Institut für Theoretische Informatik
Leibniz Universität Hannover

CQ: A high-level imperative classical-quantum programming language

Bachelor's Thesis

Lennart Binkowski
10011560

Supervisor:

Prof. Dr. Heribert Vollmer

January 14, 2025

Abstract

In this thesis, I propose CQ, an imperative high-level programming language for simple boolean and integer manipulation via classical and quantum hardware. CQ inherits its core syntax directly from C, but dispenses with classical low-level control in favor of a joint abstraction of classical and quantum variables. For this purpose, I introduce the `quantum` type qualifier, marking variables which are to be realized as quantum states of multi-qubit registers; the interpretation of quantum states as (superpositions of) classical booleans or integers is always understood w.r.t. the computational basis of the multi-qubit system. By fixing a reference basis, arithmetic and bitwise operations can be seamlessly applied to both quantum and classical variables. Furthermore, the classical control structures `if`/`else`-statements and `switch`-statements are suitably extended to conditions and statements involving quantum variables, utilizing controlled quantum gates as counterpart to classical implementations. Functions that are defined for classical parameters and only involve quantizable operations may also be applied to quantum variables of the same type. Beside these hybrid functionalities, manipulating the phase of quantum variables (via the `phase` keyword) and measuring them (via the `measure` keyword) are two additional, purely quantum operations. Alongside the conceptualization of the CQ-language, I provide a prototypical parser and semantic analyzer to process CQ-source code, and discuss its application to Grover's famous quantum search algorithm as a concrete example.

Contents

CHAPTER 1

Introduction

Quantum computing is an emerging field at the intersection of quantum mechanics and computer science with an impressive track record of recent advancements in both soft- and hardware development. In a nutshell, quantum computing describes the approach of utilizing fundamental quantum phenomena such as superposition and entanglement to perform calculations, aiming at improved computational power over conventional, classical computers. This quest necessitates, on the one hand, the development of hardware with precise control over quantum systems, and on the other, the design of algorithms that incorporate the enlarged set of fundamental principles. While this gives an application-oriented characterization of quantum computing, the field arguably has a rather theoretical origin: Benioff's [1] and Deutsch's [2] pioneering work on quantum Turing machines during the early eighties, simultaneously introducing the idea of quantum computers and spawning the field of quantum complexity theory [3].

But also the practical implications of realized quantum computers were also discussed early on. In particular, Feynman [4] proposed to simulate quantum systems with the aid of other quantum systems, thus essentially formulating the concept of a quantum computer. And indeed, nowadays, the discipline of quantum simulation [5] constitutes the most promising area for practical quantum advantage: solving a problem of practical relevance with quantum computers significantly faster or with higher quality than any existing classical (super-)computer could. Use cases of quantum simulation cover disciplines such as chemistry [6], material sciences [7], and many more.

Furthermore, quantum computers are not limited to solving quantum-mechanical problems alone. From a theoretical standpoint, it is well-established that quantum computers can simulate classical computers efficiently [8], making them viable candidates for addressing purely classical problems as well. While merely simulating a classical algorithm on a quantum computer will not offer practical advantages due to higher operational costs, slower gate execution speeds, and the increased complexity of error correction, specialized quantum algorithms have the potential to solve certain problems faster than any known classical algorithm. Several quantum primitives such as the Deutsch-Jozsa algorithm [9] or Simon's algorithm [10] solve artificial problems provably

exponentially faster than any possible deterministic or even probabilistic classical algorithm. Although constituting impressive results in the field of quantum complexity theory, the problems solved by these algorithms are typically of little use in practice. However, Simon's algorithm also served as the inspiration for Shor's famous quantum algorithms for efficient prime factorization and calculating discrete logarithms [11], two tasks of high practical relevance, especially in many public-key cryptography schemes. Another pivotal discovery has been Grover's algorithm [12], establishing a quadratic speed-up over exhaustive, unstructured search. Assuming P $\neq$ NP, Grover's algorithms and its generalization, such as quantum amplitude amplification [13], are leading candidates for providing an asymptotic quadratic speed-up for general NP-complete decision problems.

It is important to note that a true quantum advantage is still pending since the hardware, necessary to reliably execute quantum routines such as Shor's algorithms on instance sizes of practical relevance, has not yet been built. Manufacturing a large-scale, fault-tolerant quantum computer faces various technical challenges such as maintaining quantum coherence and properly handling error correction. Both the construction and maintenance of quantum computers are significantly more involved than those of classical computers, primarily due to the quantum systems' higher susceptibility to environmental noise; today's quantum computer – also called noisy intermediate-scale quantum (NISQ) devices [14] – are indeed very restricted, both in storage and runtime capacity, and are severely suffering from noise effects. Consequently, most proposed quantum algorithms are not yet (reliably) testable on real devices. Nevertheless, their capabilities can still be assessed by various methods besides asymptotic runtime analysis, such as classical simulation for small problem instances and recently developed hybrid frameworks [15, 16, 17] that are able to give precise runtime predictions, even for real-world instance sizes. Hence the development and evaluation of quantum software can fortunately proceed mostly independently of advancements in quantum hardware.

Indeed, the software landscape is flourishing with several specialized algorithms and general frameworks being formulated on a weekly basis. Most algorithms can be loosely classified either as pure (far-term) quantum algorithms, demanding fault-tolerant quantum computers, or NISQ-friendlier hybrid algorithms that outsource certain subroutines to classical computers (mostly variational quantum algorithms [18]). Since NISQ-friendly algorithms aim at addressing hardware-specific shortcomings explicitly, their design is typically more involved and harder to abstract away from low-level control of the quantum hardware. In contrast, far-term algorithms such as Grover's or Shor's algorithm may be formulated more problem- than hardware-oriented; quantum operations are typically applied as conceptual constructs, allowing, in principle, for a high-level description of the algorithms' subroutines that does not require any knowledge about the quantum hardware, merely about the available primitives.

The rapidly increasing number of quantum algorithms is complemented by a similarly fast growing stack of (mostly open-source) quantum computing software development frameworks such as Xanadu's PennyLane [19], IBM's Qiskit [20], or Google's Cirq [21], and quantum cloud services such as Amazon Braket [22] or Microsoft's Azure Quantum [23]. Most services come as Python modules which is why Python is often also recognized as a quantum programming language. However, there are specialized autonomous programming languages used in the background such as IBM's OpenQASM [24] and Microsoft's Q# [25]. Other particularly influential quantum languages are Quipper [26] (functional language embedded in Haskell), QCL [27] (imperative, C-based language), and Silq [28]. Another honorable mention is the recently introduced Qrisp [29] (python-based). These programming languages are crafted to facilitate both low-level, gate-wise control and high-level conceptualization through predefined primitives. For instance, OpenQASM is referred to as a "broader and deeper quantum assembly", emphasizing the language's versatility, but also indicating its extensive scope. This extensive scope can make these languages quite challenging for newcomers to learn. Additionally, despite the many compound routines already embedded in these languages, practitioners ultimately cannot avoid delving into low-level gate operations, making it more difficult to focus on high-level algorithmic design.

In this thesis, I explicitly tackle these issues, being present in most existing quantum programming languages. I introduce the CQ programming language for classical and quantum manipulation of booleans and integers. The language uses a syntax which is, restricted to classical variables, a proper subset of the pioneering C programming language with some quality-of-life changes, making the transition from classical software design as easy as possible. Additionally, CQ allows declaring quantum variables, which may be initialized and processed in quantum superposition, and manipulating them together with classical variables via bitwise and arithmetic operations. The symbols for those operators are the same as in plain C and are suitably overloaded to handle also quantum operands. Furthermore, the classical control structures `if`/`else`- and `switch`-statements are extended to allow for quantum conditions and statements. Logical comparison operators are extended for relating classical to quantum and quantum to quantum variables. Statements controlled on quantum conditions are realized as (multi-)controlled quantum circuits, where the control pattern can be directly inferred from the provided boolean expression. By overloading elementary operations and control structures, CQ is able to provide handling of classical and quantum variables with only three additional non-C keywords: `quantum` (type qualifier for quantum variables), `measure` (measurement of a quantum variable), and `phase` (addressing the phase of a quantum variable). With this drastically reduced set of additional keywords, CQ aims to be easy for beginners to learn and to become a proper, classical programmer-friendlier alternative to Silq which offers similar features. Most importantly, quantum hardware details such as the implementation of integers as quantum states of multi-qubit registers or the gate decomposition of subroutines like integer comparison are abstracted away.

The second chapter establishes all necessary preliminaries for designing a custom quantum programming language. In Section 2.1, I detail general principles of programming language design. I examine the necessary building blocks, present in every proper programming language, and summarize the usual workflow associated with the implementation of a new programming language. Subsequently, I formulate the foundations of quantum computing in Section 2.2, following the axiomatic description of quantum mechanics. I mainly focus on concepts that are incorporated within CQ's scope, but also mention and discuss additional phenomena that may be of independent interest.

In the third chapter, I investigate quantized versions of several classical primitives. Section 3.1 deals with bitwise operations on integers and with their quantum-computational analogues. In Section 3.2, I cover all basic arithmetic operations for integers and their realizations as quantum circuits. I conclude this chapter with the treatment of boolean expressions and conditions in Section 3.3. The focus lies on translating classical conditioning to suitable quantum control structures, making extensive use of manipulation of boolean formulas.

