parrondo's game presented in Chapter 4 shows that in many situations one has to use very low-level construction to implement the elements used in an algorithm. This fact has motivated the research on quantum data types described in Chapter 5.

3.3 Imperative quantum programming

First we focus on quantum programming languages which are based on the imperative paradigm. They include quantum pseudocode, discussed in Chapter 2, Quantum Computation Language (QCL) created by Ömer [93] and LanQ developed by Mlnařík [83].

Below we provide an introduction to QCL. It is one of the most popular quantum programming languages. Moreover, this language was used to implement the quantum version of Parrondo's game described in Chapter 4.

Next, we introduce the basic elements of LanQ [83]. This language provides the support for quantum protocols. This fact reflects the recent progress in quantum communication theory.

3.3.1 Quantum Computation Language

QCL (Quantum Computation Language) [91, 93] is the most advanced implemented quantum programming language. Its syntax resembles syntax of the C programming language [59] and classical data types are similar to data types in C or Pascal.

The basic built-in quantum data type in QCL is qureg (quantum register). It can be interpreted as the array of qubits (quantum bits).

```
qureg x1[2]; // 2-qubit quantum register x1
qureg x2[2]; // 2-qubit quantum register x2
H(x1); // Hadamard operation on x1
```

```
H(x2[1]); // Hadamard operation on the second qubit of the x2
```

QCL standard library provides standard quantum operators used in quantum algorithms, such as:

- Hadamard H and Not operations on many qubits,
- controlled not CNot with many target qubits and Swap gate,
- rotations: RotX, RotY and RotZ,
- phase Phase and controlled phase CPhase.

Most of them are described in Table 2.2 in Chapter 2.

Since QCL interpreter uses qlib simulation library, it is possible to observe the internal state of the quantum machine during the execution of the quantum programmes. The following sequence of commands defines two-qubit registers a and b and executes H and CNot gates on these registers.

```
qcl¿ qureg a[2];
qcl¿ qureg b[2];
qcl¿ H(a);
[4/32] 0.5 —0,0¿ + 0.5 —1,0¿ + 0.5 —2,0¿ + 0.5 —3,0¿
qcl¿ dump
: STATE: 4 / 32 qubits allocated, 28 / 32 qubits free
0.5 —0¿ + 0.5 —1¿ + 0.5 —2¿ + 0.5 —3¿
qcl¿ CNot(a[1],b)
[4/32] 0.5 —0,0¿ + 0.5 —1,0¿ + 0.5 —2,0¿ + 0.5 —3,0¿
qcl¿ dump
: STATE: 4 / 32 qubits allocated, 28 / 32 qubits free
0.5 —0¿ + 0.5 —1¿ + 0.5 —2¿ + 0.5 —3¿
```

Using dump command it is possible to inspect the internal state of quantum computer. This can be helpful for checking if our algorithm changes the state of quantum computer in the requested way.

One should note that dump operation is different from measurement, since it does not influence the state of quantum machine. This operation can be realised using simulator only.

Quantum memory management

Quantum memory can be controlled using quantum types qureg, quconst, quvoid and quscratch. Types qureg is used as a base type for general quantum registers. Other types allow for the optimisation of generated quantum circuit. The summary of types defined in QCL is presented in Table 3.3.1.

Type	Description	Usage
qureg	general quantum register	basic type
quvoid	register which has to be empty when operator is called	target register
quconst	must be invariant for all operators used in quantum conditions	quantum conditions
quscratch	register which has to be empty before and after the operator is called	temporary registers

Table 3.2: Types of quantum registers used for memory managemnt in QCL.

Classical and quantum procedures and functions

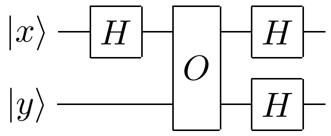
QCL supports user defined operators and functions know from languages like C or Pascal. Classical subroutines are defined using procedure keyword. Also standard elements, know from the C programming language, like looping (eg. for i=1 to n { ... }) and conditional structures (eg. if x==0 { ... }), can be used to controll the execution of quantum and classical elements. In addition to this, it provides two types of quantum subroutines.

The first type is used for unitary operators. Using it one can define new operations, which in turn be used to manipulate quantum data. For example operator diffuse defined in Listing 3.1 defines inverse about the mean operator used in Grover's algorithm [46]. This allows to define algorithms on the higher level of abstraction and extend the library of functions available for programmer. This feature will be used in Chapter 4 to implement basic elements of Parrondo's scheme.

```
operator diffuse(qureg q) {
  H(q);                  // Hadamard Transform
  Not(q);                // Invert q
  CPhase(pi,q);          // Rotate if q=1111..
  !Not(q);               // undo inversion
  !H(q);                 // undo Hadamard Transform
}
```

Listing 3.1: The implementation of the inverse about the mean operation in QCL [93]. Constant pi represents number π. Exampl ation mark ! is used to indicate that the interpreter should use the inverse of a given oprator. Operation diffuse is used in quantum search algorithm [46].

Using subroutines it is easy to describe quantum algorithms. Figure 3.1 presents QCL implementation of Deutsch's algorithm, along with the quantum circuit for this algorithm. This simple algorithm uses all the



```
operator U(qureg x,qureg y) {
  H(x);
  Oracle(x,y);
  H(x & y);
}

procedure deutsch() {           // Classical control structure
  qureg x[1];                   // allocate 2 qubits
  qureg y[1];
  int m;
  {                             // evaluation loop
    reset;                      //   initialise 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
}
```

Figure 3.1: Quantum circuit for Deutsch's algorithm and QCL implementation of this algorithm (see [93] for more examples). Evaluation loop is composed of preparation (performed by reset instruction), unitary evolution (U(x,y) operator) and measurement. Subroutine Oracle() implements function used in Deutsch's algorithm [25, 28] (see also Appendix B).

main elements of QCL. It also illustrates all the main ingredients of existing quantum algorithms.

The second type of quantum subroutine is called quantum function.[2] They can be defined using qufunct keyword. The subroutine of type qufunct is used for all transformations of the form

$$|n\rangle = |f(n)\rangle, \tag{3.1}$$

where $|n\rangle$ is a base state and f is a one-to-one Boolean function. The example of quantum function is presented in Listing 3.2.

[2] Quantum function are also called pseudo-classic operators.

Quantum conditions

QCL introduces quantum conditional statemens, ie. conditional constructions where quantum state can be used as a condition.

QCL, as well as many classical programming languages, provides the conditional construction of the form

```
if be then
  block
```

where *be* is a Boolean expression and *block* is a sequence of statements.

QCL provides the means for using quantum variables as conditions. Instead of a classical Boolean variable, the variable used in condition can be a quantum register.

```
qureg a[2];
qureg b[2];
// the sequence of statements
// ...
// perform CNot if a=11
if a {
  CNot(b[0], b[1]);
}
```

In this situation QCL interpreter builds and executes sequence of $CNOT$ gates equivalent to the above condition. Here register a is called enable register.

In addition, quantum conditional structures can be used in quantum subroutines. Quantum operators and functions can be declared as conditional using cond keyword. For example

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

declares conditional Phase gate and controlled NOT gate. Keyword extern indicates that the definition of subroutine is specified in external file. The enable register (ie. quantum condition) is passed as an implicit parameter if the operator is used within the body of a quantum if-statement.

In the case of inc procedure, presented in Listing 3.2, the enable register is passed as an implicit argument. This argument is set by a quantum if-statement and transparently passed on to all suboperators. As a result, all suboperators have to be conditional. This is illustrated by the following example [93]

```
qcl¿ qureg q[4];qureg e[1];   // counting and control registers
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
```

```
cond 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
}

// equivalent implementation with constant enable register
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);
  }
}
```

Listing 3.2: Operator for incrementing quantum state in QCL defined as a conditional quantum function. Subroutine inc is defined using cond keyword and it does not require the second argument of type quconst. Subroutine cinc provides equivalent implementation with explicit-declared enable register.

```
[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¿
```

Finally we should note that a conditional subroutine can be called outside of a quantum if-statement. In such situation enable register is empty and as such ignored. Subroutine call is in this case unconditional.

3.3.2 LanQ

Imperative language LanQ is the first quantum programming language with full operation semantics specified [83].

Its main feature is the support for creating multipartite quantum protocols. LanQ, as well as cQPL presented in next section, are build with quantum communication in mind. Thus, in contrast to QCL, they provide the set of features for facilitating simulation of quantum communication.

Syntax of the LanQ programming language is very similar to the syntax of C programming language. In particular it supports:

- Classical data types: int and void.

- Conditional statements of the form

```
if ( cond ) {
  ...
} else {
  ...
}
```

- ❏ Looping with while keyword

```
while ( cond ) {
  ...
}
```

- ❏ User defined functions, for example

```
int fun( int i) {
  int res;
  ...
  return res;
}
```

Process creation

LanQ is built around the concepts of process and interprocess communication, known for example from UNIX operating system. It provides support for controlling quantum communication between many parties. The implementation of teleportation protocol presented in Listing 3.3 provides an example of LanQ features, which can be used to describe quantum communication.

Function main() in Listing 3.3 is responssible for controlling quantum computation. The execution of protocol is divided into the following steps:

1. Creation of the classical channel for communicating the results of measurement: channel[int] c withends [c0,c1];.
2. Creation of Bell state (see Appendix B) used as a quantum channel for teleporting a quantum state (psiEPR aliasfor [psi1, psi2]); this is acomplished by calling external function createEPR() creating an entangled state.
3. Instruction fork executes alice () function, which is used to implement sender; original process continues to run.
4. In the last step function bob() implementing a receiver is called.

```
void alice(channelEnd[int] c0, qbit auxTeleportState) {
  int i;
  qbit phi;
  // prepare state to be teleported
  phi = computeSomething();
  // Bell measurement
  i = measure (BellBasis, phi, auxTeleportState);
  send (c0, i);
}

void bob(channelEnd[int] c1, qbit stateToTeleportOn) {
  int i;
  i = recv(c1);
  // execute one of the Pauli gates according to the protocol
  if (i == 1) {
    Sigma˙z(stateToTeleportOn);
  } else if (i == 2) {
    Sigma˙x(stateToTeleportOn);
  } else if (i == 3) {
    Sigma˙x(stateToTeleportOn);
    Sigma˙z(stateToTeleportOn);
  }
  dump_q(stateToTeleportOn);
}

void main() {
  channel[int] c withends [c0,c1];
  qbit psi1, psi2;
  psiEPR aliasfor [psi1, psi2];

  psiEPR = createEPR();

  c = new channel[int]();
  fork alice(c0, psi1);
  bob(c1, psi2);
}
```

Listing 3.3: Teleportation protocol implemented in LanQ [82]. Functions Sigma˙x(), Sigma˙y() and Sigma˙z() are responsible for implementing Pauli matrices. Function createEPR() (not defined in the listing) creates maximally entangled state between parties — Alice and Bob. Quantum communication is possible by using state, which is stored in a global variable psiEPR. Function computeSomething() (not defined in the listing) is responsible for preparing a state to be teleported by Alice.

Communication

Communication between parties is supported by providing send and recv keywords. Communication is synchronous, ie. recv delays programme execution until there is a value received from the channel and send delays a programme run until the sent value is received.

Processes can allocate channels. It should be stressed that the notion of channels used in quantum programming is different from the one used in quantum mechanics. In quantum programming channel refers to a variable shared between processes. In quantum mechanics channel refers to any quantum operation.[3]

Another feature used in quantum communication is variable aliasing. In the teleportation protocol presented in Listing 3.3 the syntax for variable aliasing

```
qbit psi1, psi2;
psiEPR aliasfor [psi1, psi2];
```

is used to create quantum state shared among two parties.

Types

Types in LanQ are used to control the separation between classical and quantum computation. In particular they are used to prohibit copying of quantum registers. The language distinguishes two groups of variables [83, Chapter 5]:

- Duplicable or non-linear types for repsenting classical values, eg. bit, int, boolean. The value of a duplicable type can be exactly copied.
- Non-duplicable or linear types for controlling quantum memory and quantum resources, eg. qbit, qtrit channels and channel ends (see example in Listing 3.3). Types from this group do not allow for cloning [122].

One should note that quantum types defined in LanQ are mainly used to check validity of the program before its run. However, such types do not help to defined abstract operations. As a result, even simple arithmetic operations have to implemented using elementary quantum gates.

3.4 Functional quantum programming

During the last few years many quantum programming languages based on functional programming paradigm were proposed [102]. As we have point out in the Introduction, the lack of progress in creating new quantum algorithm is cause by the problems with operating on complex quantum states. Classical functional programming languages have many features which allow to clearly express algorithms [80]. In particular they allow for writing better modularised programmes than in the case of imperative programming languages [55]. This is important since this allows to debug

[3] See Appendix B.

programmes more easily and reuse software components, especially in large and complex software projects.

Quantum functional programming attempts to merger the concepts know from classical function programming with quantum mechanics. The program in functional programming language is written as a function, which is defined in terms of other functions. Classical functional programming languages contain no assignment statements, and this allows to eliminate side-effect.[4] It means that function call can have no effect other than to compute its result [55]. In particular it cannot change value of a global variable.

The first attempts to define functional quantum programming language were made by using quantum lambda calculus [115], which was based on lambda calculus. For the sake of completeness we can also point out some research on modelling quantum computation using Haskell programming language [100, 58]. However, here we focus on high-level quantum programming languages. Below we present recently proposed languages QPL and cQPL, which are based on functional paradigm. They aim to provide mechanism known from programming languages like Haskell [56] to facilitate the modelling of quantum computation and quantum communication.

3.4.1 QPL and cQPL

In [103] Quantum Programming Language (QPL) was described and in [71] its extension useful for modelling of quantum communication were proposed. This extended language was name cQPL – communication capable QPL. Since cQPL compiler is also QPL compiler, we will describe cQPL only.

The compiler for cQPL language described in [71] is built on the top of libqc simulation library used in QCL interpreter. As a result, cQPL provides some features known from QCL.

Classical elements of cQPL are very similar to classical elements of QCL and LanQ. In particular cQPL provides conditional structures and loops introduces with while keyword.

```
new int loop := 10;
while (loop > 5) do {
    print loop;
    loop := loop − 1;
};
if (loop = 3) then {
    print ”loop is equal 3”;
} else {
    print ”loop is not equal 3”;
};
```

Listing 3.4: Classical control structures in cQPL.

[4] This is true in so-called pure functional programming languages like Haskell.

Procedures

Procedures can be defined to improve modularity of programmes.

```
proc test: a:int, q:qbit {
  ...
}
```

Procedure call has to know the number of parameters returned by the procedure. If, for example, procedure test is defined as above, it is possible to gather calculated results

```
new int a1 = 0;
new int cv = 0;
new int qv = 0;
(a1) := call test(cv, qv);
```

or ignore them

```
call test(cv, qv);
```

In the first case the procedure returns the values of input variables calculated at the end of its execution.

Classical variables are passed by value ie. their value is copied. This is impossible for quantum variable, since a quantum state cannot be cloned [122]. Thus, it is also impossible to assign value of quantum variable calculated by procedure.

Note that no cloning theorem requires quantum variables to be global. This show that in quantum case it is impossible to avoid some effect known from imperative programming and typically not present in functional programming languages.

Global quantum variables are used in Listing 3.6 to create maximally entangles state in a teleportation protocol. Procedure createEPR(epr1, epr2); operates on two quantum variables (subsystems) and produces a Bell state.

Quantum elements

Quantum memory can be accessed in cQPL using variables of type qbit or qint. Basic operations on quantum registers are presented in Listing 3.5. In particular, the execution of quantum gates is performed by using $*=$ operator.

```
new qbit q1 := 0;
new qbit q2 := 1;
// execute CNOT gate on two qubits
q1, q2 *= CNot;
q1 *= Phase 0.5;
```

Listing 3.5: State initialisation and basic gates in cQPL. Data type qbit represents a single qubit.

It should be pointed out that qint data type provides only a shortcut for accessing table of qubits.

Only few elementary quantum gates are built into the language:

- ❑ Single qubit gates H, Phase and NOT implementing basic gates listed in Table 2.2 in Chapter 2.
- ❑ CNOT operator implementing controlled negation and FT(n) operator for n-qubit quantum Fourier transform.

This allows to simulate an arbitrary quantum computation. Besides, it is possible to define gates by specifying their matrix elements.

Measurement is performed using measure/then keywords and print command allows to display the value of a varaible.

```
measure a then {
  print "a is |0>";
} else {
  print "a is |1>";
};
```

In similar manner like in QCL, it is also possible to inspect the value of a state vector using dump command.

Quantum communication

The main feature of cQPL is its ability to simulate quantum communication protocols easily. The implementation of teleportation protocols in cQPL is presented in Listing 3.6.

Communicating parties are described using modules. In analogy to LanQ, cQPL introduces channels, which can be used to send quantum data. Once again we stress that notion of channels used in cQPL and LanQ is different from that used in quantum theory. Quantum mechanics introduces channels to describe allowed physical transformations, while in quantum programming they are used to describe communication links.

3.5 Summary

First we should note that languages presented in this chapter provide very similar set of basic quantum gates and allow to operate only on the arrays of qubits. Most of the gates provided by these languages correspond to the basic quantum gates presented in Chapter 2. Thus, one can conclude that the presented languages have the ability to express quantum algorithms similar to the abilities of a quantum circuit model.

The biggest advantage of quantum programming languages is their ability to use classical control structures for controlling the execution of quantum operators. This is hard to achive in quantum circuits model and it requires the introduction of non-unitary operations to this model.

```
module Alice {
    proc createEPR: a:qbit, b:qbit {
       a *= H;
       b,a *= CNot;   /* b: Control, a: Target */
    } in {
      new qbit teleport := 0;
      new qbit epr1 := 0;
      new qbit epr2 := 0;

      call createEPR(epr1, epr2);
      send epr2 to Bob;

      /* teleport: Control, epr1: Target
         (see: Figure B.1 in Appendix B)   */
      teleport, epr1 *= CNot;

      new bit m1 := 0;
      new bit m2 := 0;
      m1 := measure teleport;
      m2 := measure epr1;

      /* Transmit the classical measurement results to Bob */
      send m1, m2 to Bob;
};

module Bob {
   receive q:qbit from Alice;
   receive m1:bit, m2:bit from Bob;

   if (m1 = 1) then {
      q *= [[ 0,1,1,0 ]];   /* Apply sigma_x */
   };

   if (m2 = 1) then {
      q *= [[ 1,0,0,-1 ]];   /* Apply sigma_z */
   };

   /* The state is now teleported */
   print "Teleported state:";
   dump q;
};
```

Listing 3.6: Teleportation protocol implemented in cQPL (from [71]). Quantum circuit for this protocol is presented in Appendix B. Two parties – Alice and Bob – are described by modules. Modules in cQPL are introduced using module keyword.