The fourth chapter constitutes the heart of this thesis: the design of CQ, strictly following the workflow detailed in Section 2.1. In Section 4.1, I define the C-like vocabulary of CQ and construct a suitable lexer. The lexer is implemented using the open-source flex-software [30]. Section 4.2 then defines CQ's grammar which allows me to construct a parser for syntax checking. The parser is written in GNU's Bison [31] framework. Due to its low complexity, CQ's syntax and semantics, as detailed in Section 4.3, can be analyzed simultaneously in a single pass. That is, the parser only accepts syntactically and semantically correct programs. The source code which is being described in this chapter can be found under [32].

As a proof of concept, I give a CQ implementation of Grover's famous search algorithm in the fifth chapter. First, I recap Grover's algorithm in Section 5.1 and already characterize its building blocks to match them with CQ primitives in Section 5.2. Here, I provide a concrete implementation and investigate all intermediate as well as the final result generated from the initial source code. In Section 5.3, I compare the effort of implementing Grover's algorithm in CQ to formulations found in existing frameworks.

In the last chapter, I draw a conclusion and present an outlook on further investigations. In particular, I address concrete extensions to the CQ programming language and their implications on the scope of the language. Furthermore, I comment on possible connections between CQ and other parts of the quantum software stack, highlighting potential next steps from a software engineering point of view.

CHAPTER 2

Preliminaries

2.1 Principles of Programming Language Design

Developing and implementing a new programming language is a complex task which involves not only the abstract formulation of consistent grammar rules but also their thorough implementation across several stages. The software stack for high-level programming languages like C, Java, or Python is far more sophisticated than a simple program that accepts or rejects source code. It also incorporates features such as debugging capabilities, the generation of intermediate representations, code optimization, instructions for executing code on hardware (for interpreted languages like Python), and further translation into lower-level languages (for compiled languages like C). The scope of a programming language's implementation strongly depends on design intent behind the language. For instance, early programming languages were designed with a strong emphasis on efficiency due to the limited computational power of early hardware. This focus on efficiency minimized overhead and avoided extensive abstraction from the hardware level. Consequently, there was little need for intermediate translation units, as source code was often executed directly on hardware. As advancements in hardware fabrication made computational resources more abundant, the focus shifted toward greater abstraction and portability. This evolution led to more human-readable syntax, which in turn increased the semantic gap between high-level programming languages and machine code. Bridging this gap necessitated the development of additional intermediate layers. As a result, translating high-level source code into machine code directly became increasingly cumbersome and error-prone. Instead, compilers and interpreters were developed to break down high-level source code into intermediate representations systematically, which could then be translated into machine code more effectively and reliably. Together with these translation units, debugging tools and sophisticated code optimization techniques were introduced to improve the quality of the code before its execution. All together, the history of programming language design perfectly showcases how each design choice directly influences the entire software stack that is behind each language.

Two core features of every programming language are its syntax and its semantics. Syntax, or grammar, defines the formal rules and structure of the language, dictating how symbols, keywords, and operators must be arranged to form valid statements or expressions. Semantics, on the other hand, deals with the meaning behind the syntactically correct constructs, specifying what actions or operations a piece of code will perform eventually. As most programming languages are formal languages (exceptions are, e.g., the visual programming languages like Snap! [33]), their syntaxes are indeed formal grammars. That is, on the most fundamental level, one has to specify four quantities: a set of variables/non-terminals which are used internally for production rules and are not visible to the programmer, an alphabet of terminal symbols, i.e. the set of all symbols that are allowed to appear in the source code, a set of production rules that map combinations of non-terminals and terminals to other combinations, and a start variable (again invisible to the programmer). In many programming languages, there is a clear separation between names of variables, which can be arbitrarily long concatenations of symbols from a fixed alphabet, and other "words" of fixed appearance such as operators, keywords, and delimiters. This is reflected in the common approach to introduce an additional layer of grammar verification: the lexer. The lexer, short for lexical analyzer, is responsible for analyzing the raw sequence of characters in the source code and transforming them into a stream of meaningful tokens. These tokens are atomic units of syntax, such as identifiers, keywords, operators, and punctuation, that conform to the language's terminal symbols. By separating the recognition of individual lexical elements from the higher-level syntactic structure, the lexer simplifies the parsing process. It ensures that the input adheres to the language's lexical rules, already identifying errors such as unrecognized symbols. Additionally, the lexer typically eliminates whitespace and comments, streamlining the input for subsequent stages while preserving the code's meaning.

After the lexer has transformed the raw source code into a stream of tokens, the parser takes over. The parser is responsible for analyzing the token sequence according to the language's syntax rules, as defined by its formal grammar. Its primary goal is to determine whether the token stream forms a syntactically valid structure, i.e. whether the source code is a valid "sentence" in the given formal language, and to build a corresponding abstract representation, often called a parse tree or abstract syntax tree (AST). At its core, the parser operates based on a set of production rules, which describe how terminal and non-terminal symbols combine to form higher-level constructs, such as expressions, statements, or program blocks. Using these rules, the parser recursively applies a series of reductions (in bottom-up parsing) or expansions (in top-down parsing) to map the token sequence to the grammar's start variable, thereby constructing the AST either from its leafs or its root. Introducing a single placeholder for all the previously tokenized identifiers and working entirely on the level of valid tokens drastically reduces the set of production rules.

After the parser has verified the syntactic correctness of the source code and generated an AST, the semantic analyzer takes the next step to ensure the program's meaning is valid and consistent. Semantic analysis involves checking the AST against the language's semantic rules, which go beyond syntax to enforce logical correctness. These rules may include verifying type compatibility, ensuring that variables are declared before use, enforcing access control for private or protected members, and resolving function calls with the correct parameters and return types. Semantic analysis typically augments the AST with additional information, such as type annotations or symbol table references, creating a more detailed intermediate representation (IR). The symbol table is a crucial component in this process, mapping identifiers like variable names and function names to their respective declarations, scopes, and attributes. Additionally, semantic analysis detects issues that cannot be identified by the parser alone, such as type mismatches in assignments, undefined or duplicate identifiers, and violations of scope rules. Note that, depending on the concrete syntax and semantics, it may be possible to evaluate syntax and semantics at the same time, thus creating an enriched AST already during the parsing process; this is then called a one-pass compiler.

Once the semantic analysis has verified the program's correctness and enriched the IR with semantic details, possible next steps are code optimization and code generation. First and foremost, this is the case for compiled languages such as C, but also for interpreted languages like Python just-in-time (JIT) compilation, which is currently rising in popularity, demands similar steps. Code optimization involves improving the IR to make the resulting program more efficient in terms of execution speed, memory usage, or other performance metrics, without altering its semantics. Before applying any code optimizations, the initially tree-structured IR is linearized to an abstract instruction list. Subsequent optimization techniques may include eliminating redundant calculations, inlining functions, unrolling loops, or reorganizing code for better utilization of hardware features like CPU pipelines or caches. Following optimization, the code generation phase translates the optimized IR into a target language, which is typically low-level machine code or assembly. This step involves mapping high-level constructs to hardware-specific instructions, allocating registers, managing memory layout, and embedding runtime support elements. A key component of code generation is instruction selection, where abstract operations in the IR are matched to specific instructions in the target architecture, and instruction scheduling, which arranges these instructions to minimize execution stalls. The result of this stage is a complete and optimized binary or executable, ready for execution on the targeted hardware.

2.2 Foundations of Quantum Computing

Quantum computing is a sub-discipline of quantum information theory, aiming at utilizing quantum-mechanical concepts as computational tools. Formulated in the language of quantum mechanics and heavily influenced by classical computer science, the theory of quantum computing delivers mathematically elegant and concise descriptions of both fundamental quantum phenomenology and application-relevant computational problems, along with algorithmic frameworks to address them. Most concepts of practical relevance can be directly derived from the postulates of modern quantum mechanics, relying on basic linear algebra. In doing so, I primarily follow Nielsen and Chuang's highly influential book on *Quantum Computation and Quantum Information* [34] for the remainder of this chapter.

Postulate 1. The *state space* of any isolated physical system is given by a complex Hilbert space (a vector space with an inner product). The system's state is completely described by a *state vector*, a vector of unit length in the system's state space.

The first postulate defines the underlying mathematical structure of quantum mechanics – and hence quantum computing. It places the theory right in the fields of linear algebra and functional analysis. For the scope of this thesis, however, linear algebra will prove to be a sufficient tool as I will only deal with finite-dimensional spaces. Quantum mechanics often comes with its own, somewhat peculiar notation: the *bra-ket* or *Dirac notation*. Given some Hilbert space $\mathcal{H}$, a unit vector, i.e. some system's state vector, is denoted as *ket* $|\phi\rangle \in \mathcal{H}$. The symbol between the vertical bar and the right angle bracket may be arbitrary. In particular, I may use descriptive labels such as numbers directly for the vector without sub- or superscripting. The inner product with another ket $|\psi\rangle \in \mathcal{H}$ is denoted as $\langle\phi|\psi\rangle$.[1]

Note that Postulate 1 is not constructive; it does not dictate a system-describing Hilbert space, but merely asserts its existence. Accordingly, I will prescribe a certain state space rather than a physical system. Inspired by the bit, the most basic unit of classical information, one is seeking a suitable state space characterizing the most basic unit of quantum information. The simplest, non-trivial (complex) Hilbert space is the two-dimensional $\mathbb{C}^2$ and any system described by this state space is, in analogy to the bit, called a qubit (from **qu**antum **bit**). Physical realizations of qubits are two-level

[1] The dual notion of a ket is the *bra*. A bra $\langle\phi|$ denotes an element (of unit norm) in the dual space of $\mathcal{H}$, i.e. the space of (continuous) linear maps from $\mathcal{H}$ to $\mathbb{C}$. Hilbert spaces are naturally isometric (anti-)isomorphic to their dual spaces via the Riesz representation theorem which allows for uniquely identifying a bra $\langle\psi| \in \mathcal{H}^*$ with a ket $|\phi\rangle \in \mathcal{H}$ and vice versa. In this light, the inner product notation $\langle\phi|\psi\rangle$ can really be understood of a short-hand for $\langle\phi|\,(|\psi\rangle)$.

quantum systems such as the polarization direction of light [35, 36], electronic [37] and nuclear spins [38], atomic energy levels of neutral atoms [39] and trapped ions [40], and superconducting phases of Josephson junctions [41].