In addition, LanQ and cQPL provide the syntax for clear description of communication protocols.

The syntax of presented languages resembles the syntax of popular classical programming languages like C [59] or Java [45]. As such, it can be easily mastered by programmers familiar with classical languages. Moreover, the description of quantum algorithms in quantum programming languages is better suited for people unfamiliar with the notion used in quantum mechanics.

The main disadvantage of described languages is the lack of quantum data types. The types defined in described languages are used mainly for two purposes:

- ❏ To avoid compile-time errors caused by the copying of quantum registers (cQPL and LanQ).
- ❏ Optimisation of memory management (QCL).

Both reasons are important from the simulations point of view, since they facilitate writing of correct and optimised quantum programmes. However, these features do not provide a mechanism for developing new quantum algorithms or protocols.

In the next chapter we will see how structured quantum programming can be used to develop a new model of a quantum game. This example also shows problems one encounters while using quantum programming languages. In Chapter 5 we provide the description of some research results addressing these problems. Experminental quantum programming language kulka, described in Appendix A, has been developed in attempt to test these results.

3.6 Further reading

Recent developments in the field of quantum programming are described in [102] and [113]. Gay [43] provides an extensive bibliography.

One of the first imperative programming languages was Q Language. It was described in [12, 11]. As it was implemented as an extension of C++ programming language, it is relatively easy to use it and integrate it with existing software. It provides classes for basic quantum operations like QHadamard, QFourier, QNot, QSwap, which are derived from the base class Qop. New operators can be defined using C++ class mechanism.

Research in functional quantum programming languages started by introducing quantum lambda calculus [115]. It was introduced in a form of simulation library for Scheme programming language.

Due to the large amount of resources needed to execute quantum programmes (ie. simulate quantum computation) it is reasonable to provide parallel simulator. In particular [44] describes initial parallel version of the simulation library used in QCL interpreter.

Chapter 4

Application in quantum game theory

In this chapter we describe the application of quantum programming in the quantum game theory. The results presented in this chapter are based on work described in [78] and [67].

We start by introducting basic facts from game theory. We provide an example of Prisoner's dilemma and Parrondo's paradox. Next, we show how quantum games can be constructed using the example of quantum Prisoner's dilemma. This was one of the first quantum games proposed. In spite of its simplicity it shows the impact of quantum mechanics on game theory.

Quantum programming is presented as the method of developing the quantum version of Parrondo's paradox. We present the implementation of this paradox in QCL quantum programming language. We also provide the results of simulations indicating that the presented construction does have the properties know from the classical version.

The presented implementation is also discussed from the quantum programming point of view. We argue that some elements of the proposed Parrondo's scheme could be described using quantum data types. This shows potential applications of the results presented in Chapter 5.

4.1 Introduction

Game theory is used to describe the situations of mutual interaction of several parties, where each party aims to maximise its gain [99]. Parties are usually called players and they are free to choose among allowed moves in order to maximise payoff function.

Classical game theory is based on two assumptions:

- ❑ Players can choose from the set of well-defined strategies.
- ❑ Payoff function is defined for any choice of strategies.

However, if we consider the physical system used to define a game, it is reasonable to ask the question about the influence of physical laws on games. This motivation resembles one used to formulate Deutsch–Church–Turing hypothesis in Chapter 2.

As a result, we should consider more general notion of a game, where strategies are defined in terms of quantum evolution. In such case the payoff function must be defined in terms of quantum observable on the spaces of all strategies. The players in quantum games can use larger space of strategies and this should influence the expected value of the payoff.

Formally, the classical game can be defined as follows.

Definition 4.1 (Strategic form of a game [36, 98]) A classical N-party game is the triple (A, S, f), where

- ❑ $A = \{A_1, A_2, \ldots, A_N\}$ is the set of players,
- ❑ $S = S_1 \times \cdots \times S_N$, S_j is the strategy space of the j-th player for $j = 1, \ldots, N$,
- ❑ $f : S \to \mathbb{R}^N$ is the payoff function.

A strategy is a rule that prescribes the action of a player upon the game situation. Strategy can be pure, in which case it specifies a unique move in a given configuration, or mixed, in which case players use a randomising device to select among alternatives for some or all configurations.

In this chapter we give examples of 1-party games and 2-party games, ie. games with $A = \{A_1\}$ and $A = \{A_1, A_2\}$. In particular, Parrondo's game discussed in this chapter is the example of 1-party (or 1-player) game. Such games are sometimes referred to as games agains nature [94]. We start, however, by introducing a well known example of classical game, namely Prisoner's dilemma.

4.1.1 Prisoner's dilemma

Classical game described as Prisoner's dilemma is the example of two-party (two players) game. In this scenario two players (suspects) — Alice and Bob — are interrogated by a prosecutor. He offers them separately to pass the offence if they provide evidence against each other. We describe player's moves as cooperation (C) of defection (D). In the first case this means that a player decides not to provide the evidence against his partner. In the second case a player decides to provide the evidence. Depending on their decision, players can be put in the jail for the different period of time. Payoff function, presented in Table 4.1.1, defines the reductions of the verdict (in years).

Using Definition 4.1 we can describe this situation by

- $A = \{Alice, Bob\}$,
- $S = \{(C, C), (D, D), (C, D), (D, C)\}$,
- Payoff function for this game is given in Table 4.1.1.

	Bob: C	Bob: D
Alice: C	(3,3)	(0,5)
Alice: D	(5,0)	(1,1)

Table 4.1: Payoff function for the Prisoner's dilemma game for each player (Alice, Bob) either cooperating (C) or defecting (D). The number in parenthesis represent some kind of units, which can be used to describe the reward for each player. For example, if Alice decides to defeat (D) and Bob decides to cooperate (C), they will have their verdicts reduced by 5 and 0 years, respectively.

To explain why the above game is described as dilemma, we introduce two important terms used in game theory.

We say that the combination of strategies is Nash equilibrium, if no player can improve his payoff by unilateral change of strategy. We say that the combination of strategies is Pareto optimal, if no player can increase his gain (the value of payoff function) without decreasing the gain of other players.

Clearly, in the game described above Nash equilibrium is formed by the combination of strategies (D, D). On the other hand, combination (C, C) is Pareto optimal. As the result a dilemma arises: one player cannot obtain higher gain without reducing the gain of the other player and, at the same time, he cannot increase his gain by changing his strategy only. On average, it is better for both players to defeat. Dilemma is solved by allowing for communication between the players, since in such case they can both use strategy C to maximise their gain.

4.1.2 Classical version of Parrondo's game

Before discussing the quantum version of the Parrondo's game we present its classical version. In Section 4.3 we will introduce the quantum implementation of this scheme.

Classical Parrondo's game is an example of 1-player game. Parrondo's scheme is constructed by using two games and combining them into a sequence. Each game can be interpreted as a toss of an asymmetrical coin. Every success means that the player gains one dollar, every loss means that the player loses one dollar.

Definition 4.2 (Parrondo's game [96, 67]) Let $S = \{\mathbb{A}, \mathbb{B}\}$ be the strategy space; the strategies consist of coin tosses with different coins. The Parrondo's game is given by the sequence $\{s_n \in S : n = 1, 2 \dots\}$ and the capital sequence $\{c_n \in \mathbb{Z} : n = 0, 1, \dots\}$, where c_0 is given and $c_{n+1} = c_n \pm 1$ depending on whether the outcome of s_n is "heads up" ($\equiv s_n > 0$) or not. The probability that $s_n > 0$ is

$$\mathrm{Prob}(s_n > 0) = \begin{cases} p & s_n = A \\ p_0 & s_n = B \wedge 3 \mid c_n \\ p_1 & \text{otherwise} \end{cases} \tag{4.1}$$

The Parrondo's game is called winning if

$$\bigvee_{n_0, \epsilon} \bigwedge_{n > n_0} c_n \geq \epsilon > 0 \tag{4.2}$$

and losing if

$$\bigvee_{n_0, \epsilon} \bigwedge_{n > n_0} c_n \leq \epsilon < 0. \tag{4.3}$$

We will refer to the presented game as Parrondo's game or Parrondo's scheme.

Let us assume that the first game $\mathbb{A}$ has the probability of winning $1/2 - \epsilon$. The second game $\mathbb{B}$ depends on the amount of capital accumulated by a player.

If his capital is a multiple of three, the player tosses coin B_1, which has probability of wining $1/10 - \epsilon$, otherwise the player tosses coin B_2 which has probability of wining $3/4 - \epsilon$. Originally $\epsilon = 0.005$, but generally it can be any small real number.

Both games $\mathbb{A}$ and $\mathbb{B}$ are biased and have negative expected gain. But when a player has the option to choose which game he wants to play at each step of the sequence, he can choose such a combination of games which allows him to obtain positive expected gain.

It is known that sequences $(\mathbb{ABBAB})+$ or $(\mathbb{AABB})+$ give relatively high expected gain. This fact is known as Parrondo's paradox.

4.2 Quantum games

Quantum game theory [38, 97] has its roots in both game theory and quantum information theory. The investigation of different quantum games may bring new insight into the development of quantum algorithms and provide new methods of designing quantum algorithms. Quantum games can be also used to describe decoherence in quantum computers. Such games fit naturally in

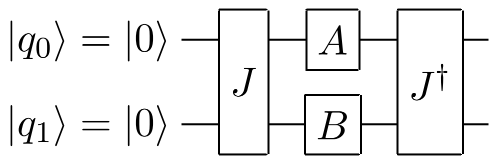


Figure 4.1: Protocol for two-person quantum game [38]. Gate J is used to introduce an entangled state. In the case of Prisoner's dilemma it is used to produce initial maximally entangled state.

the scheme of 1-player games. Thus, in the following section we will study Parrondo's game, which is an example of such scheme.

The first example of a quantum game was provided in the form of simple penny flip game [72]. In this scenario a player with the ability to use quantum strategies (moves) can always win with a player using only classical moves.

Eisert et al. [34] provided more elaborated example of two-person quantum game — quantum Prisoner's dilemma. Below we briefly describe this scheme. We start by introducing some definitions.

Quantum game theory introduces the description of physical system to the definition of a game. Following Meyer [38] we will define a quantum game as follows.

Definition 4.3 (Quantum game [38]) Quantum N-party game is a tuple $(\mathcal{H}, \rho, A, S, f)$, where

- $\mathcal{H}$ is a Hilbert space,
- $\rho \in \mathcal{S}(\mathcal{H})$ is the initial state,
- $A = \{A_1, A_2, \ldots, A_N\}$ is the set of players,
- $S = S_1 \times \cdots \times S_N$, S_j is the strategy space of the j-th player for $j = 1, \ldots, N$,
- $f : S \to \mathbb{R}^N$ is the payoff function.

General protocol for two-person quantum game is presented in Figure 4.1. Gate J is used to introduce an entangled state shared by the players. In particular, this gate can be used to study the behaviour of quantum games when the initial state is not maximally entangled [31]. However, we will neglect this gate and, in the case of quantum Prisoner's dilemma, use entangled initial state instead. In the case of Parrondo's game we don't need to introduce entanglement since this game is a single-player game.

4.2.1 Quantum Prisoner's dilemma

The quantum version of Prisoner's dilemma was introduced in [34]. The original scheme for Quantum Prisoner's dilemma is constructed using the following settings:

- ❏ Hilbert space of two qubits $\mathcal{H} = \mathbb{C}^2 \otimes \mathbb{C}^2$,
- ❏ maximally entangled initial state $\rho = |\phi\rangle\langle\phi|$, where $|\phi\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + i|11\rangle\right)$,
- ❏ set of strategies consisting of the subset of $U(2)$,
- ❏ payoff function calculated using measurement operators
$$\{\pi_{CC}, \pi_{CD}, \pi_{DC}, \pi_{DD}\}, \tag{4.4}$$
defined as
$$\begin{aligned} \pi_{CC} = |\phi_{CC}\rangle\langle\phi_{CC}|,\ |\phi_{CC}\rangle &= 1/\sqrt{2}\left(|00\rangle + i|11\rangle\right), \\ \pi_{CD} = |\phi_{CD}\rangle\langle\phi_{CD}|,\ |\phi_{CD}\rangle &= 1/\sqrt{2}\left(|01\rangle - i|10\rangle\right), \\ \pi_{DC} = |\phi_{DC}\rangle\langle\phi_{DC}|,\ |\phi_{DC}\rangle &= 1/\sqrt{2}\left(|10\rangle - i|01\rangle\right), \\ \pi_{DD} = |\phi_{DD}\rangle\langle\phi_{DD}|,\ |\phi_{DD}\rangle &= 1/\sqrt{2}\left(|11\rangle + i|00\rangle\right). \end{aligned}$$

In this scheme the classical strategies C and D correspond to matrices
$$C \sim \left(\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right) \text{ and } D \sim \left(\begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array}\right). \tag{4.5}$$

Payoff is calculated by performing the measurement on the final state $\sigma = (S_A \otimes S_B)\rho(S_A \otimes S_B)^\dagger$. For example, if the measured state was $|\phi_{DC}\rangle$ and Alice receives payoff A_{DC} and Bob – B_{DC}, specified in Table 4.1.1.

Average payoff functions for Alice and Bob read
$$P_A(s_A, s_B) = \sum_{x,y\in\{C,D\}} A_{xy}\mathrm{tr}\left[\pi_{xy}\sigma\right], \tag{4.6}$$
$$P_B(s_A, s_B) = \sum_{x,y\in\{C,D\}} B_{xy}\mathrm{tr}\left[\pi_{xy}\sigma\right]. \tag{4.7}$$

It can be shown [34, 38] that the described setting allows for solving the dilemma observed in the classical case. We can restrict allowed strategies to unitary strategies of the form (see Theorem 2.6 in Chapter 2)
$$U(\theta, \psi) = \left(\begin{array}{cc} e^{i\phi}\cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & e^{-i\phi}\cos(\theta/2) \end{array}\right). \tag{4.8}$$

In this case strategy (Q, Q) with matrix
$$Q = U(0, \pi/2) = i\sigma_z = \left(\begin{array}{cc} i & 0 \\ 0 & -i \end{array}\right) \tag{4.9}$$
gives the expected payoff $P_A(Q, Q) = P_B(Q, Q) = 3$ [34, 33], which is Pareto optimal. This strategy also gives Nash equilibrium, and thus solves the dilemma observed in the classical case.

4.3 Quantum implementation of Parrondo's game

The previous example shows the impact of quantum mechanics on quantum games. One should note that there are many methods for quantisation of a given game. Quantum version can be changed by, for example, changing allowed set of strategies.

In this section an example of one-player quantum games is provieded. It is based on classical Parrondo's game described in Section 4.1.2.

We present the implementation of the Parrondo's game in QCL quantum programming language. The choice of QCL for this implementation was motivated by its support for user-defined quantum operators and by the fact that the standard library for this language provides many usefoul subroutines. Another reason for choosing this language was the execution speed of the interpreter, which is implemented in C++.

We start by specifying the elements of the proposed scheme in term of quantum circuits.

4.3.1 Elements of the scheme

According to Definition 4.3 it is necessary to specify a Hilbert space, which is used to defined the presented scheme. The quantum register used to perform this scheme consists of three subregisters:

- ❏ $|c\rangle$ – one-qubit register representing the coin,
- ❏ $|\$\rangle$ – n-qubit register storing player's capital,
- ❏ $|o\rangle$ – three-qubit auxiliary register used to store results of internal calculations.

As the results, we will operate on the Hilbert space $\mathcal{H}_c \otimes \mathcal{H}_\$ \otimes \mathcal{H}_o$. Dimensions of $\mathcal{H}_c$ and $\mathcal{H}_o$ are fixed and equal to 2 and 3 respectively.

Register $|c\rangle$ holds the state of the quantum coin. Gates A, B_1 and B_2, acting on this register, represent quantum coin tosses. One should note that the register $|c\rangle$ does not store the information about history of the games.

After every execution of gates A, B_1 and B_2, the state of the register $|\$\rangle$ is changed, according to the result of the quantum coin toss. This register is responsible for storing player's capital.

Register $|o\rangle$ is an auxiliary register, which is used to check if the state of the $|\$\rangle$ register is a multiple of three. At the beginning of the scheme and after the application of the games' gates this register is always set to $|000\rangle$.

The presented scheme is defined using the following elements:

- ❏ $\{\delta_A, \alpha_A, \beta_A, \theta_A, \delta_{B_1}, \alpha_{B_1}, \beta_{B_1}, \theta_{B_1}, \delta_{B_2}, \alpha_{B_2}, \beta_{B_2}, \theta_{B_2}\}$ – the set of parameters for constructing gates A, B_1 and B_2 according to theorem 2.6 (see Table 4.2 for parameters used in the simulation),

- ❑ s_n – the strategy of the player, which in this case is composed of the sequence of games $\mathbb{A}$, $\mathbb{B}$; games $\mathbb{A}$ and $\mathbb{B}$ are constructed using gates A, B_1, and B_2 described below,
- ❑ N – the number of qubits used to store the value of capital in the register $|\$\rangle$ (outcome register); it has to be chosen depending on the number of iterations we wish to study (see below);
- ❑ offset – initial capital offset, which is used to study the influence of the initial state on the behaviour of strategies.

Games $\mathbb{A}$ and $\mathbb{B}$ are constructed using three one-qubit quantum gates A, B_1 and B_2. Each gate is described by four real parameters according to the decomposition from Theorem 2.6 in Chapter 2.

Games $\mathbb{A}$ and $\mathbb{B}$ are implemented using the conditional incrementation/decrementation (CID) gate and gates A, B_1 and B_2 described below. Gate CID is responsible for controlling the conditional execution of elementary gates. In addition, game $\mathbb{B}$ uses the gate $mod3$. Quantum circuits for implementing these gates are presented in Figure 4.2.