I introduce the notation $\mathbb{b} \coloneqq \{0, 1\}$ for classical bits, and $\mathbb{q} \coloneqq \mathbb{C}^2$ for qubits. The former can be embedded into the latter by choosing an orthonormal basis of $\mathbb{q}$ and labeling its two elements with $|0\rangle$ and $|1\rangle$, respectively; $\mathbb{q}$ may then be interpreted as the linear span of the two *bit states* $|0\rangle$ and $|1\rangle$, that is

$$\mathbb{q} \cong \{\alpha\,|0\rangle + \beta\,|1\rangle \;:\; \alpha, \beta \in \mathbb{C}\}. \tag{2.2.1}$$

This gives an easy mathematical grasp of the concept of (*coherent*) *superposition*: Besides the "classical" states $|0\rangle$ and $|1\rangle$ also all linear combinations of them are valid quantum state, as long as they are normalized to unit length, i.e. if $|\alpha|^2 + |\beta|^2 = 1$. While difficult to comprehend intuitively, quantum superpositions are thus simple mathematical objects. Furthermore, the interpretation of a given quantum state as superposition of other states is entirely based on what states one declares as reference states or, equivalently, which orthonormal basis of the underlying Hilbert space one chooses. Take, for example, the following two qubit states

$$|+\rangle \coloneqq \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \;\text{ and }\; |-\rangle \coloneqq \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle\,. \tag{2.2.2}$$

They are introduced as superpositions of the two bit states. However, $|+\rangle$ and $|-\rangle$ together constitute an equally valid orthonormal basis of $\mathbb{q}$ and, w.r.t. this basis, the bit states

$$|0\rangle = \frac{1}{\sqrt{2}}\,|+\rangle + \frac{1}{\sqrt{2}}\,|-\rangle \;\text{ and }\; |1\rangle = \frac{1}{\sqrt{2}}\,|+\rangle - \frac{1}{\sqrt{2}}\,|-\rangle \tag{2.2.3}$$

are actually superpositions; none of these interpretations is, in any sense, more correct than the other. However, in order to properly express calculations and as an interface with classical information, a choice has to be made, falling on $\{|0\rangle\,, |1\rangle\}$ as the reference basis.

The correspondence between quantum states and unit vectors in an associated Hilbert space which is introduced by Postulate 1 is actually not one-to-one. All physically observable properties of a quantum state, described by some $|\phi\rangle \in \mathcal{H}$, are equally validly encoded into every unit vector collinear to $|\phi\rangle$, i.e. by vectors of the form $\gamma\,|\phi\rangle$, where the scalar $\gamma \in \mathbb{C}$ is of unit magnitude and is called a *global phase*. Another way of of phrasing it is that the representing state vector is unique up to a global phase.[2]

[2]There is a more general formulation of the first postulate that avoids the appearance of the global phase all together: Any isolated physical system is still assigned a Hilbert space $\mathcal{H}$. However, the

Postulate 2. The state space of a composite physical system is the tensor product of the state spaces of the component physical system. Moreover, assuming an enumeration of individual system from 0 to $n-1$, if the i-th system is prepared in the state $|\phi_i\rangle$, the joint state of the total system is $|\phi_{n-1}\rangle \otimes \cdots \otimes |\phi_i\rangle \otimes \cdots \otimes |\phi_0\rangle$.

Unlike the first postulate, Postulate 2 gives an explicit state space construction, provided that the individual state spaces are already known. Applying it to an n-qubit system yields the state space

$$\mathfrak{q}^{\otimes n} \coloneqq \bigotimes_{i=0}^{n-1} \mathfrak{q} = \bigotimes_{i=0}^{n-1} \mathbb{C}^2 \cong \mathbb{C}^{2^n}. \tag{2.2.4}$$

Furthermore, the second postulate asserts that the joint state of individually prepared states is a (tensor) product state of the individual states. In particular, I may apply this instruction to the bit states of individual qubits. Assuming the i-th qubit being in the state $|x_i\rangle$, $x_i \in \mathbb{b}$, the n-qubit state is given by

$$|x_{n-1} \dots x_i \dots x_0\rangle \coloneqq |x_{n-1}\rangle \otimes \cdots \otimes |x_i\rangle \otimes \dots \otimes |x_0\rangle . \tag{2.2.5}$$

In fact, the set of all states of the form (2.2.5) constitutes itself an orthonormal basis of $\mathfrak{q}^{\otimes n}$, the *computational basis*

$$\{|\boldsymbol{x}\rangle \coloneqq |x_{n-1} \dots x_0\rangle \,:\, x_i \in \mathbb{b}\} = \{|\boldsymbol{x}\rangle \,:\, \boldsymbol{x} \in \mathbb{b}^n\}. \tag{2.2.6}$$

The computational basis is thus the embedding of the set of all n-bit strings into the n-qubit state space. Analogously to (2.2.1), this means that every n-qubit state can be expressed as a superposition of n-bit strings. Quite frequently, I will exploit the binary representation of integers to label the elements of the computational basis with consecutive integers rather than bit strings, that is

$$|x_{n-1} \dots x_0\rangle \cong |j\rangle\,,\ j = \sum_{i=0}^{n-1} x_i 2^i \Rightarrow \{|\boldsymbol{x}\rangle \,:\, \boldsymbol{x} \in \mathbb{b}^n\} \cong \{|j\rangle \,:\, j = 0, \dots 2^n - 1\}. \tag{2.2.7}$$

I highlight that Postulate 2 makes a statement about how to construct the joint state of known individual states, but not the other way around. This is no shortcoming of

system's state is described by a *density operator* $\rho \in \mathcal{L}(\mathcal{H})$, a positive operator of unit trace. A density operator which is also a projection is called a *pure state*, other states are called *mixed states*. A pure state can be written as $|\phi\rangle\langle\phi|$ for some normalized vector $|\phi\rangle \in \mathcal{H}$ onto whose span the pure state projects, establishing a connection to the initially given formulation of the postulate. Choosing instead of $|\phi\rangle$ a collinear unit vector $|\psi\rangle = \gamma\,|\phi\rangle$ gives rise to the same density operator as $|\psi\rangle\langle\psi| = |\gamma|^2\,|\phi\rangle\langle\phi| = |\phi\rangle\langle\phi|$, eliminating this unphysical degree of freedom.

the postulate, but indeed a generally ill-posed question: Consider, for example, the 2-qubit state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle \in \mathfrak{q}^{\otimes 2}, \tag{2.2.8}$$

which is one of the famous four *Bell states* [42]. It can be shown that there are exist no two single-qubit states $|\phi\rangle, |\psi\rangle \in \mathfrak{q}$ such that $|\Phi^+\rangle$ can be written as the tensor product of $|\phi\rangle$ with $|\psi\rangle$.[3] Nevertheless, $|\Phi^+\rangle$ is a perfectly valid and realizable quantum state [43]. This phenomenon that composite systems can exist in states in which single constituents cannot be fully described without considering the others is known as (*quantum*) *entanglement*. It is a distinct feature of quantum mechanics. Whether a state is entangled does not, in contrast to the question whether it is a superposition, depend on the choice of basis.

Postulate 3. The evolution of a closed quantum system with associated Hilbert space $\mathcal{H}$ is described by a unitary, linear operator $U \in \mathcal{L}(\mathcal{H})$. That is, if the system's initial state is $|\phi\rangle \in \mathcal{H}$, its evolved state is given by $U|\phi\rangle$.

The third postulate is again non-constructive, but crucially establishes the linearity of quantum mechanics.[4] Additionally, unitarity is imposed in order to maintain normalization of quantum states: Unitary operators $U \in \mathcal{L}(\mathcal{H})$ may be defined as satisfying $UU^\dagger = U^\dagger U = \mathbb{1}$, where $U^\dagger$ is the adjoint of U and $\mathbb{1} \in \mathcal{L}(\mathcal{H})$ is the identity operator. These properties are equivalent to preserving the Hilbert space's inner product and hence, especially, its norm. Therefore, a normalized state $|\phi\rangle$ indeed remains normalized after the application of some unitary U.

The set of linear operators over a d-dimensional complex Hilbert space $\mathcal{H}$ can be identified with the set of complex $d \times d$ matrices by choosing a (not necessarily orthonormal) basis $\{|j\rangle : j = 0, \ldots, d-1\}$ of the underlying Hilbert space and, for a given operator $A \in \mathcal{L}(\mathcal{H})$, recording the matrix elements $M_{ij} \coloneqq \langle i|A|j\rangle$ for all $i, j = 0, \ldots, d-1$. Conversely, given a $d \times d$ matrix M, the abstract operator can be constructed via

$$A = \sum_{i,j=0}^{d-1} M_{ij}\,|i\rangle\langle j| \in \mathcal{L}(\mathcal{H}). \tag{2.2.9}$$

[3] In mathematical terms, $|\Psi^+\rangle$ is called a *non-pure* tensor.

[4] A more refined version of the postulate states that the time evolution of the state of a closed quantum system is described by the *Schrödinger equation* $i\hbar\frac{\mathrm{d}|\phi\rangle}{\mathrm{d}t} = H|\phi\rangle$, where $\hbar$ is *Planck's constant* and $H \in \mathcal{L}(\mathcal{H})$ is the system's *Hamiltonian*.

The action of the operator $|\phi\rangle\langle\psi|$ on some state $|\xi\rangle$ is given by $|\phi\rangle\langle\psi|\xi\rangle$ (first taking the inner product of $|\psi\rangle$ with $|\xi\rangle$ and then outputting the vector $|\phi\rangle$, scaled by this factor), i.e. the symbols are simply concatenated. For an n-qubit system, the allowed evolutions are therefore described by unitary $2^n \times 2^n$ matrices, where I fix again the computational basis as the preferred basis to translate between the abstract operators and concrete matrices. In analogy to classical logical gates and circuits, these multi-qubit evolutions are primarily called *quantum gates* and *circuits*. I may also speak of gates/circuits directly, when it is clear from the context whether classical or quantum gates/circuits are addressed.

Most importantly, the tensor structure of $\mathbb{q}^{\otimes n}$ is also present in $\mathcal{L}(\mathbb{q}^{\otimes n})$. For example, if I only want to apply a single-qubit gate U (thus represented by a 2×2 matrix) to, say, the i-th qubit contained in a composite n-qubit system, the corresponding n-qubit gate is of the form

$$U_i \coloneqq \mathbb{1} \otimes \cdots \otimes \mathbb{1} \otimes \underset{\substack{\uparrow \\ i\text{-th position}}}{U} \otimes \mathbb{1} \otimes \cdots \otimes \mathbb{1} \in \mathcal{L}(\mathbb{q}^{\otimes n}). \tag{2.2.10}$$

Similarly, also m-qubit gates with $1 < m < n$ can be embedded as n-qubit gates. In the same spirit, applying multiple quantum gates each acting on distinct qubits, simultaneously in an n-qubit register is mathematically described by taking their tensor product, filled up with an identity for each unaffected qubit. In contrast, applying quantum gates sequentially is described by their usual matrix/operator product.

In addition to the matrix representation of quantum gates, a visual representation, especially of longer gate sequences, is often insightful. A common depiction are quantum circuit diagrams which are inspired by and named after the logic circuit diagrams. Here, individual qubits are represented as wires, and quantum gates are depicted as boxes, connected to the qubits/wires they operate on. Consider, e.g., a two qubit system $\mathbb{q}^{\otimes 2}$, initialized in the $|00\rangle$ state, whose components I address with q_0 and q_1. Now, I wish to apply first a single-qubit gate $U \in \mathcal{L}(\mathbb{q})$ on the zeroth qubit q_0, followed by a two-qubit gate $V \in \mathcal{L}(\mathbb{q}^{\otimes 2})$, acting on both q_0 and q_1. The corresponding quantum circuit diagram is shown in Fig. 2.1.

One particular type of quantum gates will play a crucial role throughout this thesis: controlled gates. Consider again a two-qubit system with components q_0 and q_1, and let U be a single-qubit gate. Applying U on q_0 controlled on q_1 being unset/set are the abstract operations

$$\overline{\mathtt{C}}_1(U_0) \coloneqq |0\rangle\langle 0| \otimes U + |1\rangle\langle 1| \otimes \mathbb{1} \text{ and } \mathtt{C}_1(U_0) \coloneqq |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes U. \tag{2.2.11}$$

q_0 is called the target qubit of this operation and q_1 is its control qubit. These constructions readily extend to multi-controlled multi-target gates. Intuitively, controlled gates

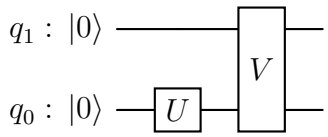


Figure 2.1: Quantum circuit diagram for a two-qubit system. Both qubits are initialized in the $|0\rangle$ state. First, a single-qubit gate U is applied to the zeroth qubit. Second a two-qubit gate V is applied to both qubits.

act non-trivially only on some portion of a given input superposition. The behavior is thereby controlled by the state of a certain subsystem of the input, relative to the classical bit states (or more generally, relative to the computational basis). Controlled gates are so common that instead of depicting them as regular multi-qubit gates they possess their own notation where the control qubits are vertically connected to the actual gate that is applied to the target qubit(s). The control qubits receive white/black circles when the control is on their $|0\rangle/|1\rangle$ state as depicted in Fig. 2.2.

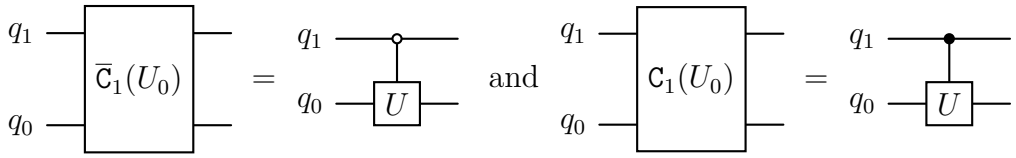


Figure 2.2: Quantum circuit diagram definitions for controlled gates. Controlling a single-qubit operation U on the zeroth qubit on the first qubit being in the state $|0\rangle/|1\rangle$ is depicted by a vertical line connecting the U-box on the q_0-wire with a white/black circle on the q_1-wire.

Postulate 4. Given a quantum system with associated Hilbert space $\mathcal{H}$, a measurement of that system is described by a collection $\{M_m\} \subset \mathcal{L}(\mathcal{H})$ of *measurement operators* satisfying the *completeness equation*

$$\sum_m M_m^\dagger M_m = \mathbb{1}. \tag{2.2.12}$$

The index m refers to the measurement outcomes that may occur in the experiment. If the system's state is $|\phi\rangle$ immediately before the measurement then the probability that result m occurs is given by

$$p(m) = \langle\phi|M_m^\dagger M_m|\phi\rangle \tag{2.2.13}$$

and the state of the system right after the measurement is

$$\frac{M_m\,|\phi\rangle}{\sqrt{\langle\phi|M_m^\dagger M_m|\phi\rangle}}. \tag{2.2.14}$$

Given a family of measurement operators $\{M_m\} \in \mathcal{L}(\mathcal{H})$ as being described in the fourth postulate, I verify now that the function p defined in (5.1.6) is indeed a valid probability distribution on the set of possible measurement outcomes. First, the basic properties of inner products and the definition of the adjoint entail that $p(m) \geq 0$ for all m. Second, the completeness equation (2.2.12) ensures that

$$\sum_m p(m) = \sum_m \langle\phi|M_m^\dagger M_m|\phi\rangle = \left\langle \phi \middle| \sum_m M_m^\dagger M_m \middle| \phi \right\rangle = \langle\phi|\mathbb{1}|\phi\rangle = \langle\phi|\phi\rangle = 1 \quad (2.2.15)$$

due to the normalization of the state $|\phi\rangle$; hence p is a probability distribution.[5]

The only measurements I will consider throughout this thesis are measurements of qubits in the computational basis. Consider first the case of a single qubit: The two possible outcomes of a measurement in the computational basis are 0 and 1 with associated measurement operators $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$, respectively. Mathematically, these operators are projections on the respective bit states $|0\rangle$ and $|1\rangle$.[6] This means that it holds that $M_{0/1} = M_{0/1}^\dagger = M_{0/1}^2$ which simplifies the calculations as I can replace each occurrence of $M_{0/1}^\dagger M_{0/1}$ with $M_{0/1}$. Now, $|0\rangle$ and $|1\rangle$ together constituting an orthonormal basis of $\mathbb{q}$ immediately implies that the completeness equation indeed holds: $M_0 + M_1 = |0\rangle\langle 0| + |1\rangle\langle 1| = \mathbb{1}$. Postulate 4 states that, given an initial state $|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, the probabilities of measuring 0 and 1 are

$$p(0) = \langle\phi|M_0|\phi\rangle = \langle\phi|0\rangle\langle 0|\phi\rangle = |\langle 0|\phi\rangle|^2 = |\alpha\,\langle 0|0\rangle + \beta\,\langle 0|1\rangle|^2 = |\alpha|^2 \text{ and} \quad (2.2.16)$$

$$p(1) = \langle\phi|M_1|\phi\rangle = \langle\phi|1\rangle\langle 1|\phi\rangle = |\langle 1|\phi\rangle|^2 = |\alpha\,\langle 1|0\rangle + \beta\,\langle 1|1\rangle|^2 = |\beta|^2, \quad (2.2.17)$$

respectively. Accordingly, the two possible post-measurement states are

$$\frac{M_0\,|\phi\rangle}{\sqrt{\langle\phi|M_0|\phi\rangle}} = \frac{|0\rangle\langle 0|\phi\rangle}{|\alpha|} = \frac{\alpha}{|\alpha|}\,|0\rangle \text{ and } \frac{M_1\,|\phi\rangle}{\sqrt{\langle\phi|M_1|\phi\rangle}} = \frac{|1\rangle\langle 1|\phi\rangle}{|\beta|} = \frac{\beta}{|\beta|}\,|1\rangle\,, \quad (2.2.18)$$

hence effectively $|0\rangle$ and $|1\rangle$, as $\alpha/|\alpha|$ and $\beta/|\beta|$ have unit magnitude and are thus merely global phases.