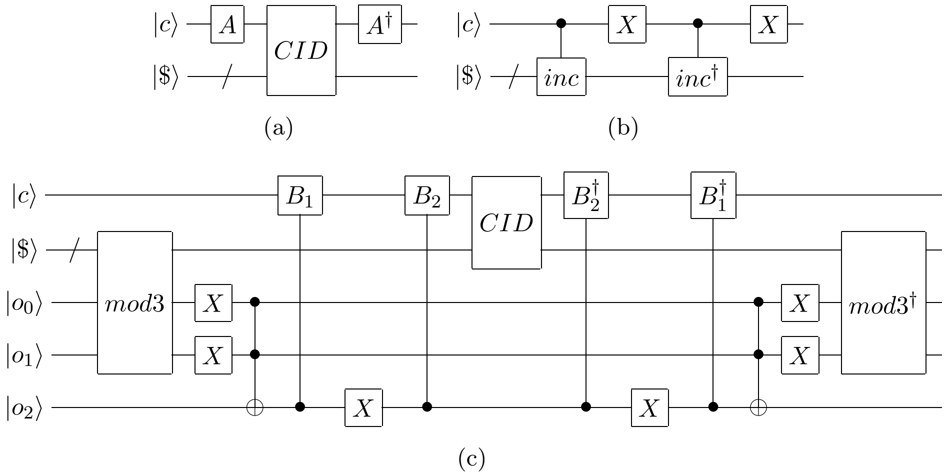


Figure 4.2: Gates used to implement Parrondo's game. (a) Circuit for the game $\mathbb{A}$. See Listing 4.2 for implementation in QCL (b) Conditional incrementation/decrementation (CID) circuit. Its QCL implementation is presented in Listing 4.1 (c) Circuit for the game $\mathbb{B}$. See Listing 4.3 for implementation in QCL

Gate $mod3$ sets $|o_1\rangle$ and $|o_2\rangle$ registers to state $|1\rangle$ if the $|\$\rangle$ register contains a number that is a multiple of three (see Definition 4.2):

$$mod3|x\rangle|0\rangle = |x\rangle|x \pmod 3\rangle. \tag{4.10}$$

It is implemented using single-qubit operations, since it is impossible in QCL to store integer numbers in quantum registers.

The CID gate is responsible for increasing and decreasing the player's capital. The circuit for this gate is presented in Figure 4.2(b). This gate increments register $|\$\rangle$ if $|c\rangle$ is in state $|1\rangle$ and decrements if it is in state $|0\rangle$.

$$CID|\$\rangle|c\rangle = \left\{ \begin{array}{l} |\$+1\rangle|c\rangle \text{ if } |c\rangle = |0\rangle \\ |\$-1\rangle|c\rangle \text{ if } |c\rangle = |1\rangle \end{array} \right. \tag{4.11}$$

One should note that this operation can be easily translated to the quantum conditional structure in QCL described in Section 3.3.1. This fact is used in its QCL implementation presented in Listing 4.1.

```
operator CID(qureg p, quconst c) {
  // if c==1 p=p+1 else p=p-1
  if c {
    inc(p);
  } else {
    !inc(p);
  }
}
```

Listing 4.1: The implementation of CID gate used in Parrondo's scheme. Operator inc is implemented in QCL standard library. The implementation of this procedure is presented in Listing 3.2 in Chapter 3.

Game $\mathbb{A}$ is implemented directly by gate A as presented in Figure 4.2(a). Game $\mathbb{B}$, presented in Figure 4.2(c), is more complicated. It uses gate $mod3$ to check if the player's capital is a multiple of three. If it is the case, gate B_2 is applied to register $|c\rangle$, otherwise, B_1 is applied.

One can easily check that all gates used in this scheme are unitary because they are composed of elementary unitary operations.

To obtain Parrondo's paradoxical effect we need to combine games $\mathbb{A}$ and $\mathbb{B}$ into some sequence s_n. The initial state of the system is prepared as follows:

1. Preparation of $|c\rangle$ in state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

2. Preparation of $|\$\rangle$ in state $|(2^{(n-1)} + \text{offset})\rangle$, where offset is a small integer number.

3. Preparation of the auxilary register $|o\rangle = |o_1o_2o_3\rangle$ in the state $|000\rangle$.

Parrondo's game consists of the application of games $\mathbb{A}$ and $\mathbb{B}$ in some chosen order s_n on the state prepared according to the above description.

Pseudocode describing the execution of Parrondo scheme is presented in Listing 4.4.

After each application of gate $\mathbb{A}$ or $\mathbb{B}$ the number stored in register $|\$\rangle$ is either incremented or decremented. The initial state of register $|\$\rangle$ must be chosen in such way that integer overflow is avoided. The maximum number of elementary games cannot exceed the capacity of the register $|\$\rangle$. Using N qubits for $|\$\rangle$ register it is possible to perform 2^{N-1} elementary games or to perform $\frac{1}{n}2^{N-1}$ Parrondo's games with strategy of length n.

If the presented scheme is implemented on a physical quantum device it should be finalised by measurement. This would give a single outcome representing the final capital. Thus, to obtain the expected gain, the experiment should be repeated several times. Note that the payoff must be calculated as the difference between initial capital and actual capital.

Simulation allows to observe the state vector of the quantum system. Using this property the expected gain is calculated as the average value of σ_z in state

$$|\$\rangle\langle\$| = \mathrm{tr}_{|c\rangle\otimes|o\rangle}\left[|c,\$,o\rangle\langle c,\$,o|\right], \tag{4.12}$$

obtained after tracing out the register with respect to coin and auxiliary subregisters

$$\langle\$\rangle = \mathrm{tr}\left[\sigma_z^{\otimes n}|\$\rangle\langle\$|\right]. \tag{4.13}$$

In physical implementation one has to repeat the experiment to calculate the expected payoff.

4.3.2 Simulation results

As we have seen in Chapters 2 and 3, quantum pseudocode is, in many cases, very similar to existing quantum programming languages. Here we can use this observation to implement classical control used to control a quantum game (see Listing 4.4).

Listing 4.5 contains subrouties controlling the execution of described Parrondo's scheme. It includes external files with the definition of quantum gates (presented in Listings 4.1, 4.2 and 4.3) and parameters used in simulation (Listing 4.6).

Table 4.2 contains the parameters used in the simulation of the presented scheme. These parameters are used to define quantum gates used in the scheme, as well as for controlling classical computation. Parameters of the scheme can be controlled using a single QCL file (see Listing 4.6), which is included in other files.

First, we note that games $\mathbb{A}$ and $\mathbb{B}$ gave, as expected, negative expected value of payoff. The behaviour of Parrondo's game composed only of game $\mathbb{A}$ or only of game $\mathbb{B}$ is presented in Figure 4.3. One can see that game $\mathbb{B}$ gives worse results (ie. larger negative expected payoff) than game $\mathbb{A}$.

δ_A	α_A	β_A	θ_A
0	1	0	$2(\frac{\pi}{2}+0.01)$

δ_{B_1}	α_{B_1}	β_{B_1}	θ_{B_1}
0	1	0	$2(\frac{\pi}{10}+0.01)$

δ_{B_2}	α_{B_2}	β_{B_2}	θ_{B_2}
0	1	0	$2(\frac{3\pi}{4}+0.01)$

Table 4.2: Parameters used in the simulation of Parrondo's scheme. Note that δ_A, δ_{B_1} and δ_{B_2} are set to 0. The behaviour of games $\mathbb{A}$ and $\mathbb{B}$ is controlled by parameters θ_A, θ_{B_1} and θ_{B_2} mainly.

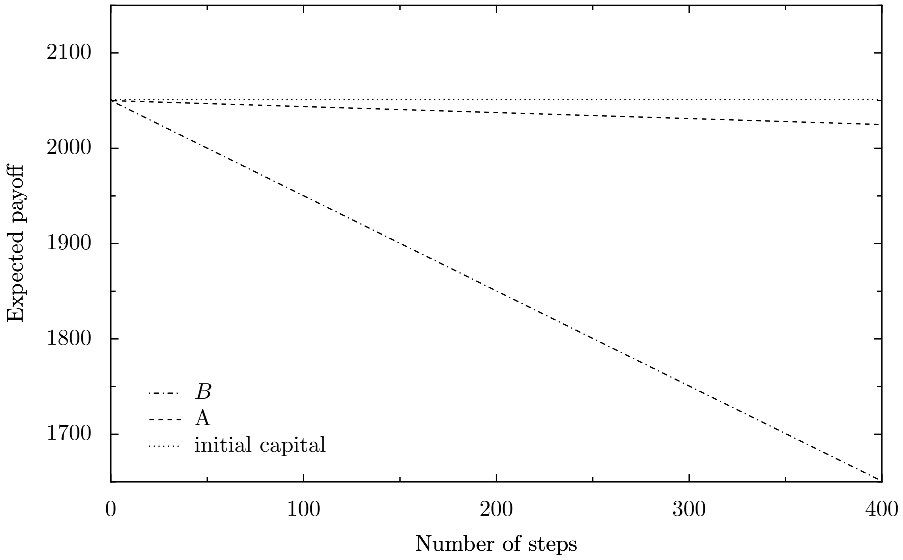


Figure 4.3: Parrondo's game composed only of game $\mathbb{A}$ or only of game $\mathbb{B}$. By playing game $\mathbb{B}$ only a player obtains bigger negative expected payoff. In this case the sequence of games is composed of one game. As a result it is possible to perform 2^{11} steps using 12 qubits.

As in classical case the combination of two losing games leads to the game of different behaviour. First of all, one can observer that by using strategies $\mathbb{AB}$ or $\mathbb{BA}$ we can change the expected payoff dramatically. One can observe that, by combining games, we can obtain much better results than by using only one of the games. The comparison of strategies $\mathbb{AB}$ and $\mathbb{BA}$ is presented in Figure 4.4. There exist four strategies of the length two. Obtained game is losing, but the average payoff is higher than in the case presented in Figure 4.3.

Classical results suggest that one needs to study more complicated strategies to obtain more interesting situations. This also shows that it is necessary

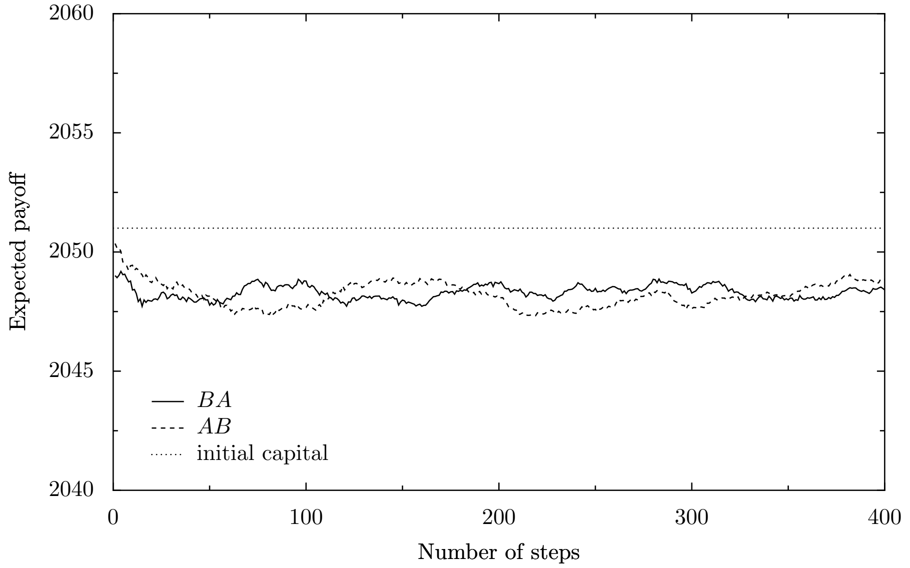


Figure 4.4: The comparison of strategies $\mathbb{AB}$ and $\mathbb{BA}$. One can note that, by using simple combination of initial games, a player can obtain the significant gain when compared to the strategy with one game only (see Figure 4.3). Note that the number of steps refers to number of sequence s_n execution.

to use simulation since the number of strategies of the length n (composed of n elementary games $\mathbb{A}$ and $\mathbb{B}$) is equal to 2^n.

Figure 4.5 presents the comparison of strategies of length 3. There exists 8 such strategies. The presented results show that it is possible to obtain paradoxical behaviour even for such a short sequence of games. It is suprising that strategy $\mathbb{ABB}$ gives the positive expected payoff, since it is composed of two $\mathbb{B}$ games, which gave worse results. One should also note that a small change in strategy alters the behaviour of the game — strategy $\mathbb{AAB}$ is losing. As the reslt, one can conclude that the descibed quantum game is very sensitive for anych changes or errors.

Simulation results for strategies composed of 5 games are presented in Figure 4.6. In this case we have $2^5 = 32$ possible strategies. In addition we can control the behaviour of the Parrondo's scheme by changing the initial state. The strategies in Figure 4.6 were found to be the best winning strategies for different values of an offset parameter.

Another interesting feature of the presented scheme is the dependency of its behaviour on the initial state. First, we note that the behaviour of strategies $\mathbb{A}$ and $\mathbb{B}$ does not depend on an initial offset. On the other hand, the initial state, controlled by the parameter offset, can influence obtained results. In Figure 4.7 one can observe that, depending on the initial offset, the strategy can change its behaviour completely. In this case strategy $\mathbb{BBABA}$ is losing for *offset* $= 0$ or *offset* $= 2$, but it gives positive expected gain for *offset* $= 1$. This can be explained by the fact that the presented scheme is very sensitive to the initial conditions. We have already observed similar effect in the case of strategies $\mathbb{ABB}$ and $\mathbb{AAB}$, presented in Figure 4.5.

4.3.3 Discussion

The presented scheme allows to study the behaviour of Parrondo's game using a relatively small number of qubits. The simulation of Parrondo's scheme composed of k games requires approximately $O(\log(nk))$ qubits, where n is the number of simulation steps. This allows us to study the behaviour of proposed scheme for large number of games played. This is significant improvement to the previously proposed schemes [40, 39], which require a linear number of qubits.

Using simulation it was possible to find winning strategies. This shows that the proposed construction has properties of classical Parrondo's scheme. Nevertheless, for the given set of initial parameters, it is not common to find a winning strategy. This suggests that paradoxical behaviour is hard to achieve.

The initial value of the payoff register $|\$\rangle$ can change the behaviour of strategy. The initial value of the payoff is controlled using offset parameter. For example strategy $\mathbb{BBABA}$ is losing for *offset* $= 0$ or *offset* $= 2$, but it gives positive expected gain for *offset* $= 1$. The behaviour of this strategy

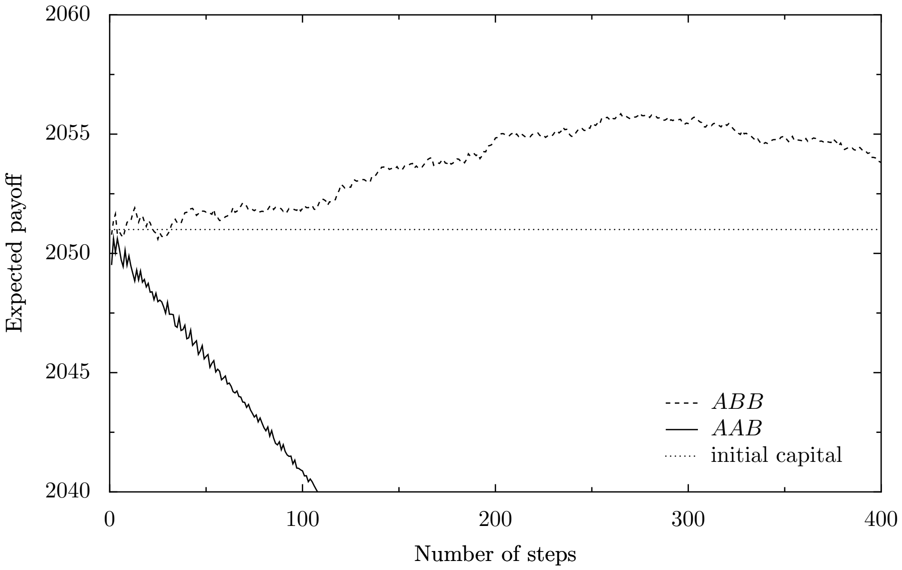
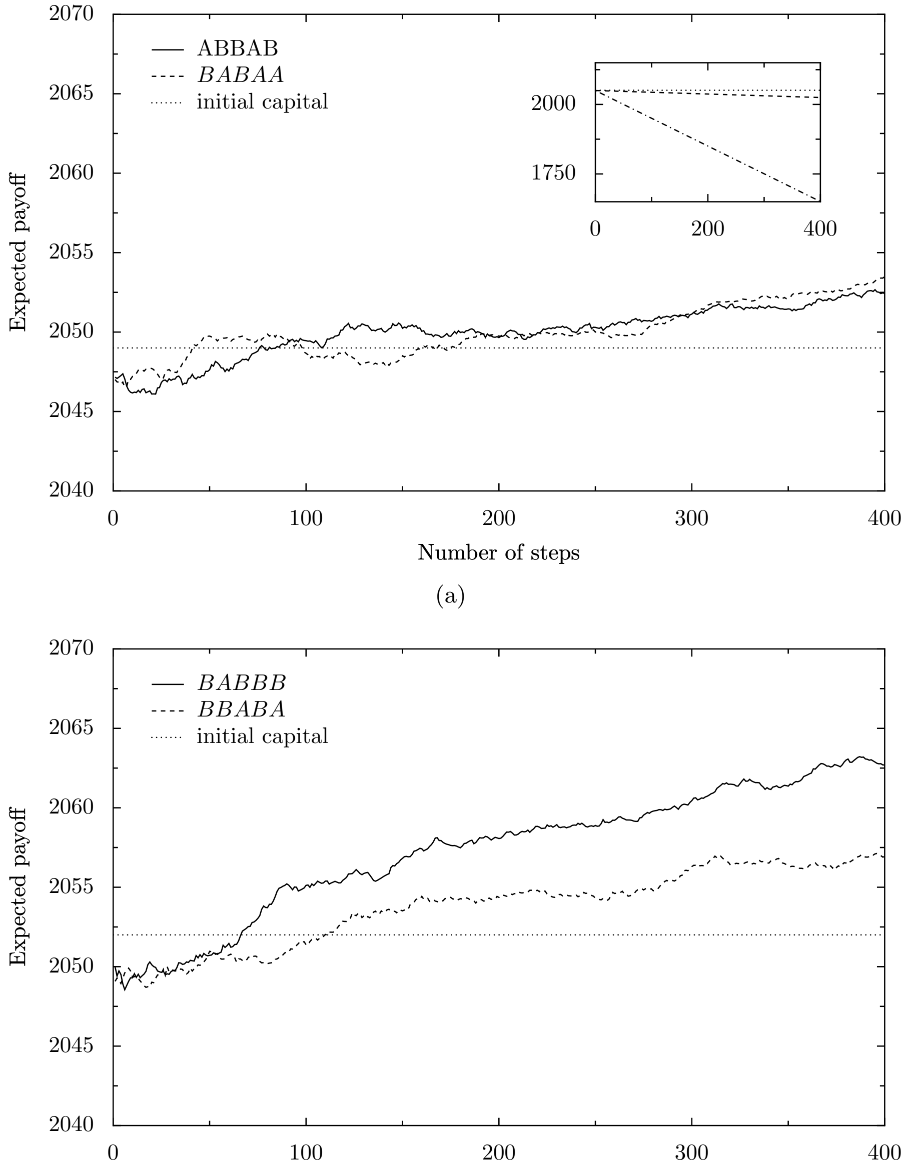


Figure 4.5: The comparison of strategies $\mathbb{ABB}$ and $\mathbb{AAB}$. This plot shows that it is possible to obtain Parrondo's effect for relatively short strategy (the sequence of games $\mathbb{A}$ and $\mathbb{B}$) used in Parrondo's scheme.