In a similar fashion, I can also measure multiple qubits at once which will turn out to be mathematically equivalent to consecutive single-qubit measurements. As a concrete case, consider a two-qubit system which I wish to measure in the computational

[5] A simplification of Postulate 4 can be made, if the post-measurement state is not of relevance. A family of positive operators $\{E_m\} \subset \mathcal{L}(\mathcal{H})$ is called a *positive operator-valued measure* (POVM) if it fulfills the adapted completeness equation $\sum_m E_m = \mathbb{1}$. A measurement associated with a such a POVM produces, given an initial state $|\phi\rangle$, outcome m with probability $p(m) = \langle\phi|E_m|\phi\rangle$. As one can readily check, a given collection $\{M_m\} \subset \mathcal{L}(\mathcal{H})$ of measurement operators canonically gives rise to a POVM $\{E_m\}$ via $E_m = M_m^\dagger M_m$. Describing measurements with POVMs thus allows to perform calculations more compactly. However the information about the post-measurement state is lost, as many sets of measurement operators share the same POVM.

[6] These kinds of measurements are also called *projection-valued measures* (PVMs).

basis. The possible outcomes are the four 2-bit strings 00, 01, 10, and 11. The associated measurement operators are composed of the familiar single-qubit measurement operators:

$$M_{00} = M_0 \otimes M_0 = |0\rangle\langle 0| \otimes |0\rangle\langle 0| = |00\rangle\langle 00|\,, \tag{2.2.19}$$
$$M_{01} = M_0 \otimes M_1 = |0\rangle\langle 0| \otimes |1\rangle\langle 1| = |01\rangle\langle 01|\,, \tag{2.2.20}$$
$$M_{10} = M_1 \otimes M_0 = |1\rangle\langle 1| \otimes |0\rangle\langle 0| = |10\rangle\langle 10|\,, \text{ and} \tag{2.2.21}$$
$$M_{11} = M_1 \otimes M_1 = |1\rangle\langle 1| \otimes |1\rangle\langle 1| = |11\rangle\langle 11|\,. \tag{2.2.22}$$

As for the single-qubit case, I obtain the collection of all one-dimensional projectors onto the computational basis states, implying that the completeness equation holds. Let $|\phi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle$ be the initial two-qubit state of the system. Then, according to Postulate 4 and following essentially the same calculations as in (2.2.16) and (2.2.17), I obtain

$$p(00) = |\alpha|^2,\ p(01) = |\beta|^2,\ p(10) = |\gamma|^2, \text{ and } p(11) = |\delta|^2. \tag{2.2.23}$$

Furthermore, analogously to (2.2.18), the possible post-measurement states are essentially given by the four computational basis states themselves. This readily generalizes to n-qubit systems: Measuring all n qubits in the computational basis has as possible outcomes all the bit strings of length n, i.e. $\mathbb{b}^n$. The measurement operator associated to a given bit string $\boldsymbol{x}$ is $M_{\boldsymbol{x}} = |\boldsymbol{x}\rangle\langle\boldsymbol{x}|$, the probability of obtaining $\boldsymbol{x}$ is given by the initial state's coefficient in front of the basis state $|\boldsymbol{x}\rangle$, and the post-measurement state after measuring $\boldsymbol{x}$ is given by $|\boldsymbol{x}\rangle$.

Next, I consider the case of measuring only a proper subset of all qubits. For simplicity, I assume to be given a two-qubit system of which I wish to measure the zeroth qubit in the computational basis, again yielding possible outcomes 0 and 1. The "act" of not measuring a subsystem is encoded by the identity operator $\mathbb{1}$ in the subsystem's tensor factor. Accordingly, the associated measurement operators are $M_0 = \mathbb{1} \otimes |0\rangle\langle 0|$ and $M_1 = \mathbb{1} \otimes |1\rangle\langle 1|$. These operators are again projections, simplifying the following calculations; M_0 projects onto the subspace $\mathbb{q}\otimes\mathrm{span}(\{|0\rangle\})$ and M_1 onto $\mathbb{q}\otimes\mathrm{span}(\{|1\rangle\})$. Furthermore, it holds that

$$M_0 + M_1 = \mathbb{1} \otimes |0\rangle\langle 0| + \mathbb{1} \otimes |1\rangle\langle 1| = \mathbb{1} \otimes (|0\rangle\langle 0| + |1\rangle\langle 1|) = \mathbb{1} \otimes \mathbb{1} = \mathbb{1}, \tag{2.2.24}$$

i.e. the completeness equation is again fulfilled. Consider again an arbitrary initial state $|\phi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle$. Naively performing the calculation of the measurement probabilities quickly becomes unhandy since e.g. $\langle\phi|M_0|\phi\rangle$ would expand into 16 individual terms, although most terms will vanish in the end. Instead I make the observation that applying $M_{0/1}$ to a computational basis state either eliminates the state or leaves it invariant. Especially, sandwiching $M_{0/1}$ between two orthogonal computational basis states in an inner product will result in a vanishing term, already

eliminating the 12 cross-terms. Furthermore, two out of the four remaining terms also vanish, as the projection in the zeroth factor eliminates each computational basis state that carries the "wrong" bit in its zeroth component; in the other two terms, $M_{0/1}$ does not have any effect. Therefore, I have just have argued that

$$p(0) = \langle\phi|M_0|\phi\rangle = |\alpha|^2 + |\gamma|^2 \text{ and } p(1) = \langle\phi|M_1|\phi\rangle = |\beta|^2 + |\delta|^2. \tag{2.2.25}$$

With analogous reasoning I can determine the two possible post-measurement states:

$$|\psi_0\rangle = \frac{M_0\,|\phi\rangle}{\sqrt{\langle\phi|M_0|\phi\rangle}} = \frac{\alpha\,|00\rangle + \gamma\,|10\rangle}{\sqrt{|\alpha|^2 + |\gamma|^2}} = \frac{\alpha\,|0\rangle + \gamma\,|1\rangle}{\sqrt{|\alpha|^2 + |\gamma|^2}} \otimes |0\rangle \text{ and} \tag{2.2.26}$$

$$|\psi_1\rangle = \frac{M_1\,|\phi\rangle}{\sqrt{\langle\phi|M_1|\phi\rangle}} = \frac{\beta\,|01\rangle + \delta\,|11\rangle}{\sqrt{|\beta|^2 + |\delta|^2}} = \frac{\beta\,|0\rangle + \delta\,|1\rangle}{\sqrt{|\beta|^2 + |\delta|^2}} \otimes |1\rangle\,. \tag{2.2.27}$$

A subsequent measurement of the first qubit would be modeled by the measurement operators $N_0 = |0\rangle\langle 0| \otimes \mathbb{1}$ and $N_1 = |1\rangle\langle 1| \otimes \mathbb{1}$. Provided that the previous measurement result for the zeroth qubit was 0, the probabilities of obtaining the result 0 and 1 for the first qubit are

$$p(0\,|\,0) = \langle\psi_0|N_0|\psi_0\rangle = \frac{|\alpha|^2}{|\alpha|^2 + |\gamma|^2} \text{ and } p(1\,|\,0) = \langle\psi_0|N_1|\psi_0\rangle = \frac{|\gamma|^2}{|\alpha|^2 + |\gamma|^2}, \tag{2.2.28}$$

respectively. Similarly, the probabilities of obtaining 0 and 1 for the first qubit, conditioned on the measurement of the zeroth qubit yielding 1, are

$$p(0\,|\,1) = \langle\psi_1|N_0|\psi_1\rangle = \frac{|\beta|^2}{|\beta|^2 + |\delta|^2} \text{ and } p(1\,|\,1) = \langle\psi_1|N_1|\psi_1\rangle = \frac{|\delta|^2}{|\beta|^2 + |\delta|^2}, \tag{2.2.29}$$

respectively. Therefore, the overall probability of obtaining the bit strings 00, 01, 10, and 11 match with the ones from the simultaneous measurement of both qubits:

$$p(00) = p(0\,|\,0)\,p(0) = |\alpha|^2, \qquad p(01) = p(0\,|\,1)\,p(1) = |\beta|^2, \tag{2.2.30}$$

$$p(10) = p(1\,|\,0)\,p(0) = |\gamma|^2, \quad \text{and} \quad p(11) = p(1\,|\,1)\,p(1) = |\delta|^2. \tag{2.2.31}$$

An exemplary calculation of the post-measurement state after subsequently obtaining 0 for both qubits yields

$$\frac{N_0\,|\psi_0\rangle}{\sqrt{\langle\psi_0|N_0|\psi_0\rangle}} = \frac{\alpha\sqrt{|\alpha|^2 + |\gamma|^2}\,|00\rangle}{|\alpha|\sqrt{|\alpha|^2 + |\gamma|^2}} = \frac{\alpha}{|\alpha|}\,|00\rangle\,. \tag{2.2.32}$$

Analogously, one can verify that the post-measurement states for 01, 10, and 11 also agree with the ones resulting from a simultaneous measurement of both qubits. In

summary, I have shown that simultaneous and consecutive measurement of distinct qubits are physically equivalent; they yield the same measurement statistics and post-measurement states. This holds generally true for the measurement of $1 \leq m \leq n$ qubits within an n-qubit system and is a consequence of the associated measurement operators' tensor product structure.