(a)

(b)

Figure 4.6: Influence of the initial state changes on strategies. See also Figure 4.7. (a) The comparison of two winning strategies for offset = 0. The values for strategies $\mathbb{A}$ and $\mathbb{B}$ are presented in an internal figure (see also Figure 4.3). (b) The comparison of two best-found winning strategies of the length 5 for offset = 3.

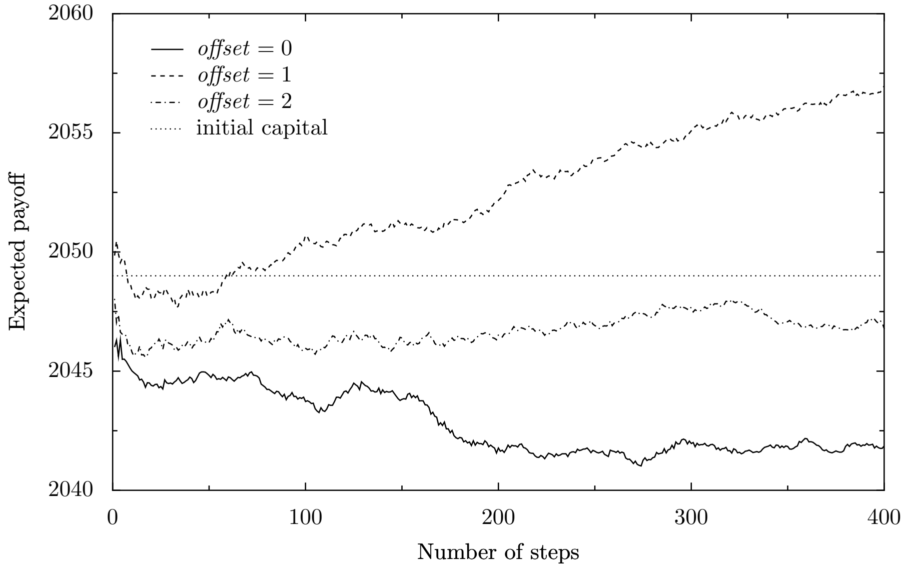


Figure 4.7: Influence of the initial state on the strategy $\mathbb{BBABA}$. This strategy $\mathbb{BBABA}$ is losing for *offset* $= 0$ or *offset* $= 2$, but it gives positive expected gain for *offset* $= 1$. In this case initial state was set to $|2048 + \mathit{offset}\rangle$ and the game was played using 12 qubits.

for different offsets is presented in Figure 4.7

4.4 Summarry

We have presented the implementation of quantum Parrondo's scheme, which is the example of a single-player quantum game. The presented model was developed using QCL quantum programming language. As a result, we have shown how quantum programming can be used to design quantum algorithms. The presented construction exploits the main advantages of existing quantum programming languages. In particular we have used:

- ❏ quantum subprocedures for defining basic structures (see Listings 4.2 and 4.3),
- ❏ conditional structures based on quantum registers to implement parts of the scheme (see Listing 4.1),
- ❏ classical control structures to prepare simulation of the scheme (see Listing 4.6).

The main problem, which can be observed in the presented implementation, is the lack of mechanism for managing quantum memory representing data other than simple qubits. We have used mechanism built on the top of QCL to control the behaviour of games. In particular modulo arithmetic has to be implemented using single-qubit operations. One has to use elementary operations like $CNOT$ and NOT to calculate simple functions. This is suprising since such procedures are implemented as a part of the syntax in many modern programming languages.

Also unitary circuits for basic arithmetic operations were proposed [117] and some research was made to use quantum computers to implement arithmetics [29]. This suggests that quantum data type for operating on integers can be useful in developing new quantum algorithms or creating quantum versions of existing classical algorithms.

In the next chapter we use this motivation to develop basic data types, which can be used to operate on integer numbers using quantum registers. Experimental quantum programming language kulka, described in Appendix A, was created to test these data types.

```
operator Ph(real alpha, qureg r) {
  Matrix2x2(E^(I*alpha),0,0,E^(I*alpha),r);
}

operator Ai(real de, real al, real th, real be, qureg c) {
  Ph(de,c);
  RotZ(al,c);
  RotY(th,c);
  RotZ(be,c);
}

operator A(qureg p, qureg c) {
  //p - payoff, c - coin
  Ai(Ade,Aal,Ath,Abe,c);
  CID(p,c);
  !Ai(Ade,Aal,Ath,Abe,c);
}
```

Listing 4.2: Game $\mathbb{A}$ used in Parrondo's scheme. This game is defined in terms of operator Ai(), which represents a biased coin toss. Operator Ph implements a phase shift gate (see Chapter 2 and [93]).

```
operator B(qureg p, qureg c, quvoid s, quvoid o) {
  muln(1, 3, p, s); //s<-p*1 mod 3
  Not(s);
  CNot(o,s);

  Bi(B1de,B1al,B1th,B1be,c,o); // B1
  Not(o);
  Bi(B2de,B2al,B2th,B2be,c,o); // B2

  CID(p,c);

  !Bi(B2de,B2al,B2th,B2be,c,o); // !B2
  Not(o);
  !Bi(B1de,B1al,B1th,B1be,c,o); // !B1

  CNot(o,s);
  Not(s);//
  !muln(1, 3, p, s);
}
```

Listing 4.3: Game $\mathbb{B}$ used in Parrondo's scheme. The execution of game is based on the current capital of a player. Register $|o\rangle$ is in the state $|111\rangle$ if the current value of player's capital is divided by 3. Operator Bi() (not shown) is defined, in analogy to operator Ai(), by using elementary rotations.

```
Procedure: Parrondo($\underline{x}, s_n, k, l$)
Input: Quantum register $\underline{x}$ with $N$ qubits, strategy $s_n$, initial offset $k$ and number of simulation steps $l$.
Output: Payoff $p$.
C: calculate the initial state of payoff register
$x \leftarrow 2^{N-1} + k$
C: prepare initial state
$\underline{c} \leftarrow H(\underline{0})$
$\underline{o} \leftarrow 000$
$\underline{\$} \leftarrow x + k$
C: perform sequence of games
for $i = 1$ to $i = l$
  $s_n(\underline{\$}, \underline{c}, \underline{o})$
C: perform final measurement and store its outcome in $p$
$p \leftarrow \underline{\$}$
C: calculate the gain
$p \leftarrow p - x$
```

Listing 4.4: Parrondo's scheme described in quantum pseudocode. Classical control structures used in this listing are very similar to the structures used in Listing 4.5.

```
// include required external files
<<modarith; // standard library
<<examples; // standard library
<<parrondo˙operators; // definition of opertators
<<data; // parameters for simulation

procedure strategy(qureg p, qureg c, qureg s, qureg o) {
  // the example of strategy
  B(p,c,s,o);
  B(p,c,s,o);
  A(p,c);
  B(p,c,s,o);
  A(p,c);
}

// classical control
qureg c[1]; // coin
qureg p[regsize]; // payoff
qureg s[2]; // used to perform operation mod3 (s <- p mod 3)
qureg o[1]; // used to controll game 𝔹 (o==1 <=> p|3)

// initial payoff is set to avoid overflow
set(2^(regsize-1)+offset,p);

// initial Hadamard gate
H(c);

int i;
for i=1 to steps {
  strategy(p, c, s, o);
  print "i=",i;
  dump p;
}
```

Listing 4.5: Classical routine controlling the execution of strategies in Parrondo' scheme. Procedure strategy() is used to encapsulate the sequence of games used by a player.

```
// File: data.qcl

// three sets of parameters for quantum gates
const epsilon = -0.01*pi;
const Ade = 0;                    // V
const Aal = 0;                    // RotZ
const Ath = 2*(pi/2-epsilon);     // RotY
const Abe = 0;                    // RotZ

const B1de = 0;
const B1al = 0;
const B1th = 2*(pi/10-epsilon);
const B1be = 0;

const B2de = 0;
const B2al = 0;
const B2th = 2*(3*pi/4-epsilon);
const B2be = 0;

// parameters for classical control
const regsize= 12;
const steps= 400;
const offset= 3;
```

Listing 4.6: Parameters used in the simulation of Parrondo's scheme. This file (data.qcl) contains the parameters listed in Table 4.2. Not all the parameters have been used to obtain the presented results, since in the presented results we neglect the global phase introduced by the phase gate.

Chapter 5

Operating on quantum data types

Quantum computation is about generating interesting probability distributions using the subtle rules of quantum mechanics. The best example of such probability engineering is the quantum algorithm for factorisation. In this chapter we aim to show how quantum computers can be used to obtain probability distributions useful for solving other problems like sorting.

We start by discussing the problem of state initialisation. This is motivated by the quantum data types introduced in the experimental quantum programming language kulka, described in Appendix A. Presented procedure may be regarded as the generalisation of classical variable initialisation problem.

The second presented algorithm is the quantum version of classical radix sort algorithm. Described algorithm is obtained by the modification of the state initialisation procedure. The results of sorting are encoded in the form of probability distribution. The time complexity of described sorting algorithm depends on complexity of read-out procedure, which consists of the series of measurements. Sorting procedure on a quantum computer is equivalent to generating appropriate probability distribution.

5.1 Motivation

As we have seen in previous chapters, quantum algorithms [46, 107] and communication protocols [9, 18] are described using the language of quantum circuits. While this method is convenient for simple algorithms, it is very difficult to operate on abstract data types using this notation.

This lack of data types and control structures motivated the development of the quantum pseudocode and quantum programming languages described in Chapters 2 and 3.

We have pointed out that existing quantum programming languages are based on mathematical formalism of quantum theory ie. state vectors or density matrices. They do not provide abstract data types for operating on quantum memory. Similar situation exists in some old programming languages like Fortran 77, where it is possible to use only the arrays of numbers and no mechanism for creating new data types is provided. In contrast, many modern programming languages are built on object-oriented paradigm (eg. Java [45]). In these languages data types are one of the main elements and they provide mechanisms for seamless creation of new data types. In fact, appropriate choice of data types is sometimes the most important decision during the implementation [101].

Definition 5.1 (Data type) A data type is a set of values along with set of operations allowed on those values.

The set of data type values is called its range.

For example in Java programming language, the type int represents the set of 32-bit integers $\{-2^{31}, \ldots, 0, \ldots, -2^{31} - 1\}$ and the set of allowed operations on these numbers like addition, subtraction and multiplication [45].

5.2 Quantum data types

In the existing quantum programming languages one needs to operate using arrays of qubits. It is impossible to create a quantum programme without using elementary quantum gates. In most cases it is possible to perform only basic operations like $CNOT$ or Hadamard gates.

The only advantage of existing programming languages over quantum circuits model is that they allow for classical control of quantum computation. But taking into account the provided level of abstraction they are very similar to specialised packages for simulating quantum computers [77].

As we have seen in Chapter 4 one needs to use elementary gates to implement arithmetic operations on data encoded in quantum registers. This is very inconvenient since many algorithms operate mainly on numbers.

Quantum programming language kulka, developed during the work on this thesis, introduces a new element when compared to existing quantum programming languages. It allows for operating on quantum data types.

Definition 5.2 (Quantum data type) A quantum data type is a classical data type which allows for operating on superposition of elements in its range.

The main advantage of quantum data types is that they allow for the automatic generation of quantum gates for typical tasks. Below we focus on the state initialisation. We also suggest the possible application of quantum data types in the implementation of the sorting procedure.

Using the presented procedure and results on quantum arithmetic described in [117] it is possible to implement quantum data types, which is an extension of the classical data type in the sense of Definition 5.2.

Quantum programming language kulka, described in Apendix A, introduce quantum data type qint. Variables of this type can be used to store integer numbers. It is possible to initialise a variable of this type to the superposition of integer numbers and to perform basic arithmetic operations using the methods described in [117]. For example

```
qint v1 = (5|10|15);
```

prepares quantum register v1 in the state

$$|v_1\rangle = \frac{1}{\sqrt{3}}(|5\rangle + |10\rangle + |15\rangle) = \frac{1}{\sqrt{3}}(|0101\rangle + |1010\rangle + |1111\rangle). \tag{5.1}$$

This operation has to be represented using quantum gates — this is one of the requirements for quantum programming language discussed in Chapter 3. Like quantum conditional structures available in QCL [93], initialisation to superposition facilitates operations on quantum memory. It has also the advantage similar to quantum conditional statements in QCL (see Chapter 3), ie. it eliminates the number of elementary quantum gates needed to write quantum programmes.

5.3 Initialisation of quantum registers

We start by discussing the initialisation problem and introducing the notation used in this chapter.

The problem of state preparation was discussed by Grover. He proposed [47] an algorithm using $O(\sqrt{N})$ steps. In contrast the algorithm presented below uses $O(\log^3 N)$ steps to prepare the superposition of N elements. The algorithm proposed in [47] uses the modification of the quantum search algorithm as the base step and allows to prepare more general class of states. General methods for constructing a quantum state were also discussed in [87]. The procedure proposed therein used $2^{n+2} - 4n - 4$ $CNOT$ gates and $2^{n+2} - 5$ single-qubit operators.

The procedures proposed in [47] and [87] are suited for preparing an arbitrary quantum pure state. In contrast, the algorithm discussed in this section allows to prepare only a uniform superposition. However, the presented algorithm is simpler and can be easily implemented. This is important since as such it can be easyly incorporated as a part of the quantum programming language interpreter. It is also much faster and does not require a quantum oracle.

In what follows we will use numbers which can be written using M bits. By $a^k_i, i = 1, 2, \ldots, M$ we denote the i-th bit of the number a^k and by the control α_i we denote the string of bits which defines projection operator $|\alpha_i\rangle\langle\alpha_i|$. We also introduce the following notation

❏ $c_k(\alpha_j)$ — the number of elements a^i, such that $a^{k-1}a^{k-2}\ldots a^1 = \alpha_j$,

❑ $t_k(\alpha_j)$ — the number of elements a^i, such that $a^{k-1}a^{k-2}\dots a^1 = \alpha_j$ and $a^i_k = 1$.

Below we describe the procedure for generating quantum gate R used to implement an initialisation algorithm [76]. Generated quantum gate acts on M-qubit Hilbert space and outputs the uniform superposition of input numbers.

As the analogue of a classical bits, qubits are described by systems with two base states (see Appendix B). We introduce gate G, which can be used to accomplish the following task:

Task 5.1 Starting from the state $|0\rangle$ prepare the superposition

$$\sqrt{1-p}|0\rangle + \sqrt{p}|1\rangle, \tag{5.2}$$

with $0 \leq p \leq 1$.

Since

$$|\langle 1|(\sqrt{1-p}|0\rangle + \sqrt{p}|1\rangle)|^2 = p, \tag{5.3}$$

p denotes here the probability of measuring the state $|1\rangle$. On the other hand the probability of measuring $|0\rangle$ in this state reads

$$|\langle 0|(\sqrt{1-p}|0\rangle + \sqrt{p}|1\rangle)|^2 = 1-p. \tag{5.4}$$

It is easy to see that the desired state can be reached by using R_y rotation [89]

$$R_y(\theta) = \left(\begin{array}{cc} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{array} \right), \tag{5.5}$$

with the parameter

$$\theta = -2\arctan\sqrt{\frac{p}{1-p}}. \tag{5.6}$$

This rotation is equivalent to

$$\begin{aligned} G^{<}(p) &= R_y\left(-2\arctan\sqrt{\frac{p}{1-p}}\right) & (5.7) \\ &= \left(\begin{array}{cc} \sqrt{1-p} & -\sqrt{p} \\ \sqrt{p} & \sqrt{1-p} \end{array} \right), & (5.8) \end{aligned}$$

and we have clear probabilistic interpretation of the parameter p as the probability of measuring the system in the state $|1\rangle$.

For $p = 1$ all we need is to perform $NOT \equiv \sigma_x$ gate. In what follows we use the quantum gate defined as

$$G(p) = \left\{ \begin{array}{ll} G^{<}(p), & 0 \leq p < 1 \\ \sigma_x, & p = 1 \end{array} \right. . \tag{5.9}$$

Since more than one qubit is needed to perform useful quantum computation, we can formulate our state initialisation task as follows

Task 5.2 For the given set of integers $A = \{a^1, a^2, \dots, a^K\} \subset \mathbb{Z}_{2^M}$ generate unitary operation $R(A)$ such that

$$R(A)|0\dots0\rangle = \sum_{i=1}^{K} |a^i\rangle. \tag{5.10}$$

Here A may be associated with the probability distribution P^A such that

$$\mathrm{P}^A(a) = \left\{ \begin{array}{ll} \frac{1}{K}, & a \in A \\ 0, & a \notin A \end{array} \right. , \tag{5.11}$$

on the $\mathbb{Z}_{2^M}$, where K is a number of elements in A.

The task defined this way can be regarded as a variant of the quantum searching algorithm [46]. In this case our goal is to prepare the probability distribution, not only to amplify the chosen probability.

One should note that the procedure defined below uses only classical data to a generate unitary evolution. As such it does not require any information about the quantum state [87].

5.3.1 Generating unitary matrix

The quantum gate, which is used to solve Task 5.2, is generated from the list of input numbers. It is convenient to divide the procedure for generation of this gate into two parts — first, we process bits on the first position and then the other bits.

We assume that the initial state of the systems is $|\phi_0\rangle = |0\dots0\rangle \in \mathbb{C}^{2^M}$, where the most significant bit of the input is encoded into the leftmost qubit.

Part 1 — the first qubit

Operation for the first qubit depends only on the information in the first bits $a_1^1, a_1^2, \dots, a_1^K$ of the input numbers $a^1, a^2, \dots, a^K$. By t_1 we denote the number of occurrence of 1 at the first position. The first operation is defined as rotation

$$R_1 = \mathbb{I}^{M-1} \otimes G(t_1/K). \tag{5.12}$$

The resulting state reads

$$\begin{aligned} |\psi_1\rangle &= R_1|\underbrace{0\dots0}_{M}\rangle && (5.13)\\ &= |\underbrace{0\dots0}_{M-1}\rangle \otimes \left(R_y \left(-2\arctan\sqrt{\frac{t_1}{K-t_1}} \right) |0\rangle \right) && (5.14)\\ &= |\underbrace{0\dots0}_{M-1}\rangle \otimes (\sqrt{1-t_1/K}|0\rangle + \sqrt{t_1/K}|1\rangle). && (5.15) \end{aligned}$$

Part 2 – qubits $2, \ldots, M$

Unitary gates to be performed on the k-th qubit depend on the information in the bits $1, \ldots, k-1$. At the k-th step of the procedure we process k-th bits of the input numbers.