Quantum measurements also possess a graphical representation within quantum circuit diagrams; they are depicted as boxes with an anlog display, inputting the qubits to be measured. The output of such a measurement can further be used to classically control the application of quantum gates on other qubits. In analogy to controlled gates, this is depicted with two vertical bars, connecting the measurement box and the gate box. The classical condition that has to be fulfilled in order to apply the quantum gate is annotated next to the vertical bars. Fig. 2.3 shows an example of a two-qubit system initialized in the $|00\rangle$ state, where the zeroth qubit is measured after applying a two-qubit gate V on both qubits. Subsequently, a single-qubit gate U is executed on the first qubit if the measurement of the zeroth qubit has yielded the outcome 0. After this, also the first qubit is measured.

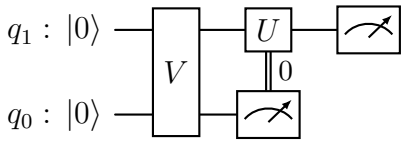


Figure 2.3: Quantum circuit diagram for a two-qubit system involving quantum measurements and classical control. After applying a two-qubit gate V to both qubits q_0 and q_1, the zeroth qubit q_0 is measured, classically controlling the application of a single-qubit gate U on q_1 on the measurement outcome 0. Subsequently, q_1 is measured, too.

Lastly, I will study the interplay of entanglement with measurements on subsystems. Consider again the general probabilities (2.2.25) and post-measurement states (2.2.26) and (2.2.27) for the one-out-of-two-qubit measurement. If the initial state is an arbitrary product state, i.e. of the form

$$(\lambda_0 |0\rangle + \lambda_1 |1\rangle) \otimes (\kappa_0 |0\rangle + \kappa_1 |1\rangle), \tag{2.2.33}$$

expanding it into the two-qubit computational basis yields that

$$\alpha = \lambda_0\kappa_0,\ \beta = \lambda_0\kappa_1,\ \gamma = \lambda_1\kappa_0,\ \text{and}\ \delta = \lambda_1\kappa_1. \tag{2.2.34}$$

These coefficients give rise to output probabilities of

$$p(0) = |\lambda_0\kappa_0|^2 + |\lambda_1\kappa_0|^2 = \Big(|\lambda_0|^2 + |\lambda_1|^2\Big)|\kappa_0|^2 = |\kappa_0|^2 \text{ and} \tag{2.2.35}$$

$$p(1) = |\lambda_0\kappa_1|^2 + |\lambda_1\kappa_1|^2 = \Big(|\lambda_0|^2 + |\lambda_1|^2\Big)|\kappa_1|^2 = |\kappa_1|^2, \tag{2.2.36}$$

where I have used that $|\lambda_0|^2 + |\lambda_1|^2 = 1$. This means, the probability of measuring either 0 or 1 on the zeroth qubit only depends on the zeroth qubit's initial state, not on the state of the first qubit. This matches the classical intuition perfectly. Furthermore, given the coefficients above, I would obtain the following post-measurement states:

$$|\psi_0\rangle = \frac{\lambda_0\kappa_0 |0\rangle + \lambda_1\kappa_0 |1\rangle}{\sqrt{|\lambda_0\kappa_0|^2 + |\lambda_1\kappa_0|^2}} \otimes |0\rangle = (\lambda_0 |0\rangle + \lambda_1 |1\rangle) \otimes \frac{\kappa_0}{|\kappa_0|} |0\rangle \text{ and} \tag{2.2.37}$$

$$|\psi_1\rangle = \frac{\lambda_0\kappa_1 |0\rangle + \lambda_1\kappa_1 |1\rangle}{\sqrt{|\lambda_0\kappa_1|^2 + |\lambda_1\kappa_1|^2}} \otimes |1\rangle = (\lambda_0 |0\rangle + \lambda_1 |1\rangle) \otimes \frac{\kappa_1}{|\kappa_1|} |1\rangle . \tag{2.2.38}$$

Intuitively, the state of the first qubit does not change due to the measurement of the zeroth qubit. In particular, both post-measurement states have the same first tensor factor.

However, intuition is about to break down when considering the entangled Bell state $|\Phi^+\rangle$ (2.2.8) instead. This time, the coefficients are given by

$$\alpha = \frac{1}{\sqrt{2}},\ \beta = 0,\ \gamma = 0,\ \text{and } \delta = \frac{1}{\sqrt{2}}. \tag{2.2.39}$$

Here, I would obtain the uniform measurement statistics of $p(0) = p(1) = 1/2$. The two possible post-measurement states, however, turn out to be $|\psi_0\rangle = |00\rangle$ and $|\psi_1\rangle = |11\rangle$. It would not make sense to claim that the measurement of the zeroth qubit affected the state of the first qubit since both individual qubit states were not well-defined in the first place. What I would observe, in turn, is that the two post-measurement states' first tensor factors do not agree. Measuring a subsystem has therefore yielded information also about the other, unmeasured subsystem. In case of the Bell state $|\Psi^+\rangle$ I would even know with certainty that a subsequent measurement of the first qubit would give the same result as the previous measurement of the zeroth qubit.

CHAPTER 3

Quantized Classical Circuits

In the previous chapter, I have detailed how classical bit strings can be represented as multi-qubit states, forming the computational basis of the quantum system's Hilbert space. In this chapter, I "quantize" classical logical gates that operate on bit strings to quantum gates, i.e. unitary operators acting on multi-qubit states.

Definition 3.1. Given a logical function $f : \mathbb{b}^n \to \mathbb{b}^n$, an operator $U \in \mathcal{L}(\mathbb{q}^{\otimes n})$ is a *quantization* of f if it is unitary and for all $\boldsymbol{x} \in \mathbb{b}^n$ it holds that $U|\boldsymbol{x}\rangle = |f(\boldsymbol{x})\rangle$. If such a unitary quantization of f exists, f is called *quantizable*.

Note that by linearity, the action of an operator $U \in \mathcal{L}(\mathbb{q}^{\otimes n})$ on each element of the computational basis uniquely determines its action on all other states $|\phi\rangle \in \mathbb{q}^{\otimes n}$. Therefore, each logical function $f : \mathbb{b}^n \to \mathbb{b}^n$ canonically gives rise to an operator $U_f \in \mathcal{L}(\mathbb{q}^{\otimes n})$ by simply defining the operator to fulfil $U_f|\boldsymbol{x}\rangle \coloneqq |f(\boldsymbol{x})\rangle$, for all $\boldsymbol{x} \in \mathbb{b}^n$. However, these operators are generally not unitary and thus no valid quantizations. In order to ensure unitarity, it is necessary and indeed sufficient that f is one-to-one, turning U_f into a permutation of computational basis states which is a unitary operation.

Corollary 3.2. A logical function $f : \mathbb{b}^n \to \mathbb{b}^n$ is quantizable if and only if it is bijective.

Therefore, the task of quantizing logical functions is in alignment with the efforts of constructing classically reversible operations, which are essential in the model of reversible computing [44]. The first section of this chapter covers the quantization of bitwise operations and borrows all major quantization strategies directly from designs made for reversible computations. In comparison, the second section addresses the quantization of arithmetic operations which is based on the quantum Fourier transform (QFT). The third section treats the quantization of boolean expressions as conditions, again building on knowledge gained from the field of reversible computing.

3.1 Bitwise Operations

The simplest (non-trivial) logical operation is the NOT gate with its truth table shown in Table 3.1. This elementary operation is reversible, hence quantizable, and its quantum version is the single-qubit `X` gate which has the matrix representation

$$\oplus \equiv \mathtt{X} \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{3.1.1}$$

Input	Output
b_0	$\neg b_0$
0	1
1	0

Table 3.1: Truth table of the NOT gate.

There are several important logical operations that act on two bits. One of them is the XOR gate which returns 1 if and only if both input bits are unequal, and therefore acts like an addition modulo 2. As a function from $\mathbb{b}^2$ to $\mathbb{b}$ it cannot be reversible. However, by additionally outputting one of the input bits beside the calculation, it is promoted to a reversible two-bit maps with truth table shown in Table 3.2. In this form, the XOR gate can be semantically described as a conditional NOT (CNOT) gate that inverts the target bit only if the control bit has the value one. Accordingly, the quantum analogue is given by the controlled `X` gate in its two possible orientations

$$\mathtt{C}_1(\mathtt{X}_0) \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad \mathtt{C}_0(\mathtt{X}_1) \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{3.1.2}$$

Another important two-bit operation is the AND gate which returns 1 if and only if both input bits are set to 1 as well. Unlike the XOR gate, the AND gate cannot be made reversible on two bits, simply because three configurations (00, 01, and 10) are mapped to 0, and there are only two 2-bit strings which have a zero at the same position. This issue can be circumvented by adding an additional input and output bit (by convention the zeroth bit) where the result is stored. However, in order to guarantee reversibility, the value of this additional bit also has to depend on its initial value, giving the correct result of the AND operation on the two target bits only if initially set to zero, and the inverted result otherwise. This gate is known as the Toffoli gate and constitutes a universal gate [45], that is, all Boolean functions can be expressed

Input		**Output**	
b_1	b_0	b_1	$b_0 \oplus b_1$
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Table 3.2: Truth table of the reversible XOR gate, storing the calculation in the zeroth bit.

as a sequence of Toffoli gates, operating on the same input bits and some additional temporary bits. The Toffoli gate's truth table is depicted in Table 3.3. Semantically, it inverts the extra bit conditioned on the other two bits being set to one. Therefore, the quantum analogue is the doubly-controlled `X` gate with its three possible orientations. The matrix representation of the lexographical order is

$$\mathtt{C}_2\mathtt{C}_1(\mathtt{X}_0) \equiv \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3.1.3}$$

Input			**Output**		
b_2	b_1	b_0	b_2	b_1	$(b_2 \wedge b_1) \oplus b_0$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Table 3.3: Truth table of the Toffoli gate, storing the result of $b_2 \wedge b_1$ in the zeroth bit if the latter is initially set to zero. If it is instead set to one, the inverted result is stored.

The last two-bit operation to be considered is the OR gate which returns 1 if and only if at least one of the input bits is set to 1. Similarly to the AND gate, one needs at

least 3 input and output bits in order to render the gate reversible (by convention, the extra bit is again the zeroth bit). However, there is no elementary gate that constitutes a reversible version of the OR gate. Instead, one constructs a two-gate sequence which consists of a CNOT gate with the first bit as control and the extra bit as target, followed by a Toffoli gate targeting the extra bit, but with inverted condition on the first bit; the corresponding truth table is shown in Table 3.4. Accordingly, the quantum version of the reversible OR gate consists of the quantized versions of the CNOT gate, followed by the quantized Toffoli gate, admitting the following matrix representation of the lexographical order:

$$\mathtt{C}_2\overline{\mathtt{C}}_1(\mathtt{X}_0)\,\mathtt{C}_1(\mathtt{X}_0) \equiv \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3.1.4}$$

Input			**Output**		
b_2	b_1	b_0	b_2	b_1	$(b_2 \vee b_1) \oplus b_0$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	1	0

Table 3.4: Truth table of the reversible OR gate, storing the result of $b_2 \vee b_1$ in the zeroth bit if the latter is initially set to zero. If it is instead set to one, the inverted result is stored.

This completes the treatment of the most relevant logical/bitwise operations and their quantizations. More complex boolean functions can be quantized by classically decomposing them into a sequence of NOT gates, AND gates, OR gates, and XOR gates with subsequent quantization of these elementary operations. With this technique also arithmetic operations are, in principle, already covered. However, I will present in the next section more elegant approaches for implementing these operations.

3.2 Arithmetic Operations

Interpreting bit strings, and hence the computational basis states, as binary representations of integers, one can also formulate arithmetic operations on multi-qubit systems. In the following, I will distinguish between encoding signed and unsigned integers into the computational basis states. Via the identification (2.2.7), I may represent the unsigned integers $0, 1, \ldots, 2^n - 1$ within an n-qubit system. For the signed version, I choose the commonly used *two's complement* representation of numbers $a \in \{-2^{n-1}, -2^{n-1}+1, \ldots, 0, \ldots, 2^{n-1}-1\}$ as n-bit strings $\boldsymbol{x}$ via

$$a = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i. \tag{3.2.1}$$

After having fixed a representation of (un)signed integers, arithmetic operations can now be implemented on a bitwise level. This is the usual approach on classical computers and the resulting gate sequences can be quantized in alignment with the methods presented in the previous section (see e.g. [46]). However, transforming arithmetic circuits such as the ripple-carry adder or the more complex Brent-Kung adder [47] into reversible ones introduces a larger overhead, both in additional (qu-)bits and gates. Instead, I will focus on implementations that are based on the quantum Fourier transform (QFT) which I describe in the following.

Definition 3.3. Given a complex vector $\boldsymbol{v} \in \mathbb{C}^d$, its *discrete Fourier transform* (DFT) is given by the vector $\boldsymbol{w} \in \mathbb{C}^d$ with entries

$$w_k = \frac{1}{\sqrt{d}} \sum_{j=0}^{d-1} v_j e^{-i2\pi jk/d}, \quad k = 0, \ldots, d-1. \tag{3.2.2}$$

The naive implementation of the DFT requires $O(d^2)$ operations. In comparison, fast Fourier transform algorithms such as the Cooley-Turkey algorithm [48] can calculate the DFT with merely $O(d \log d)$ operations. As of today, a proof that this is the optimal complexity for calculating general DFTs is still pending; optimality could only be shown under additional assumptions (see e.g. [49]). Nevertheless, it is widely believed that the scaling of $O(d \log d)$ is indeed generally optimal.

I may interpret a given complex vector $\boldsymbol{v} \in \mathbb{C}^d$ as quantum state of a quantum system to which I associate the Hilbert space $\mathbb{C}^d$, provided that the vector is normalized. In this context, one can define the quantum version of the DFT.

Definition 3.4. The *quantum Fourier transformation* (QFT) is the linear operator $\mathtt{QFT} : \mathbb{C}^d \to \mathbb{C}^d$ defined by its action on the orthonormal basis states

$$\mathtt{QFT}\,|j\rangle \coloneqq \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} e^{2\pi i j k/d}\,|k\rangle\,, \quad k = 0, \ldots, d-1.$$

Note that the DFT and QFT differ in the sign of the exponents. Therefore, the QFT is technically a quantum version of the inverse DFT. Crucially, as stated in Proposition 3.5, the QFT is a unitary operator and therefore a valid quantum circuit.

Proposition 3.5. The QFT is a unitary operator.

Proof. The case $d = 1$ is trivial, as in this case $\mathtt{QFT} = \mathbb{1}$. Therefore let $d > 1$. It suffices to prove the assertion for ONB states. Since the adjoint is given by

$$\mathtt{QFT}^\dagger\,|k\rangle = \frac{1}{\sqrt{d}} \sum_{\ell=0}^{d-1} e^{-2\pi i k \ell/d}\,|\ell\rangle\,,$$

it holds that

$$\begin{aligned}
\mathtt{QFT}^\dagger \mathtt{QFT}\,|j\rangle &= \frac{1}{\sqrt{d}} \sum_{k=0}^{d-1} e^{2\pi i j k/d} \mathtt{QFT}^\dagger\,|k\rangle \\
&= \frac{1}{d} \sum_{k=0}^{d-1} e^{2\pi i j k/d} \sum_{\ell=0}^{d-1} e^{-2\pi i k \ell/d}\,|\ell\rangle \\
&= |j\rangle + \frac{1}{d} \sum_{\substack{\ell=0 \\ \ell \neq j}}^{d-1} \sum_{k=0}^{d-1} \Big(e^{2\pi i (j-\ell)/d}\Big)^k\,|\ell\rangle \\
&= |j\rangle + \frac{1}{d} \sum_{\substack{\ell=0 \\ \ell \neq j}}^{d-1} \omega \sum_{k=0}^{\tilde{d}-1} \Big(e^{2\pi i \phi/\tilde{d}}\Big)^k\,|\ell\rangle = |j\rangle\,,
\end{aligned}$$

where

$$(\omega,\, \phi,\, \tilde{d}) = \begin{cases} \Big(|j-\ell|,\, 1,\, d/(j-\ell)\Big), & \text{if } (j-\ell) \mid d \\ \Big(1,\, j-\ell,\, d\Big), & \text{otherwise.} \end{cases}$$

In either case, all terms vanish except for $|j\rangle$ since the sum of all n-th complex unit roots is zero, for $n > 1$. The first case is clear. In the second case, $(j-\ell) \nmid N$, thus $e^{2\pi i(j-\ell)/d}$ is a primitive unit root and the summation over k also gives the sum over all N-th complex unit roots. □

Assuming that $d = 2^n$ for some $n \in \mathbb{N}$, one especially obtains the `QFT` for an n-qubit system, first discovered by Coppersmith [50]. The multi-qubit `QFT` can be constructed from $O(n^2)$ many (controlled) elementary gates of the form

$$\mathtt{H} \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad \mathtt{R}_k \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^k} \end{bmatrix}, \quad k \in \mathbb{N}, \tag{3.2.3}$$

where `H` is the famous Hadamard gate and $\mathtt{R}_k$ applies a relative phase to the portion of a qubit that is in the state $|1\rangle$. A concrete quantum circuit implementing the n-qubit `QFT` as envisioned by Coppersmith is depicted in Fig. 3.1. I have omitted the final layer of SWAP gates which reverses the qubit order. In applications, this is usually not implemented anyway since it is more efficient to simply keep track of the reversed qubit order classically and to reorder the qubit routing of subsequent quantum circuits. This will be also the case for all operations discussed in the following. Lastly, note that the classical version, assuming optimality of the current fast Fourier transform algorithms, would need at least $O(n2^n)$ classical operations. That is, the `QFT` offers an exponential speed-up over its classical counterpart.[1]