Using information in $c_k(\alpha_j)$ and $t_k(\alpha_j)$ defined above we can build controlled gates

$$X_k(\alpha_j) = G\left(\frac{t_k(\alpha_j)}{c_k(\alpha_j)}\right) \otimes |\alpha_j\rangle\langle\alpha_j| + \mathbb{I} \otimes \left(\mathbb{I}^k - |\alpha_j\rangle\langle\alpha_j|\right), \tag{5.16}$$

which represent rotations in subspaces defined by the operators $|\alpha_j\rangle\langle\alpha_j|$. Here $\mathbb{I}^n$ is the identity operation on n qubits and α_j are the appropriate controls.

Operation R_k generated in the k-th step of our procedure reads

$$R_k = \mathbb{I}^{M-k} \otimes \left(\prod_{\alpha_j} X_k(\alpha_j)\right). \tag{5.17}$$

Quantum gate R, which is an output of the procedure, is defined as

$$R = \prod_{l=1}^{M} R_l. \tag{5.18}$$

It is easy to see by induction that operation R produces the appropriate output gate, required to obtain the state defined in Task 5.2. The first step affects only the first qubit and the rotation in the n-th step is performed only in the subspace characterised by the projection operators $|\alpha_j\rangle\langle\alpha_j|$.

Example

We present an example of state initialisation procedure for the input $\{1, 5, 10\}$. Thus we have:

Input: The list of three 4-bit numbers $\{1, 5, 10\}$.
Output: Quantum state $\frac{1}{\sqrt{3}}(|1\rangle + |5\rangle + |10\rangle) \in \mathbb{C}^{16}$.

We have $1 = 0001_2, 5 = 0101_2, 10 = 1010_2$ in binary form. Quantum gates generate during the procedure described above are:

k=1 $R_1 = \mathbb{I}^3 \otimes G(2/3)$

k=2 $R_2 = X_2(1)X_2(0)$ with
$X_2(1) = \mathbb{I}^2 \otimes \mathbb{I} \otimes |1\rangle\langle 1| + \mathbb{I}^3 \otimes |0\rangle\langle 0|$
$X_2(0) = \mathbb{I}^2 \otimes \sigma_x \otimes |0\rangle\langle 0| + \mathbb{I}^3 \otimes |1\rangle\langle 1|$

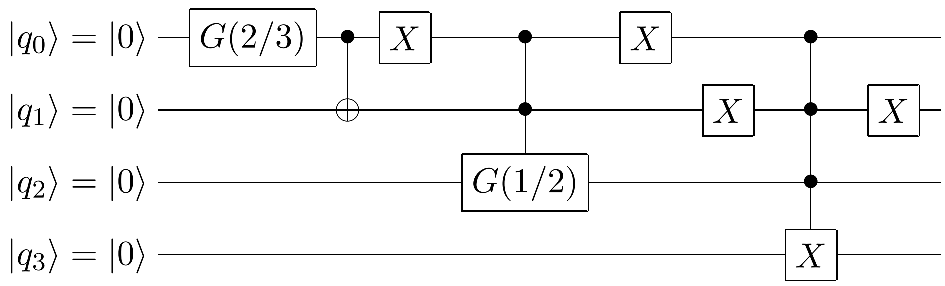

$|q_0\rangle = |0\rangle$
$|q_1\rangle = |0\rangle$
$|q_2\rangle = |0\rangle$
$|q_3\rangle = |0\rangle$
$G(2/3)$
X
X
X
X
$G(1/2)$
X

Figure 5.1: The example of register initialisation. This circuit presents final form of the initialisation procedure for input numbers $\{1, 5, 10\}$.

k=3 $R_3 = X_3(01)X_3(10)$ with
$X_3(01) = \mathbb{I} \otimes G(1/2) \otimes |01\rangle\langle 01| + \mathbb{I} \otimes \mathbb{I} \otimes (\mathbb{I}^2 - |01\rangle\langle 01|)$
$X_3(10) = \mathbb{I} \otimes \mathbb{I} \otimes |10\rangle\langle 10| + \mathbb{I} \otimes \mathbb{I} \otimes (\mathbb{I}^2 - |10\rangle\langle 10|)$

k=4 $R_4 = X_4(001)X_4(101)X_4(010)$ with
$X_4(001) = \mathbb{I} \otimes |001\rangle\langle 001| + \mathbb{I} \otimes (\mathbb{I} - |001\rangle\langle 001|)$
$X_4(101) = \mathbb{I} \otimes |101\rangle\langle 101| + \mathbb{I} \otimes (\mathbb{I} - |101\rangle\langle 101|)$
$X_4(010) = \sigma_x \otimes |010\rangle\langle 010| + \mathbb{I} \otimes (\mathbb{I} - |010\rangle\langle 010|)$

As one can see many of the operators defined during this procedure, for example $X_1(1)$ and $X_3(10)$, are equal to identity. This suggests that the generated circuit can be simplified. This can be observed in Figure 5.1, where only nontrivial gates for this example are presented. One should also note that the simulation of the above procedure is limited by the number of qubits which can be used.

5.4 Sorting integers on quantum computer

This algorithm for state initialisation can be modified to implement radix sort algorithm [23, 64] on quantum computer. The crucial observation is that we need to generate a state such that the probability distribution obtained after the final measurement encodes the order of input elements.

In the state initialisation procedure we have obtained the flat superposition of input numbers. Thus we need to change the rotation matrices to obtain the appropriate rotation of the input state. But the most important difference is that we need to read-out the result to get information about the mutual order of input elements. This step is typical for quantum algorithms.[1]

The classical sorting problem can be stated as follows [64].

Task 5.3 (Sorting) Let X be the set of elements and K the set of keys with defined ordering relation $<$. For a given sequence of elements $A =$

[1] See for example the description of Shor's algorithm in [73]

$(a^1, a^2, \ldots, a^n) \subset X$ and the function $\kappa : X \to K$ find permutation π on $\{1, \ldots, n\}$ such that

$$\kappa(a^{\pi(1)}) < \kappa(a^{\pi(2)}) < \ldots < \kappa(a^{\pi(n)}). \tag{5.19}$$

There exist many algorithms solving this problem [64]. Below we focus on a single example only and develop its quantum version.

5.4.1 Radix sort algorithm

The idea of radix sorting appeared for the first time in the context of sorting punched cards, but it can be easily extended to any elements. The following definition can be found in [23, Chapter 9] and [64, Section 5.2.5].

Let us assume that A is an array of n elements and each element of A has d digits. Using pseudocode it can be written as presented in Listing 5.1.

Procedure: RadixSort(A, d)
Input: The array A of n d-digit elements.
Output: The array A with sorted elements.

```
while $i \leq d$ do
  SomeStableSort(A[i])
```

Listing 5.1: Pseudocode for the classical radix sort algorithm. Note that the time complexity of this algorithm depends on the complexity of the sorting algorithm used as a subprocedure.

Running time of radix sort procedure depends on the stable sort procedure used to sort each column of the input array. If each digit is in range $[1, 2 \ldots, k]$ where $k = O(n)$ we can use counting sort and in this case radix sort runs in linear time.

Theorem 5.1 (Complexity of radix sort [23]) Given n d-digit numbers in which each digit can take on up to k possible values, RadixSort(A, d) correctly sorts these numbers in $\Theta(d(n + k))$ time.

Unfortunately, counting sort algorithm does not sort in place and thus it is better to use fast comparison sort algorithm such as quick sort when we have to work with limited memory.

5.4.2 Quantum radix sort algorithm

In this section we describe how the methodology used for preparing flat superposition can be used to construct quantum version of the radix sort algorithm.

It was shown in [62] that for any storage bounds $n/\log n \geq S \geq \log^3 n$ there exist an algorithm that solves this problem in time $O(n^{3/2}\log^{3/2} n/\sqrt{S})$. In quantum case time-space trade-off for sorting n numbers from a polynomial size range in a general sorting algorithm is $TS = \Omega(n^{3/2})$, while in classical case this trade-off is $TS = \Theta(n^2)$.

It is also known that the lower bound for the number of comparisons in quantum oracle model is $\Omega(n \log n)$ [54].

The results cited above use quantum oracles to perform sorting and searching. More precisely, a quantum oracle is used to perform number comparison.

In contrast, the algorithm presented below describes the generation of quantum circuit from the information contained in the input numbers and it presents the quantum version of the radix sort algorithm [64]. The generated circuit does not require a quantum oracle since it is not based on comparison.

In analogy with the state initialisation problem, we will reformulate the classical sorting problem as follows

Task 5.4 (Quantum sorting) Let X be the set of elements and K the set of keys with a defined ordering relation $<$. For a given sequence of elements $A = (a^1, a^2, \dots, a^n) \subset \mathbb{Z}_{2^M}$, find quantum gate Q such that

$$\mathrm{P}(a^k) < \mathrm{P}(a^l) \Leftrightarrow a^k < a^l. \tag{5.20}$$

We are neglecting the key function $\kappa : X \to K$ from Task 5.3, since it can be implemented by appending an additional qubits for storing the value of the key for every input element.

Again, it is convenient to distinguish the operations acting on the first qubits, but in this case we process the most significant qubit in the first step. Note, that the gate Q does not solve Task 5.3 completely, since we need to perform measurements to extract information about the permutation π from the final state. Thus, the quantum radix sort algorithm produces a quantum state encoding the sorted list of numbers. It requires the sequence of measurements to extract information from the quantum state.

Part 1 – the most significant qubit

The generation of the quantum radix sort procedure starts from the most significant bit, which is encoded into the right-most qubit. Gate Q_1 reads

$$Q_1 = \begin{cases} \mathbb{I}^{M-1} \otimes \mathbb{I} & \text{if } \sum_i a_1^i = 0 \\ \mathbb{I}^{M-1} \otimes \sigma_x & \text{if } \sum_i a_1^i = K \\ \mathbb{I}^{M-1} \otimes G\left(\frac{2^{M+1}-1}{2^{M+1}}\right) & \text{otherwise} \end{cases} . \tag{5.21}$$

Note that, to check the conditions in this equation, only bit operations are required. As in the state initialisation example the generated circuit acts on the state $|0\dots0\rangle$.

Gate Q_1 acting on the state does not introduce the superposition of the base state unless input numbers differ at the first position.

Part 2 – qubits $M-1,\ldots,1$

We will use controls α_l to define gates to be performed on the other qubits. For each bit $n = M-1,\ldots,2$ we define an operation

$$Y_n(\alpha_l) = \begin{cases} \mathbb{I}\otimes\left(\mathbb{I}^{n-1} - |\alpha_l\rangle\langle\alpha_l|\right) + \mathbb{I}\otimes|\alpha_l\rangle\langle\alpha_l| & \text{if } \sum_i a_n^i = 0 \\ \mathbb{I}\otimes\left(\mathbb{I}^{n-1} - |\alpha_l\rangle\langle\alpha_l|\right) + \sigma_x\otimes|\alpha_l\rangle\langle\alpha_l| & \text{if } \sum_i a_n^i = c_n(\alpha_l) \\ \mathbb{I}\otimes\left(\mathbb{I}^{n-1} - |\alpha_l\rangle\langle\alpha_l|\right) + G\left(\frac{2^{n+1}-1}{2^{n+1}}\right)\otimes|\alpha_l\rangle\langle\alpha_l| & \text{otherwise} \end{cases}, \tag{5.22}$$

where the summation is taken over all the numbers a^i with the bits $M,\ldots,n-1$ equal to α_l (at the n-th step we operate using controls of the length $n-1$). Also in this case the above conditions can be verified using bit operations on a classical computer only.

Unitary matrix Q_n, which is generated at the n-th step of the procedure and represents the n-th step of the quantum algorithm, is defined as

$$Q_n = \mathbb{I}^{M-n}\otimes\left(\prod_{\alpha_l} Y_n(\alpha_l)\right), \tag{5.23}$$

and the gate Q which implements the sorting algorithm for the list A reads

$$Q = \prod_{n=1}^{M} Q_n. \tag{5.24}$$

The state obtained after the execution of the gate Q contains the sorted superposition of the input numbers. To show this let us assume that we have two input numbers that differ at the M-th position. The first step divides our state space into two subspaces, and the next steps of the generated algorithm operates on these subspaces independently and thus do not change the values of probabilities, but only perform σ_x or $\mathbb{I}$ gates.

To check that Q generates the desired probability distribution, let us assume that the measurement after the $(l-1)$-th step gives for $a_2 < a_1$ the probability distribution such that $\mathrm{P}_{l-1}(a_2) < \mathrm{P}_{l-1}(a_1)$. Then, if we have $a_3 < a_2$ such that $a_3^M\ldots a_3^{l-1} = a_2^M\ldots a_2^{l-1}$, $a_3^l = 0$ and $a_2^l = 1$, the probability distribution after the $l-th$ step reads

$$\mathrm{P}_l(a_2) = \frac{2^{l+1}-1}{2^{l+1}}\mathrm{P}_{l-1}(a_2) \text{ and } \mathrm{P}_l(a_3) = \frac{1}{2^{l+1}}\mathrm{P}_{l-1}(a_2) \tag{5.25}$$

and thus $\mathrm{P}_l(a_3) < \mathrm{P}_l(a_2) < \mathrm{P}_l(a_1)$. If on the other hand we have $a_3 > a_2$ and $a_3^M\ldots a_3^{l-1} = a_2^M\ldots a_2 l-1$ but $a_3^l = 0$ and $a_2^l = 1$, then

$$\mathrm{P}_l(a_2) = \frac{1}{2^{l+1}}\mathrm{P}_{l-1}(a_2) \text{ and } \mathrm{P}_l(a_3) = \frac{2^{l+1}-1}{2^{l+1}}\mathrm{P}_{l-1}(a_2) \tag{5.26}$$

and thus $\mathrm{P}_l(a_2) < \mathrm{P}_l(a_3) < \mathrm{P}_l(a_1)$.

Thus, the presented algorithm allows us to generate the superposition required to solve the problem stated in Task 5.4.

5.5 Time and space complexity

In this section we will study the complexity of the presented procedures.

Quantum gates R_k, $1 \leq k \leq M$ executed in the k-th step of the initialisation algorithm is composed of at most K rotations. Each rotation is controlled using at most $k-1$ qubits. Using Theorem 2.9 from Chapter 2 we can conclude that such operation can be decomposed into $O(k^2)$ elementary gates (single qubit operations and $CNOT$ gates).

As a results, gate R implementing quantum initialisation algorithm can be decomposed using $O(M^3)$ elementary gates, where M is the number of bits required to encode input elements. One should note that this bound it independent of the number of input elements, but it depends on their length.

The number of elementary gates required to implement gate Q, used in the sorting procedure, is of the same order as in the case of state initialisation. One needs to execute $O(M^3)$ elementary gates to prepare the state with sorted numbers of length M. However, in the case of quantum sorting the main problem arises from the readout of the algorithm results.

To obtain information from the final state one needs to perform sequence of measurements and the measurement procedure has the major contribution to the complexity of the quantum radix sorting.

The first advantage of the sorting algorithm is the ability to sort an exponential number of input elements. This is achieved by using quantum superposition. We have to use approximately $M \approx \log n$ qubits in this procedure. As a result, time complexity of this procedure is equal to $O(\log^3 n)$. However, it does not include the procedure for reading information about the order from the final quantum state which is approximately equal to $O(n^4)$ for n input numbers [79].

The second advantage of the presented algorithm over the existing quantum sorting algorithms is that it does not use a quantum oracle. This allows for better analysis of the algorithm and allows for optimisation in the generated quantum circuit.

One should note that it is very difficult to compare the proposed quantum sorting algorithm with classical results. Time bounds for quantum sorting are described in the terms of elementary quantum operations. In classical algorithms time complexity is expressed in the terms of elementary operations like addition.

5.6 Final remarks

The presented algorithm for register initialisation generates a quantum circuit for preparing flat superposition of the list of integers and thus shows, that such operation in high level programming languages can be implemented unitarily. This introduces the means for using integer numbers as quantum data types [117]. Similar concepts were developed using fuzzy numbers [29].

The generation of code for quantum machine by classical controlling device is one of the translation phases for quantum language compiler as described in [111]. The presented algorithm for state initialisation allows for optimisation of initialisation procedure and thus could be used for code generation in quantum programming language compiler in the architecture presented in Section 2.4.4.

The feature of the presented quantum radix sort algorithm is that it does not require a quantum oracle model and thus its complexity can be calculated without making assumptions about the complexity of an oracle. It also shows that the complexity of a quantum algorithm depends heavily on the read-out procedure complexity.

Chapter 6

Conclusions

The main goal of this thesis was to show how abstract operations on quantum data types can be used to develop new quantum algorithms. We have presented two original applications of high-level quantum programming languages for developing quantum algorithms and protocols.

The example of quantum Parrondo's paradox was motivated by the recent research efforts in the field of quantum game theory. We have shown how quantum programming language allows for the quantisation of a classical game. The presented scheme uses the features provided by a programming language to build a quantum algorithm from basic operations. Quantum programming languages allow us to divide quantum algorithms into smaller parts which are easier to control. We have shown how quantum conditional structures allow for more compact description of quantum algorithms.

The described quantum programming languages were developed using the syntax known from classical programming languages. For that reason, classical control structures can be easily incorporated into quantum programmes using the syntax of quantum programming languages. The ability to use classical control structures for controlling the execution of quantum operators is the biggest advantage of quantum programming languages. This is dificult to achieve in quantum circuits model and requires the introduction of non-unitary operations to this model.

On the other hand, existing quantum programming languages do not provide the sufficient level of abstraction. For example, it is difficult to define nonstandard quantum gates using their syntax. They also provide only basic data types for operating on quantum memory. We have shown that, by introducing new quantum data types, it is possible to achieve interesting results in the scope of quantum algorithms.

By defining operations on quantum registers, interpreted as integer numbers, we have constructed the sequence of unitary operations implementing the state initialisation. This algorithm was used to implement the elements of an interpreter for high-level quantum programming language kulka. Although this algorithm can only be applied to prepare a narrow set of states ie. base states in $\mathbb{C}^{2^n}$, it is easier to implement and optimise it than in the case of general purpose methods for preparing arbitrary superposition.

We have also extended the initialisation procedure and constructed a sorting algorithm. It implements the quantum version of radix sort algorithm. Its main advantage is good memory efficiency. This was achieved by using a typically quantum phenomenon — namely the superposition of states. Since the presented algorithm does not use a quantum oracle and it is not based on comparison, it cannot be reduced to existing algorithms, based on searching. We have also seen that it is difficult to use this algorithm in a real-world application due to the complexity caused by the measurement procedure. Nevertheless, this result suggests that provided methodology can be used to develop other quantum algorithms.

The work conducted while preparing this thesis could serve as a starting point for further development of a quantum programming language. As it has been shown, quantum types can provide an insight in the quantum algorithms construction. Thus, we intend to extend the methodology used to design kulka programming language for further research in the field of quantum algorithms.

Appendix A

Experimental quantum programming language kulka

In this chapter the syntax of the experimental quantum programming language kulka is presented. Experimental quantum programming language kulka was created to test the new method of developing quantum algorithms based on introducing quantum data types.

We provide the overview of data types defined in this language and describe briefly the functionality of the existing interpreter. Also a few examples of language usage are presented.

At the moment of writing only the basic interpreter for kulka exists. Also the syntax presented below is in preliminary form and probably will be changed. In many cases it can be enriched by adding features from modern classical programming languages. However, the main goal of this language was to test operations on quantum data types, not to provide general purpose programming language.

A.1 Motivation

The main motivation for creating new quantum programming language from scratch was to test new methods of operating on quantum data types. It should be stressed that, in contrast to existing programming languages, we do not describe quantum gates or measurement in this chapter. The presented language was designed to operate on quantum memory without using low-level quantum operators such as $CNOT$ or Hadamard gates.

A.2 Grammar

Below we describe the grammar of kulka programming language using extended Backus-Naur form including the rules for operators precedences.

Keywords and other literals are typesed in fixed-width font. Optional expressions are denoted using the question mark (?), alternatives are denoted as $\text{alt}_1|\text{alt}_2$. By (elem)* we denote an arbitrary number of occurences of elem and by (elem)+ one or more occurences of elem.

Basic components

A programme written in kulka programming language is composed of statements and it is possible to include external files with definitions.

Single line comments are introduced by # symbol. For example, the following line will be ignored

```
# File: fact.kulka
# Desc: Implmentation of factorial function
```

It is also possible to use C-style multi-line comments using the following syntax

```
/*
 File: qdata.kulka
 Author: Jarek Miszczak
 Version: 0.1
 */
```

Below STRING represents a sequence of characters enclosed in single or double quotes, INT represents an integer number with possible sign, REAL represents a floating-point number with possible sign.

Programme

program ::= (includeStmt)? stmt*
includeStmt ::= include string

Expressions

exprList ::= expr (, expr)*
expr ::= (idRef|arrayRef) = expr | orExpr

orExpr ::= andExpr ((or | ——) andExpr)*
andExpr ::= eqExpr ((and | &&) eqExpr)*

eqExpr ::= compExpr ((== | != | ¡¿) compExpr)*
compExpr ::= addExpr ((¿ | ¡ | ¡= | ¿=) addExpr)*

```
addExpr ::= mulExpr ((+ | -) mulExpr)∗
mulExpr ::= notExpr ((* | / | &) notExpr)∗

notExpr ::= (! | not)? negExpr∗
negExpr ::= (-)? primary∗
primary ::= atom | ( expr )
atom ::= varRef | subCall | arrayRef | constant

varRef ::= id
subCall ::= id ( exprList ) ’
arrayRef ::=

constant ::= string | numeric | ket | superposition
string ::= STRING
ket ::= — (0|1)+ ¿
superposition ::= ( INT (, INT )+ )
numeric ::= INT | REAL | complex
complex ::= [ expr , expr ]
```

Statements

```
stmt ::= expr ; | codeBlock | loop | ifStmt | return | break | ioStmt

codeBlock ::= { stmt ∗ }

loop ::= forLoop | whileLoop
foorLoop ::= for ( varDef ; expr ; expr ) codeBlock
whileLoop ::= while ( expr ) codeBlock

ifStmt ::= if ( expr ) codeBlock else codeBlock
break ::= break ;
return ::= return expr ;

ioStmt ::= printStmt | readStmt
printStmt ::= say expr ; | say_nl expr ;
readStmt ::= ask id ;
```

Data types, subroutines and definitions

```
scalarType ::= numericType | quantumType | stringType
numericType ::= int | real | complex
```

quantumType ::= qreg | qint
stringType ::= string

varDef ::= scalarType id (= expr)∗
arrayDef ::= scalarType id (= expr)∗

subDef ::= scalarType id (paramsList) codeBlock
paramsList ::= (param (, param)∗)?
param ::= scalarType id

A.3 Data types and subprocedures

There are three data types defined for numeric values – int, real and complex. For convenience, type string is provided for dealing with string literals. For operating on quantum memory kulka introduces two types – qreg and qint.

Classical types

Numeric types in kulka programming language are implemented using Java numeric types.

Int

Data type int is implemented using java.lang.Integer standard Java class. The variable of this type can be used to store integer numbers in range $[-2^{31}, 2^{31} - 1]$.

Real

The variable of type real can store double precision floating-point numbers. It is based on java.lang.Double Java standard class and can be used to store values of 64-bit double precision numbers.

Complex

Complex numbers can be initialised using real and imaginary part

```
complex c1 = [0,1];
```

or using real or complex variables

```
real a = 1;
complex c2 = [a,1.5];
complex c3 = [1,1] + c2;
```

See also Figure A.3 for more examples.

At the moment kulka programming language does not provide syntax complex cv = 1 + 1i; provided by other programming languages (eg. Python[1] or GNU Octave[2]).

Strings

String processing is provided to facilitate the interaction with a user. Basic escape sequences – including “n (new line) and “t (tab) – can be used inside string literals. For example

—kulka¿ string s = ”This“tdog“tis“tblack!“n”;
—kulka¿ say s;

gives

This‐‐‐‐dog‐‐‐‐‐is‐‐‐‐‐‐black!
—kulka¿

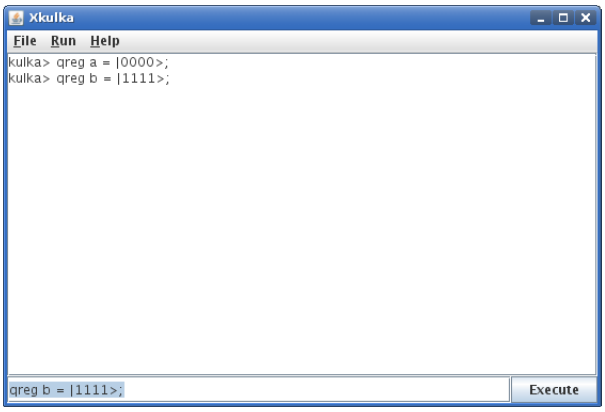
Here the symbol ‐ is used to indicate a blank symbol (space).

Quantum types

The variable of type qreg can be used to store state of quantum registers. Initialisation of quantum register can be performed usign Dirac notation. For example

```
qreg r1 = |0100>;
```

prepares register r1 in the state $|0100\rangle$.



```
kulka> qreg a = |0000>;
kulka> qreg b = |1111>;
```

```
qreg b = |1111>;
```

Figure A.1: An example of quantum register allocation using Dirac notation. Quantum registers can be initialised to any base state

[1] Python v3.0 documentation, http://docs.python.org/dev/3.0/
[2] GNU Octave documentation, http://www.gnu.org/software/octave/doc/interpreter/

Name	Description	Arguments
sin(x), cos(x), tan(x), cot(x)	trigonometric functions	numeric
sqrt(x)	square root of x	numeric
pow(x,y)	x^y	numeric, numeric
random()	random value from $[0,1)$	—
exp(x)	exponential function e^x	numeric
log(x)	natural logarithm	numeric
pi(), e(), h(), k()	mathematical and physical constants	—

Table A.1: Mathematical functions available in kulka programming language interpreter.

For a convenient manipaulation on superpositions of integer numbers (quantum integers) kulka provides qint data type. The variables of this type can be used to store a superposition of integer numbers and to operate on them. Initialisation is performed using the following syntax

```
qint q1 = (1|5|10);
```

which prepares the state

$$|q_1\rangle = \frac{1}{\sqrt{3}}(|1\rangle + |5\rangle + |10\rangle) = \frac{1}{\sqrt{3}}(|0001\rangle + |0101\rangle + |1010\rangle)$$

The initialisation of quantum integers is performed using the algorithm described in Chapter 5.

Classical functions

Subroutines can be defined in kulka using syntx similar to C programmiong language.

Standard functions

kulka programming language provides some standard mathematical functions known from any computer algebra system. Nevertheless, this list is far from being exhaustive, since we do not aim to provide a replacement for general programming language for calculations.

Quantum gates

Since the main goal of the kulka programming language was to test some new ideas, it provides a limited support for standard notion of quantum gates known from QCL, LanQ or cQPL.

At the moment kulka supports only basic quantum gates like *CNOT*, *NOT* and the Hadamard gate (see Table 2.2 in Chapter 2). They are implemented as a part of the standard library. For example, the following line in the stdlib.kulka file

```
gate Sx = { { 0, 1 } , { 1, 0 } };
```

declares Pauli gate σ_x or *NOT* gate. At the moment it is impossible to declare new quantum gates using the interpreter.

To perform Sx() operation on the register q one can use the following syntax:

```
qreg q = |0100>;
Sx(q);
```

Data conversions

Since kulka programming language allows to use classical and quantum data types, we need to specify the casting rules for conversion between them.

The mesurement of a quantum variable is performed using casting on classical variable. The value of the measurement is stored in a classical variable.

```
qint a = (2|4|6);
int b = a;
say b;
```

The output of the last statement is random, since the assignement int b = a; is performed by measuring the state of the variable a, and assigning it to a classical variable b.

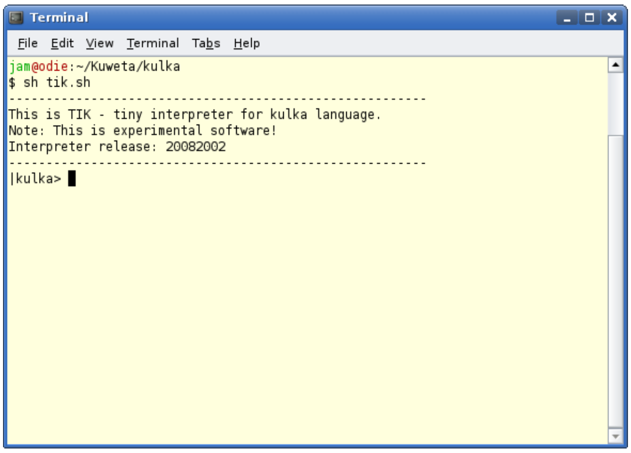
It is also possible to perform the measurement using say or say_n operators. In this case the measured value is printed to the standard output.

A.4 Interpreter implementation

The existing interpreter is implemented in Java programming language. It uses ANTLR [95] parser generator to implement lexical analysis.

Simulation layer was developed using JScience class library [24], which provides basic matrix manipulation routines. The algorithms described in Chapter 5 are implemented without any optimisation.

It should be stressed that no effort has been made toward speed optimisation. Thus, it is possible to run programmes operating only on a small number of qubits.



```
jam@odie:~/Kuweta/kulka
$ sh tik.sh
--------------------------------------------------------
This is TIK - tiny interpreter for kulka language.
Note: This is experimental software!
Interpreter release: 20082002
--------------------------------------------------------
|kulka>
```

Figure A.2: Text user interface allows for interactive work with kulka programming language interpreter.

User interface

Batch execution

Programmes written in kulka can be executed in batch mode using kulka.Exec class. In the following example, $ indicates the command line input.

```
$ cat ex/base.kulka
# example of integer and string manipulation
int x = 2 + 2;
int y = 5 + x;
say ”x = ”;
say˙nl x;
say ”x + y = ”;
say˙nl x + y;
$ java kulka.Exec ex/base.kulka
x = 4
x + y = 13
```

Text mode shell

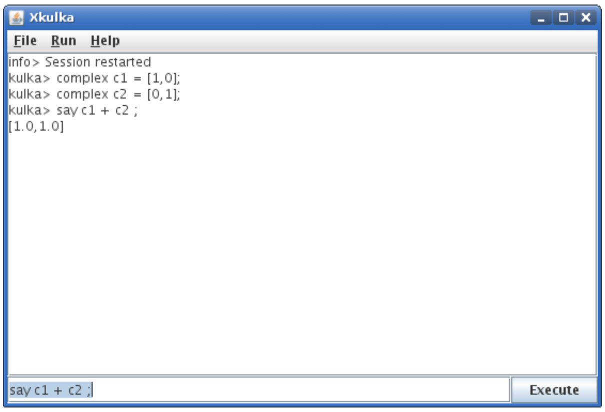
For more convinient testing of the language text mode shell is provided. It allows for interactive work with the interpreter. Also command line history is provided using JLine class library.[3]

The example of an interactive session is presented in Figure A.2.

[3] JLine – Java library for handling console input, http://jline.sourceforge.net/.

Graphical shell

Scripts can be also executed using graphical user interface. It allows for restarting the interpreter session. It can be used for batch execution of kulka programming language scripts as well as for interactive work.



```
info> Session restarted
kulka> complex c1 = [1,0];
kulka> complex c2 = [0,1];
kulka> say c1 + c2 ;
[1.0,1.0]
```

```
say c1 + c2 ;
```

Figure A.3: Graphical shell for kulka interpreter with an example of complex numbers usage

Errors

Since kulka is an experimental language, only basic functionality for handling errors is provided.

Table A.2 contains the summary of errors reported by the kulka programming language interpreter. Every class implementing the particular type of error must be a subclass of kulka.lang.errors.KulkaError, which is an abstract class for gathering base methods.

Extending interpreter

It is possible to add more built-in functions by adding new definitions to stdlib.kulka in bin/kulka/stdlib directory. For example function random() is defined by adding to this file the following line

```
real random() kulka.stdlib.methods.Random;
```

which tells the interpreter that the function random() returns the value of type real, it does not require any arguments and it is implemented in kulka.stdlib.methods.Random class. This method is used to implement all standard functions listed in Table A.1.

Name	Description
FunctionNotImplemented	Call to undefined function
InitializationError	Error caused by problems with variable initialisation
TypeMismatch	Undefined conversion between data types
UndefinedValue	Undefined value
UndefinedVariable	Reference to undefined variable
UnsupportedOperation	Operation which was not defined for particular types
VariableRedefinition	Redefinition of a variable or a function

Table A.2: Errors reported by the kulka interpreter. Error handling mechanism is implemented using Java exceptions.

A.5 Remarks and further information

The presented programming language has the syntax similar to many modern programming languages. Most of its elements should be familiar to Java and C programmers. The syntax for initialising quantum integers is based on Perl 6 syntax.[4] In particular the parts of grammar were based on Lua 5.1[5] grammar for ANTLR[6] by Nicolai Mainiero and C-- compiler and interpreter[7] by Scott Fortmann-Roe. The implementation of the standard library was based on LanQ interpreter[8] (see also [83]).

Further information related to the interpreter for the kulka quantum programming language can be obtained from [126].

[4] Perl Development: Perl 6, http://dev.perl.org/perl6/

[5] Lua 5.1 Reference Manual, http://www.lua.org/manual/5.1/

[6] List of available ANTLR grammars: http://www.antlr.org/grammar/list

[7] Source code available at: http://www.antlr.org/share/list

[8] LanQ project page: http://lanq.sourceforge.net/

Appendix B

Mathematics of quantum information

This chapter provides basic mathematical tools used in quantum mechanics and quantum information theory. We review the formalism of finite-dimensional Hilbert spaces, Dirac notation used in quantum mechanics and quantum information theory, and the formalism of density matrices. We also provide some examples of quantum algorithms and protocols. Also, a brief introduction to quantum operations is presented and some examples of quantum operations are given.

For the complete introduction to the subject please refer to [51, 89, 118] and [50]. The review of linear operators theory in finite dimensional vector spaces can be found in [14] and [53]. Recent developement in the theory of entangled states is presented in [8].

B.1 Structure of quantum theory

In this section we provide basic facts concerning mathematical structure of quantum theory.

Preliminaries

1. (a)

Let $\mathcal{H}$ be a separable, complex Hilbert space used to described the system in question. We use Dirac notation [30] for inner product in $\mathcal{H}$

$$\langle\psi|\phi\rangle, \tag{B.1}$$

where $|\psi\rangle, |\phi\rangle \in \mathcal{H}$ and $\langle\phi| = |\phi\rangle^\star$ is a complex conjugate of $|\phi\rangle$. Symbol $|\psi\rangle\langle\phi|$ denotes the operator of rank one (ie. a projection operator) which acts on a vector $|\alpha\rangle \in \mathcal{H}$ as

$$(|\psi\rangle\langle\phi|)\,|\alpha\rangle = \langle\phi|\alpha\rangle|\psi\rangle. \tag{B.2}$$

In quantum information theory we deal mainly with finite-dimensional Hilbert spaces.

Observables

The term observable is used to describe a physical quantity of a system which can be observed and measured. Observables are quantum mechanical analogous of the random variables from classical mechanics.

In quantum mechanics observables are described by self-adjoint operators, ie. linear functions $X : \mathcal{H} \to \mathcal{H}$ such that

$$\langle X\psi|\phi\rangle = \langle\psi|X^{\star}\phi\rangle, \tag{B.3}$$

for any $|\phi\rangle, |\psi\rangle \in \mathcal{H}$. An important property of self-adjoint operators is expressed by the following theorem.

Theorem B.1 (Spectral decomposition) Every self-adjoint operator A can be decomposed according to the formula

$$A = \sum_{\lambda_i \in \sigma(A)} \lambda_i |x_i\rangle\langle x_i| \tag{B.4}$$

with $\sum_i |x_i\rangle\langle x_i| = \mathbb{I}$, where $|x_i\rangle$ are the eigenvectors of A with corresponding eigenvalues λ_i.

Here $\sigma(A)$ denotes the spectrum of the operator A. The mapping $\mu : \lambda_i \to |x_i\rangle\langle x_i|$ is called spectral measure [81].

Spectral decomposition can be used to define functions on the space of self-adjoint operators. If $f : \mathbb{R} \to \mathbb{C}$ is any function, one can define $f(A)$ as

$$f(A) = \sum_{\lambda_i \in \sigma(A)} f(\lambda_i)|x_i\rangle\langle x_i|. \tag{B.5}$$

For example, if we take $A = NOT$, we have

$$NOT = +1 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} - 1 \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}, \tag{B.6}$$

and square root of NOT can be easily calculated as

$$\sqrt{NOT} = \sqrt{+1} \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} + \sqrt{-1} \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{B.7}$$

States

In quantum mechanics the state of the system is described by the density matrix, ie. Hermitian operator $\rho : \mathcal{H} \rightarrow \mathcal{H}$, which is

$$\rho \geq 0 \quad \text{(positive)} \tag{B.8}$$

and

$$\operatorname{tr}[\rho] = 1 \quad \text{(normalized).} \tag{B.9}$$

Density matrix in the analogue of the classical probability distribution.

The set of states $\mathcal{S}(\mathcal{H})$ is a convex set and thus every $\rho \in \mathcal{S}(\mathcal{H})$ can be represented as a convex combination

$$\rho = \sum_i p_i \sigma_i, \tag{B.10}$$

with $\sigma_i \in \mathcal{S}(\mathcal{H}),\ \sum_i p_i = 1$.

If the operator ρ additionally fulfils the condition

$$\rho^2 = \rho \tag{B.11}$$

ie. ρ is a projection operator, the state described by ρ is said to be pure. In this case density operator posesses only one eigenvector $|\alpha\rangle$, which can be used to describe the state. Therefore, in the case of pure states, we can write

$$\rho = |\alpha\rangle\langle\alpha|. \tag{B.12}$$

In particular, if $\mathcal{H} = \mathbb{C}^2$ we say that $|\psi\rangle \in \mathcal{H}$ describes the state of a qubit. In this case

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{B.13}$$

where $\alpha^2 + \beta^2 = 1$ and $|0\rangle, |1\rangle \in \mathbb{C}^2$ form a base $\mathbb{C}^2$.

Results of experiments

Quantum mechanics is a probabilistic theory in a sense, that it can only predict the probability of events. We have already noted, that states and observables can be understood as analogues of probability distributions and random variables from classical mechanics. Physical experiments can be interpreted using the following theorem.

Theorem B.2 (Born-von Neuman formula [51]) The results of the measurement of observable A in the state ρ are described by the probability distribution $\mathrm{P}(\alpha) = \operatorname{tr}[\rho\mu(\alpha)]$ on the spectrum of the operator A.

Unitary evolution

Let us assume that the described system is isolated during the time of evolution. If initially the system is in a pure state $|\psi\rangle$, then the evolution of the system is given by some unitary operator U. The action of the operator U on the initial state is given by a formula

$$|\psi\rangle \to U|\psi\rangle. \tag{B.14}$$

If the initial state of the system is mixed and given by the density matrix ρ, then the final state is given by

$$\rho \to U\rho U^{\dagger}. \tag{B.15}$$

However, in many situations it is impossible to avoid the interaction of the systems with an environment.

B.1.1 Operations

The most general framework for describing evolution in quantum systems is based on the completely positive operators acting on the space of states.[1]

Definition B.1 Operator $E : \mathcal{S}(H) \to \mathcal{S}(H)$ (superoperator) is called completely positive if E is positive and

$$E \otimes \mathbb{I}^{\otimes n} \tag{B.16}$$

is positive for any $n \in \mathbb{N}$.

Definition B.2 (Quantum operation) Operator $E : \mathcal{S}(H) \to \mathcal{S}(H)$ is called an operation if it is completely positive.

Definition B.3 (CP-map) A map $T : B \to B$ is called completely positive (CP) if it is positive and map $T \otimes \mathbb{I}^n$ is positive for arbitrary n.

Important representation of the CP-maps is given by the Kraus form.

Theorem B.3 (Kraus representation) Every completely positive and trace-preserving map $T : S(\mathcal{H}) \to S(\mathcal{H})$ can be expressed as

$$T(\varrho) = \sum_{k=1}^{K} t_k \varrho t_k^{\dagger},$$

for all $\varrho \in S(\mathcal{H})$ and $K \leq \dim^2 \mathcal{H}$. Kraus operators $t_k : \mathcal{H} \to \mathcal{H}$ satisfy the completeness relation

$$\sum_k t_k^{\dagger} t_k = \mathbb{I}.$$

In many cases quantum operations are also called quantum channels [60].

[1] An operator acting on the space of density operators is called a superoperator.

B.1.2 Composite systems

To perform any useful calculation more the one qubit must be used. In quantum mechanics a system composed of two subsystems A and B is described in Hilbert space $\mathcal{H}_{AB}$ which is built as a tensor product of Hilbert spaces associated with subsystems $\mathcal{H}_{AB} = \mathcal{H}_A \otimes \mathcal{H}_B$.

This representation of states allows for a phenomenon known as quantum entanglement. Any pure state $|\alpha\rangle_{AB}$ of the bipartite quantum system can be represented as [8]

$$|\alpha\rangle_{AB} = \sum_{i=1}^{k} \sqrt{\lambda_i} |a_i\rangle_A \otimes |b_i\rangle_B, \tag{B.17}$$

where vectors $|a_i\rangle_A$ and $|b_i\rangle_B$ are orthonormal for systems A and B respectively and $\sum_i \lambda_i = 1$. Decomposition in equation (B.17) is known as Schmidt decomposition.

Definition B.4 Pure state $|\alpha\rangle_{AB}$ is called separable (ie. non-entangled) if it can be expressed as

$$|\alpha\rangle_{AB} = |a_i\rangle_A \otimes |b_i\rangle_B \tag{B.18}$$

for some $|a_i\rangle_A \in \mathcal{H}_A$ and $|b_i\rangle_B \in \mathcal{H}_B$. In the other case it is called entangled.

The definition of separability in the case of a mixed state of a bipartite quantum system is more complicated. The following definition was given in [120].

Definition B.5 State $\rho \in \mathcal{S}(\mathcal{A} \otimes \mathcal{B})$ of the bipartite quantum system is called separable if it can be written in the form

$$\rho = \sum_{i=1}^{n} p_i \rho_i^A \otimes \rho_i^B \tag{B.19}$$

where $\rho_i^A \in \mathcal{S}(\mathcal{A})$, $\rho_i^A \in \mathcal{S}(\mathcal{A})$ and $\sum_{i=1}^{n} p_i = 1$.

The theory of entangled states is an important part of quantum information theory. More information concerning this topic can be found in [8].

B.2 Examples

Quantum information theory deals with the manipulation of quantum systems and aims to develop procedures that would enable us to harness quantum mechanical phenomena. The following examples present some important features of the quantum computational model. See also [74] for more examples.

B.2.1 Quantum registers

The simplest quantum system is represented by the two-dimensional Hilbert space $\mathbb{C}^2$. In this case it is convenient to introduce the logical representation for the base states.

The following standard notion used in [89, 50] defines the computational basis as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \; |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{B.20}$$

where vectors $|0\rangle$ and $|1\rangle$ are the eigenvectors of the σ_z operator

$$\sigma_z|0\rangle = +1|0\rangle, \; \sigma_z|1\rangle = -1|1\rangle. \tag{B.21}$$

Quantum registers are defined as arrays of qubits. The quantum register of dimension n is represented as a vector in $\mathbb{C}^{2^n}$ Hilbert space. In other words, the quantum state of a quantum register is represented by the vector in $\mathbb{C}^n$

$$|\alpha\rangle = |\alpha_1\alpha_2\ldots\alpha_n\rangle \in \mathbb{C}^n, \tag{B.22}$$

with convention that

$$|00\ldots0\rangle \equiv |0\rangle \otimes |0\rangle \otimes \ldots \otimes |0\rangle, \tag{B.23}$$

$$|11\ldots1\rangle \equiv |1\rangle \otimes |1\rangle \otimes \ldots \otimes |1\rangle. \tag{B.24}$$

Single-qubit quantum gate U acts on quantum register $|\alpha_n\rangle \in \mathbb{C}^n$ as

$$U^{\otimes n}|\alpha\rangle \equiv U|\alpha_1\rangle \otimes \ldots \otimes U|\alpha_n\rangle \tag{B.25}$$

See also Definition 2.16 in Chapter 2.

B.2.2 Deutsch's algorithm

In [25] Deutsch provided a simple example of quantum algorithm. The proposed algorithm is capable of determining in a single step whether the given function $f : \{0,1\} \to \{0,1\}$ is either constant ($f(1) = f(0)$) or balanced. Deutsch's algorithm uses a function implemented as an oracle (see also Definition 2.11 in Chapter 2). In other words, a function is given in the form of unitary operator

$$G|x\rangle|y\rangle = |x\rangle|f(y) \oplus y\rangle, \tag{B.26}$$

where $a \oplus b$ is addition modulo 2.

Below we present the version of Deutsch's algorithm from [93]. Different realisation can be found in [85].

Deutsch's algorithm is composed of the following steps:

1. Prepare initial state $|\psi_0\rangle = |0\rangle|0\rangle$.

2. Perform Hadamard operation on the first qubit

$$|\psi_1\rangle = H \otimes \mathbb{I}|0\rangle|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)|0\rangle. \tag{B.27}$$

3. Apply the oracle operation

$$|\psi_2\rangle = G|\psi_1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle|f(0)\rangle + |1\rangle|f(1)\rangle\right). \tag{B.28}$$

4. Perform Hadamard operation on both qubits $|\psi_3\rangle = H \otimes H|\psi_2\rangle$

$$\begin{aligned}|\psi_3\rangle &= \sum_{x,y\in\{0,1\}} \left((-1)^{yf(0)} + (-1)^{x+yf(1)}\right)|x\rangle|y\rangle & \text{(B.29)}\\ &= \frac{1}{\sqrt{2}}\left(|0\rangle|0\rangle + (-1)^{f(0)\oplus f(1)}|f(0)\oplus f(1)\rangle|1\rangle\right). & \text{(B.30)}\end{aligned}$$

5. Measure the state of both registers.

The first qubit will contain the value $f(0) \oplus f(1)$ whenever 1 has been measured in the second register. Thus, using the above procedure, it is possible to find out the type of function f with the probability $\frac{1}{2}$. Using the modified version of this procedure described in [85] it is possible to solve this problem with the probability 1.

In Chapter 3 we have described a quantum circuit realising this algorithm along with its QCL implementation.

B.2.3 Quantum teleportation

Quantum teleportation is an example of how non-classical features of quantum mechanics allow for creating new protocols, in this case for transferring an unknown quantum state between two parties.

According to the rules of quantum mechanics it is impossible to make a copy of an unknown quantum state [122, 6]. It is possible, however, to send an unknown quantum state $|x\rangle$, if we have a quantum channel to our disposal.

Let us assume that Alice and Bob share bipartite entangled state $|\phi_+\rangle$. To send a perfect copy of an unknown state $|x\rangle = \alpha|0\rangle + \beta|1\rangle$ Alice has to perform the following steps:

1. Prepare state $|x\rangle|\phi^+\rangle$.

2. Measure the state of the first two qubits in $\{|\phi_\pm\rangle, |\psi_\pm\rangle\}$,

3. Using classical channel send two bits of information describing her measurement to Bob.

4. At the last step Bob has to reconstruct state $|x\rangle$. To achieve this he performs the following steps on his part of the system, depending on the information obtained from Alice
   (a) $|\phi_+\rangle \rightarrow \mathbb{I}$,
   (b) $|\psi_+\rangle \rightarrow \sigma_x$,
   (c) $|\psi_-\rangle \rightarrow \sigma_y$,
   (d) $|\phi_-\rangle \rightarrow \sigma_z$.

Initial state of three qubits can be described using Bell states

$$\begin{array}{rcl} |x\rangle|\phi^+\rangle & = & \frac{1}{\sqrt{2}}\left(\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle\right) \\ & = & \frac{1}{2}[|\phi_+\rangle(\alpha|0\rangle + \beta|1\rangle) + |\phi_-\rangle(\alpha|0\rangle - \beta|1\rangle) \\ & + & |\psi_+\rangle(\alpha|1\rangle + \beta|0\rangle) + |\psi_-\rangle(\alpha|1\rangle - \beta|0\rangle)] \end{array}$$

If, for example, Alice gets the state $|\phi_-\rangle$ in result of her measurement, Bob gets the state

$$\alpha|0\rangle - \beta|1\rangle, \tag{B.31}$$

and thus he has to perform operation σ_z to reconstruct the input state

$$\sigma_z(\alpha|0\rangle - \beta|1\rangle) = \alpha|0\rangle + \beta|1\rangle. \tag{B.32}$$

This completes the protocol.

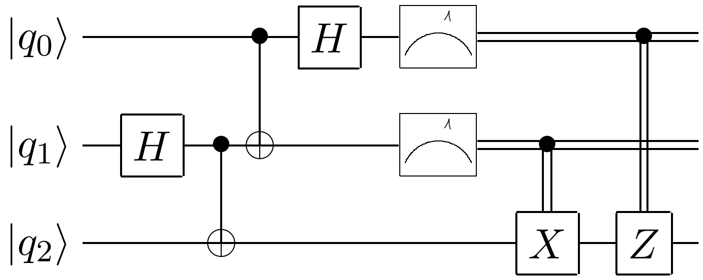


Figure B.1: Circuit for quantum teleportation. Double lines represent operation which is executed depending on the classical data obtained after the measurement on a subsystem.

The quantum circuit realizing teleportation protocol is presented in Figure B.1. One of the interesting features of this protocol is its usage of classical data obtained as the result of measurement.

In Chapter 3 we have presented the implementation of quantum teleportation in LanQ and cQPL programming languages..

B.2.4 Quantum channels and quantum errors

The introduction of general notion of completely positive maps (see Section B.1.1) is required to describe the general evolution of a quantum system. In particular, it allows us to describe the nonunitary evolution of the systems interacting with an environment.

In real-world situations it is impossible to isolate a quantum system completely. Its state becomes mixed due to the interactions with environment. This phenomenon is know as decoherence. The formalism of quantum operations described by completely positive maps allows us to describe errors in physical implementations of quantum computers.

Simple examples

Since the formalism of quantum operations is very general, it allows to express the notion of unitary evolution and measurement easily.

The simplest example of quantum operation is given by a unitary evolution described by the matrix U. In this case Kraus representation of the channel consists of one operator U only. The action of the channel on the initial state ρ is given by a standard formula B.15

$$\rho \to U\rho U^\dagger. \tag{B.33}$$

Another example of quantum channel is provided by the orthogonal or von Neumann measurement [118]. Let us assume that one measures the observable σ_z. In the case of one qubit this type of measurement is expressed using Kraus operators as

$$\left\{ |0\rangle\langle 0| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, |1\rangle\langle 1| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\}. \tag{B.34}$$

It transforms the initial quantum state

$$\rho = \begin{pmatrix} a & b^\star \\ b & 1-a \end{pmatrix} \tag{B.35}$$

as

$$\rho \to |0\rangle\langle 0|\rho|0\rangle\langle 0| + |1\rangle\langle 1|\rho|1\rangle\langle 1| = \begin{pmatrix} a & 0 \\ 0 & 1-a \end{pmatrix}. \tag{B.36}$$

Another example of measurement operators was given in Chapter 4, where it was used to discuss the Prisoner's dilemma.

Quantum errors

To describe quantum communication channels it is necessary to take the errors occuring during the transition of quantum states into account. Below Kraus operators for some standard one-qubit quantum channels are presented [89].

- ❏ Depolarizing channel:

$$\left\{\sqrt{1-\frac{3\alpha}{4}}\mathbb{I}, \sqrt{\frac{\alpha}{4}}\sigma_x, \sqrt{\frac{\alpha}{4}}\sigma_y, \sqrt{\frac{\alpha}{4}}\sigma_z\right\}.$$

- ❏ Amplitude damping:

$$\left\{\left(\begin{array}{cc} 1 & 0 \\ 0 & \sqrt{1-\alpha} \end{array}\right), \left(\begin{array}{cc} 0 & \sqrt{\alpha} \\ 0 & 0 \end{array}\right)\right\}.$$

- ❏ Phase damping:

$$\left\{\left(\begin{array}{cc} 1 & 0 \\ 0 & \sqrt{1-\alpha} \end{array}\right), \left(\begin{array}{cc} 0 & 0 \\ 0 & \sqrt{\alpha} \end{array}\right)\right\}.$$

- ❏ Phase flip:

$$\left\{\sqrt{1-\alpha}\mathbb{I}, \sqrt{\alpha}\sigma_z\right\}.$$

- ❏ Bit flip:

$$\left\{\sqrt{1-\alpha}\mathbb{I}, \sqrt{\alpha}\sigma_x\right\}.$$

- ❏ Bit-phase flip:

$$\left\{\sqrt{1-\alpha}\mathbb{I}, \sqrt{\alpha}\sigma_y\right\}.$$

Parameter α is used here to change the amount of noise in a quantum channel. For example qubit $|\psi\rangle$ transmitted through the bit-flip channel will be transmitted without an error with probability $\sqrt{1-\alpha}$ and with probability $\sqrt{\alpha}$ it will change its state to $\sigma_x|\psi\rangle$.

Appendix C

Notation

Below we summarize the notation used in this thesis. We follow the standard notation used in [89] and [50].

$\mathcal{H}_A$	complex Hilbert space representing system A
$\mathcal{B}(\mathcal{H})$	set of bounded operators acting on the complex Hilbert space $\mathcal{H}$
$\mathcal{S}(\mathcal{H})$	space of quantum states (density matrices) on the Hilbert space $\mathcal{H}$
$\dim(\mathcal{H}_A)$	dimension of the Hilbert space $\mathcal{H}_A$
$\mathcal{H}_A \otimes \mathcal{H}_B$	tensor product of Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$
$\mathcal{H}^{\otimes n}$	n-fold tensor product of the Hilbert space $\mathcal{H}$
$\|n\rangle \in \mathcal{H}_N$	normalized base vector in N-dimensional Hilbert space $\mathcal{H}_N$,
$\|\psi_1\rangle \otimes \|\psi_2\rangle$	tensor product of pure states $\|\psi_1\rangle$ and $\|\psi_2\rangle$
$\langle k\|l\rangle$	scalar product of the vectors $\|k\rangle$ and $\|l\rangle$
$\|\psi\rangle\langle\phi\|$	projection operator
$\mathrm{tr}\,[X]$	trace of the opertor $X \in \mathcal{B}(\mathcal{H})$
$\mathbb{C}^2$	2-dimensional Hilbert space representing one-qubit
$\|0\rangle^{\otimes n}$	base state of n qubits
$\rho \in \mathcal{S}(\mathcal{H})$	mixed state (density matrix) in the space $\mathcal{S}(\mathcal{H})$
$\rho_1 \otimes \rho_2$	tensor product of mixed states ρ_1 and ρ_2
$\frac{1}{N}\mathbb{I}$	maximally mixed state over N-dimensional Hilbert space

$\vert\psi\rangle\langle\psi\vert$	density matrix for a pure state $\vert\psi\rangle$
U_A	unitary matrix acting on Hilbert space $\mathcal{H}_A$
$U_1 \otimes U_2$	tensor product of unitary matrices U_1 and U_2
$U^{\otimes n}$	unitary matrix acting on Hilbert space $\mathcal{H}^{\otimes n}$
$\sigma_x, \sigma_y, \sigma_z$	Pauli matices
$\mathbb{I}$	identity operator
NOT	negation ($\sim$) quantum gate $NOT \equiv \sigma_x$
H	Hadamard quantum gate
$\sqrt{NOT}$	square root of NOT gate
$CNOT$	controlled negation gate
$\wedge$	logical and
$\vee$	logical or
$\sim$	logical not
$(Q, A, \delta, q_0, q_a, q_r)$	Turing machine over the alphabet A with transition function δ
$(q_i, x'a, b_1y')$	configuration of a Turing machine
$O(f(x))$	set of functions bounded from above by $f(x)$
$\Omega(f(x))$	set of function bounded from below by $f(x)$
$\Theta(f(x))$	set of function bounded by $f(x)$
P	complexity class **P** (polynomial)
NP	complexity class **NP** (nondeterministic polynomial)
RP	complexity class **RP** (randomized polynomial)
coRP	complexity class **coRP** – complement of **RP**
ZPP	complexity class **ZPP** (polynomial randomized with zero probability of error)
BPP	complexity class **BPP** (bounded probability of error)
BQP	complexity class **BQP** (bounded probability of error on quantum Turing machine)

List of Figures

List of Listings

List of Tables

Bibliography

[1] S. Aaronson and G. Kuperberg. Complexity ZOO. On-line aencyclopedia avaible at http://qwiki.stanford.edu/wiki/Complexity˙Zoo.

[2] H. Abelson, G. J. Sussman, and J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1996.

[3] A. Ambainis. Quantum walks and their algorithmic applications. International Journal of Quantum Information, 1:507–518, 2003. quant-ph/0403120.

[4] A. Ambainis. Quantum walk algorithm for element distinctness. SIAM Journal on Computing, 37:210–239, 2007.

[5] D. G. Angelakis, M. Christandl, A. Ekert, A. Kay, and S. Kulik, editors. Quantum information processing, volume 199 of NATO Science Series. Series III: Computer and System Sciences. IOS Press, 2006.

[6] L. E. Ballenetiene. Quantum Mechanics. A Modern Develepment. World Scientific, 1998.

[7] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. Phys. Rev. A, 52:3457, 1995.

[8] I. Bengtsson and K. yczkowski. Geometry of quantum states. Cambridge University Press, 2006.

[9] C. H. Bennett and G. Brassard. Quantum cryptography: public key distribution and coin tossing. In Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore, India, pages 175–179, 1984.

[10] E. Bernstein and U. Vazirani. Quantum complexity theory. SIAM Journal on Computing, 26(5):1411–1473, 1997.

[11] S. Bettelli. Toward an architecture for quantum programming. PhD thesis, Università di Trento, February 2002.

[12] S. Bettelli, L. Serafini, and T. Calarco. Toward an architecture for quantum programming. Eur. Phys. J. D, 25(2):181–200, 2003.

[13] S. Bettelli, L. Serafini, and T. Calarco. Toward an architecture for quantum programming. Eur. Phys. J. D, 25(2):181–200, 2003.

[14] R. Bhatia. Matrix Analysis, volume 169 of Garduate Texts in Mathematics. Springer-Verlag, 1997.

[15] C. Bohm. On a family of turing machines and the related programming language. ICC Bull., 3:187–194, 1964.

[16] J. Bouda. Encryption of Quantum Information and Quantum Cryptographic Protocols. PhD thesis, Masaryk University, 2004.

[17] D. Bouwmeester, A. Ekert, and Zeilinger A., editors. The Physics of Quantum Information: Quantum Cryptography, Quantum Teleportation, Quantum Computation. Springer, 2000.

[18] G. Brassard, A. Broadbent, and A Tapp. Quantum pseudo-telepathy. Found. Phys., 35:1877–1907, 2005.

[19] A. M. Childs and J. M. Eisenberg. Quantum algorithms for subset finding. Quantum Information and Computation, 5:593, 2005.

[20] A. Church. An unsolvable problem of elementary number theory. American Journal of Mathematics, 58:345–363, 1936.

[21] R. Cleve and D. P. DiVincenzo. Schumacher's quantum data compression as a quantum computation. Phys. Rev. A, 54(4):2636–2650, Oct 1996.

[22] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. In Proceeedings of the forth Annual ACM Symposium on Theory of Computing, pages 73–80, 1973.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. The MIT Press, 2^{nd} edition, 2001.

[24] J-M. Dautelle. JScience. http://www.jscience.org/.

[25] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. Proc. R. Soc. Lond. A, 400:97, 1985.

[26] D. Deutsch. Quantum computational networks. Proc. R. Soc. Lond. A, 425:73, 1989.

[27] D. Deutsch, A. Barenco, and A. Ekert. Universality in quantum computation. Proc. R. Soc. Lond., 449(1937):669–677, 1995.

[28] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. Proc Roy Soc Lond A, 439:553–558, 1992.

[29] B. D'Hooghe, J. Pykacz, and R. R. Zapatrin. Quantum computation of fuzzy numbers. Int. J. Theor. Phys., 43(6):1423–1432, 2004.

[30] P. A. M. Dirac. The principles of quantum meachnics. Oxford University Press, 1958.

[31] J. Du, X. Xu, H. Li, X. Zhou, and R. Han. Playing prisoner's dilemma with quantum rules. Fluctuation and Noise Letters, 2(4):R189, 2002.

[32] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? Phys. Rev., 47(10):777–780, May 1935.

[33] J. Eisert. Entanglement in Quantum Information Theory. PhD thesis, University of Potsdam, 2001.

[34] J. Eisert, M. Wilkens, and M. Lewenstein. Quantum games and quantum strategies. Phys. Rev. Lett., 83:3077–3080, October 1999.

[35] A. Ekert. Quantum cryptography based on Bell's theorem. Phys. Rev. Lett., 67:661–663, 1991.

[36] T. S. Ferguson. Game theory. Lecture notes at the Mathematics Department, University of California, 2005. http://www.math.ucla.edu/˜tom/Game˙Theory/Contents.html.

[37] R. P. Feynman. Simulating physics with computers. Int. J. Theor. Phys., 21(467):467, 1982.

[38] A. P. Flitney and D. Abbott. An introduction to quantum game theory. Fluctuation and Noise Letters, 2(4):R175–R188, 2002.

[39] A. P. Flitney and D. Abbott. Quantum models of Parrondo's games. Physica A, 324(1-2):152–156, 2003.

[40] A. P. Flitney, J. Ng, and D. Abbott. Quantum Parrondo's games. Physica A, 314:35, 2002.

[41] L. Fortnow. One complexity theorist's view of quantum computing. Theor. Comput. Sci., 292(3):597–610, 2003.

[42] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[43] S. Gay. Quantum programming languages: Survey and bibliography. Bulletin of the European Association for Theoretical Computer Science, 2005.

[44] I. Glendinning and B. Ömer. Parallelization of the QC-lib quantum computer simulator library. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, Parallel Processing and Applied Mathematics, volume 3019 of Lecture Notes in Computer Science, pages 461–468. Springer, 2004.

[45] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java(TM) Language Specification. Prentice Hall PTR, 3rd edition edition, 2005.

[46] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. Phys. Rev. Lett., 79:325–328, 1997.

[47] L. K. Grover. Synthesis of quantum superpositions by quantum computation. Phys. Rev. Lett., 85(6):1334–1337, Aug 2000.

[48] D. Harmer, G.P. Abbott. Parrondo's paradox. Statistical Science, 14:206–213, 1999.

[49] G. P. Harmer and D. Abbott. Losing strategies can win by parrondo's paradox. Nature, 402(6764):864, 1999.

[50] M. Hirvensalo. Quantum computing. Springer, 2001.

[51] A. S. Holevo. Statistical structure of Quantum theory, volume m67 of Lecture Notes in Physics. Springer-Verlag, 2001.

[52] J. E. Hopcroft and J. D. Ullman. Wprowadzenie do teorii automatw, jzykw i oblicze. Wydawnictwo Naukowe PWN, 2003.

[53] R. A. Horn and C. R. Johnson. Matrix Analysis. Cambridge University Press, 1985.

[54] P. Høyer, J. Neerbek, and Y. Shi. Quantum complexities of ordered searching, sorting, and element distinctness. In Proceedings of the 27th International Colloquium on Automata, Languages and Programming, pages 62–73, 2001.

[55] J. Hughes. Why functional programming matters. The Computer Journal, 32(2):98–107, 1989.

[56] G. Hutton. Programming in Haskell. Cambridge University Press, 2007.

[57] Moores law: Intel microprocessor transistor count chart. http://intel.com/museum/archives/history˙docs/mooreslaw.htm.

[58] J. Karczmarczuk. Structure and interpretation of quantum mechanics: a functional framework. In Proceedings of the ACM SIGPLAN workshop on Haskell, pages 50–61. ACM Press, 2003.

[59] B. W. Kernighan and D. M. Ritchie. Jezyk ANSI C. Klasyka Informatyki. Wydawnictwa Naukowo-Techniczne, 2004.

[60] M. Keyl. Fundamentals of quantum information theory. Phys. Rep., 5(369):431–548, 2002.

[61] E. Klarreich. Playing by quantum rules. Nature, 414:244–245, 2001.

[62] H. Klauck. Quantum time-space tradeoffs for sorting. In STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, pages 69–76. ACM Press, 2003.

[63] E. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.

[64] D. E. Knuth. The Art of Computer Programming: Sorting and Searching, volume 3. Addison-Wesley, 3 edition, 1998.

[65] A. Kocielski. Teoria oblicze: Wykady z matematycznych podstaw informatyki. Wydawnictwo Uniwersytetu Wrocawskiego, 1997.

[66] J. Koŝík. Two models of quantum random walk. CEJP, 4:556–573, 2003.

[67] J. Koŝík, J. A. Miszczak, and V. Bužek. Quantum Parrondo's game with random strategies. Journal of Modern Optics, 54:2275–2287, 2007.

[68] C. F. Lee and N. F. Johnson. Parrondo games and quantum algorithms. arXiv.org:quant-ph/0203043, 2002.

[69] C. F. Lee and N. F. Johnson. Game-theoretic discussion of quantum state estimation and cloning. Physics Letters A, 319(5-6):429–433, 2003.

[70] C. F. Lee, N. F. Johnson, F. Rodriguez, and L. Quiroga. Quantum coherence, correlated noise and Parrondo games. Fluctuation and Noise Letters, 2(4):L293–L298, 2002.

[71] W. Mauerer. Semantics and simulation of communication in quantum programming. Master's thesis, University Erlangen-Nuremberg, 2005.

[72] D. A. Meyer. Quantum strategies. Phys. Rev. Lett., 82(5):1052–1055, Feb 1999.

[73] J. A. Miszczak. Efficient quantum algorithm for factorization. Archiwum Informatyki Teoretycznej i Stosowanej, 2(14), 2002.

[74] J. A. Miszczak. Spltanie w algorytmach kwantowych ukrytej podgrupy. Master's thesis, University of Silesia, 2003.

[75] J. A. Miszczak. Description and visualisation of quantum circuits with XML. Archiwum Informatyki Teoretycznej i Stosowanej, 17(4), 2005.

[76] J. A. Miszczak. Initialisation of quantum registers based on probability distribution. Technical report, IITiS PAN, 2007. http://zksi.iitis.gliwice.pl/wiki/projects:kulka.

[77] J. A. Miszczak and P. Gawron. Numerical simulations of mixed states quantum computation. Int. J. Quant. Inf., 3(1):195–199, 2005. quant-ph/0406211.

[78] J. A. Miszczak and P. Gawron. Quantum implementation of Parrondo paradox. Fluctuation and Noise Letters, 5:L471–L478, 2005.

[79] J. A. Miszczak and Z. Puchaa. Quantum radix sorting algorithm. in preparation, 2007.

[80] J. C. Mitchell. Concepts in programming languages. Cambridge University Press, 2003.

[81] W. Mlak. Wstp do teorii przestreni Hilberta, volume 35 of Biblioteka Matematyczna. Pastwowe Wydawnictwo Naukowe, 1987.

[82] H. Mlnařík. Operational semantics of quantum programming language LanQ. Technical report, Faculty of Informatics, Masaryk University, Brno, Czech Republic, June 2007. Faculty of Informatics, Masaryk University.

[83] H. Mlnařík. Quantum Programming Language LanQ. PhD thesis, Masaryk University, 2007.

[84] G. E. Moore. Cramming more components onto integrated circuits. Electronics, 38:114–117, 1965.

[85] M. Mosca. Quantum Computer Algorithms. PhD thesis, Wolfson College, University of Oxford, 1999.

[86] M. Möttönen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa. Quantum circuits for general multiqubit gates. Phys. Rev. Lett., 93(13):130502, Sep 2004.

[87] M. Möttönen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa. Transformation of quantum states using uniformly controlled rotations. Quantum Information & Computation, 5(6), 2005.

[88] R. Nagarajan, N. Papanikolaou, and D. Williams. Simulating and compiling code for the sequential quantum random access machine. Electronic Notes in Theoretical Computer Science, 170:101–124, 2007.

[89] M. A. Nielsen and I. L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2000.

[90] H. Nishimura and M. Ozawa. Computational complexity of uniform quantum circuit families and quantum turing machines. Theor. Comput. Sci., 276:147–181, 2002. quant-ph/9906095.

[91] B. Ömer. A procedural formalism for quantum computing. Master's thesis, Vienna University of Technology, 1998.

[92] B. Ömer. Quantum programming in QCL. Master's thesis, Vienna University of Technology, 2000.

[93] B. Ömer. Structured Quantum Programming. PhD thesis, Vienna University of Technology, 2003.

[94] C. H. Papadimitriou. Computational complexity. Addison-Wesley Publishing Company, 1994. Polish translation: Zoono obliczeniowa, Wydawnictwa Naukowo-Techniczne, 2002.

[95] T. Parr. The Definitive ANTLR Reference. The Pragmatic Bookshelf, 2007.

[96] J. M. R. Parrondo, G. P. Harmer, and D. Abbott. New paradoxical games based on brownian ratchets. Phys. Rev. Lett., 85:5226–5229, 2000.

[97] E. W. Piotrowski and J. Sadkowski. An invitation to quantum game theory. International Journal of Theoretical Physics, 42(5):1089–1099, 2003.

[98] E. Ponka. Wykady z teorii gier. Wydawnictwo Politechniki lskiej, 2001.

[99] E. Rasmusen. Games and Information: An Introduction to Game Theory. Wiley-Blackwell, 2006. http://www.rasmusen.org/GI/.

[100] A. Sabry. Modeling quantum computing in Haskell. In ACM SIGPLAN Haskell Workshop, 2003.

[101] R. Sedgewick and M. Schidlowsky. Algorithms in Java. Addison-Wesley Professional, 2002.

[102] P. Selinger. A brief survey of quantum programming languages. In Proceedings of the 7th International Symposium on Functional and Logic Programming, volume 2998 of Lecture Notes in Computer Science, pages 1–6, 2004.

[103] P. Selinger. Towards a quantum programming language. Mathematical Structures in Computer Science, 14(4):527–586, 2004.

[104] V. V. Shende, I. L. Markov, and S. S. Bullock. Minimal universal two-qubit controlled-NOT-based circuits. Phys. Rev. A, 69:062321, 2004. quant-ph/0308033.

[105] J. C. Shepherdson and H. E. Strugis. Computability of recursive functions. Journal of the ACM, 10(2):217–255, April 1963.

[106] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computers. SIAM J. Computing, 26:1484–1509, 1997.

[107] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pages 124–134. IEEE Computer Society Press, 1994.

[108] P. W. Shor. Why haven't more quantum algorithms been found? Journal of the ACM, 50(1):87–90, 2003.

[109] P. W. Shor. Progress in quantum algorithms. Quantum Information Processing, 3(1-5), 2004.

[110] K. M. Svore, A. W. Cross, A. V. Aho, I. L. Chuang, and I. L. Markov. Toward a software architecture for quantum computing design tools. In P. Selinger, editor, Proceedings of the 2nd International Workshop on Quantum Programming Languages, 2004.

[111] K. M. Svore, A. W. Cross, I. L. Chuang, A. V. Aho, and I. L. Markov. A layered software architecture for quantum computing design tools. IEEE Computer, pages 74–83, January 2006.

[112] T. Toffoli. Bicontinuous extension of reversible combinatorial functions. Math. Syst. Theory, 14:13–23, 1981.

[113] D. Unruh. Quantum programming languages. Informatik – Forschung und Entwicklung, 21(1–2):55–63, 2006.

[114] R. Ursin, F. Tiefenbacher, T. Schmitt-Manderbach, H. Weier, T. Scheidl, M. Lindenthal, B. Blauensteiner, T. Jennewein, J. Perdigues, P. Trojek, B. Ömer, M. Fürst, M. Meyenburg, J. Rarity, Z. Sodnik, C. Barbieri, H. Weinfurter, and A. Zeilinger. Entanglement-based quantum communication over 144 km. Nature Physics, 3:481–486, 2007.

[115] A. van Tonder. A lambda calculus for quantum computation. SIAM Journal on Computing, 33(5):1109–1135, 2004.

[116] J. J. Vartiainen, M. Mottonen, and M. M. Salomaa. Efficient decomposition of quantum gates. Phys. Rev. Lett., 92:177902, 2004.

[117] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations. Phys. Rev. A, 54:147, 1996.

[118] J. von Neuman. Mathematical foundations of quantum meachnics. Princeton University Press, 1954.

[119] H. Weimer. The C library for quantum computing and quantum simulation. http://www.libquantum.de/.

[120] R. F. Werner. Quantum states with Einstein-Rosen-Podolsky correlations admitting a hidden-variable model. Phys. Rev. A, 40:4277–4281, 1989.

[121] S. Wgrzyn, J. Klamka, and J. A. Miszczak. Kwantowe Systemy Informatyki. 2003. http://www.iitis.gliwice.pl/˜miszczak/notes/.

[122] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. Nature, 299:802–803, 1982.

[123] P. Wycisk. Programowanie komputerw kwantowych. Master's thesis, Silesian University of Technology, 2006. In Polish.

[124] A. Yao. Quantum circuit complexity. In Proceedings of the 34^{th} IEEE Symposium on Foundations of Computer Science, pages 352–360. IEEE Computer Society Press, 1993.

[125] Yong-Sheng Zhang, Ming-Yong Ye, and Guang-Can Guo. Conditions for optimal construction of two-qubit nonlocal gates. Phys. Rev. A, 71(6):062331, 2005.

[126] Quantum Systems of Informatics Group at IITiS PAN: Projects. http://zksi.iitis.gliwice.pl/wiki/projects:kulka.

[127] U. Zwick. Scribe notes of the course boolean circuit complexity. http://www.math.tau.ac.il/˜zwick/scribe-boolean.html.