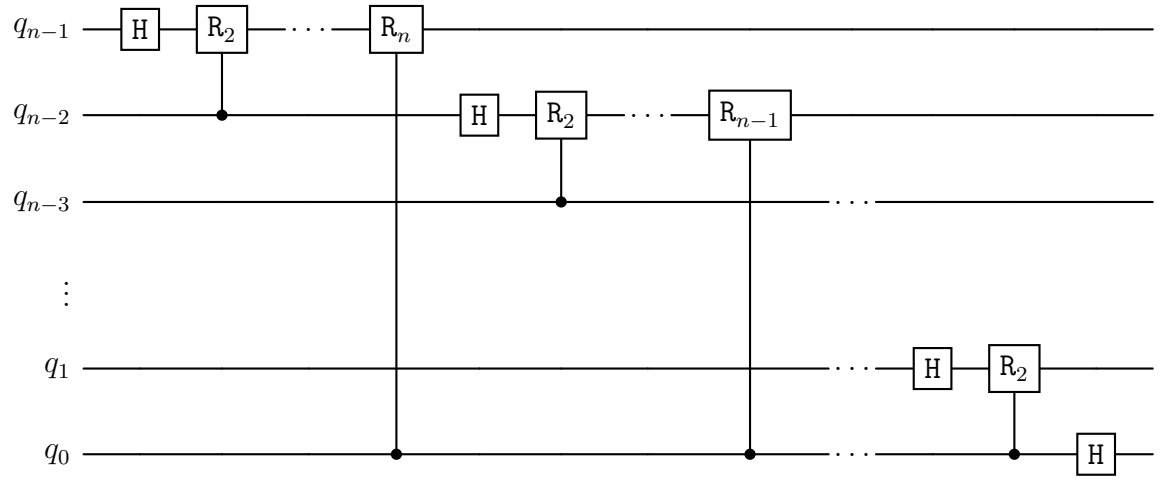


Figure 3.1: Quantum circuit diagram of quantum Fourier transform of an n-qubit register $q_0, \ldots, q_{n-1}$. On the qubit q_i, an initial Hadamard gate is followed by a cascade of $i + 1$ singly-controlled rotational gates. The control runs over the qubits $q_{i-1}, \ldots, q_0$ and the applied phase is halved from one gate to the next.

[1] Exponential speed-ups governed by pure quantum subroutines such as the `QFT` always have to be taken with a grain of salt. Even though the `QFT` might compute the DFT exponentially faster than any classical algorithm could, the result is still encoded into a quantum state. As soon as I would like to read out the DFT via quantum measurements, I would need to sample/measure the quantum state exponentially often in order to resolve the exponentially small differences between the relative phases of the computational basis states.

QFT-based arithmetic operations all follow the same design principle which was first formulated by Draper [51] for the concrete case of integer addition: acting on the qubit register of one operand with the QFT, applying rotational gates which are controlled on the qubits of the other operand, subsequently applying the inverse QFT to the first register. The key observation is that the QFT-ed state encodes its bit information in the relative phases such that simple rotational gates, which act additively on the phases, can efficiently produce the QFT-ed state of the sum of the former state and any other desired state such that subsequently acting with the inverse QFT yields the correct state. More concretely, applying the QFT on an n-qubit CB state $|\boldsymbol{x}\rangle$ in fact produces the state

$$\mathtt{QFT}\,|\boldsymbol{x}\rangle = \frac{1}{\sqrt{2^n}}\Big(|0\rangle + e^{2\pi i 0.x_{n-1}\cdots x_1 x_0}\,|1\rangle\Big)\cdots\Big(|0\rangle + e^{2\pi i 0.x_1 x_0}\,|1\rangle\Big)\Big(|0\rangle + e^{2\pi i 0.x_0}\,|1\rangle\Big), \tag{3.2.4}$$

where $0.x_k\cdots x_1x_0$ is the binary fraction which corresponds to dividing the integer represented by $\boldsymbol{x}$ by 2^k, respectively. Now assume I wish to add a single bit $y \in \mathbb{b}$ to a given integer $\boldsymbol{x}$. In order to prepare the QFT-ed state of $|\boldsymbol{x}+y\rangle$ I would have to add, for each $k = 0,\ldots,n-1$, the binary fraction consisting of k zeros and y as the trailing binary place to the $(k+1)$-th exponent in (3.2.4). This can be achieved by acting with a y-controlled $\mathtt{R}_k$ on the $(k+1)$-th qubit, respectively. If a bit string $\boldsymbol{y} = y_1y_0$ of length two should be added, the circuit for the addition of y_0 is simply appended with a cascade of y_1-controlled $\mathtt{R}_\ell$ acting on the ℓ-th qubit, respectively. Each additional bit/qubit introduces $n-k$ singly-controlled rotational gates on top. This pattern readily generalizes for arbitrary long bit strings and is depicted in Fig. 3.2 for the phase addition of two n-qubit integers. Applying this quantum circuit to the previously QFT-ed qubit register thus yields the state

$$\begin{aligned}\frac{1}{\sqrt{2^n}}\Big(\,|0\rangle + e^{2\pi i(0.x_{n-1}\cdots x_1x_0+0.y_{n-1}\cdots y_1y_0)}\,|1\rangle\,\Big)\otimes\cdots\otimes\\ \otimes\,\Big(\,|0\rangle + e^{2\pi i(0.x_1x_0+0.y_1y_0)}\,|1\rangle\,\Big)\otimes\Big(\,|0\rangle + e^{2\pi i(0.x_0+0.y_0)}\,|1\rangle\,\Big)\end{aligned} \tag{3.2.5}$$

in the first register which readily is the QFT-ed version of the CB state $|\boldsymbol{x}+\boldsymbol{y}\rangle$. Therefore, subsequently applying the inverse QFT indeed yields the latter state. As for all the quantized circuits for logical/bitwise operations, it suffices to verify the correctness of this operation on the CB states. By linearity, I may employ the very same circuit to add together two superpositions of quantum integers. Furthermore, note that, due to the properties of the two's complement, the (quantum) circuit looks the same for both unsigned and signed integers. By exchanging the quantum controls in Fig. 3.2 with classical ones, I may use the same circuit idea to add a classical integer to a quantum integer. Note that this circuit construction also already covers subtraction; flipping the signs of all the relative phases incurred from the rotational gates clearly performs a phase subtraction rather than a phase addition.

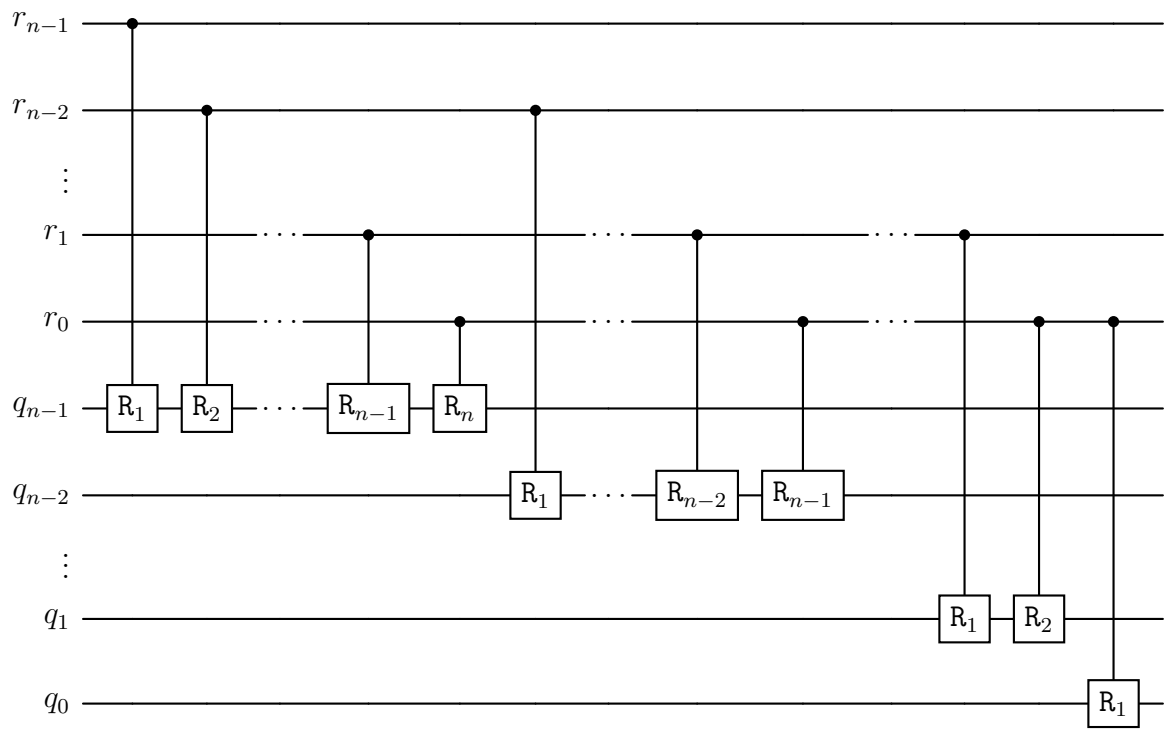


Figure 3.2: Quantum circuit diagram for phase addition. The qubits $q_0, \ldots, q_{n-1}$ belong to the previously QFT-ed integer register while the qubits $r_0, \ldots, r_{n-1}$ represent the second quantum integer which is to be added to the first one. On the qubit q_i, a cascade of $i+1$ controlled rotational gates is applied. Within each of these cascades the control runs over the last $i+1$ qubits in the r-register and the applied phase is halved from one gate to the next. The result is the QFT-ed state of the sum of both quantum integers.

The ingenious construction by Draper has spawned QFT-based circuits also for the other arithmetic operation. However, for multiplication, division, and modulo the fact that one of the operands may be zero complicates the circuit design. Integer multiplication simply is not reversible and can therefore, unlike integer addition and subtraction, not admit a straight-forward quantization. This issue can be resolved by introducing an additional register where the result is stored such that the initial operands remain unchanged in their respective registers. Then, even in the case where one of the operands is zero, different operands get mapped to different output (considering all three registers) and reversibility has been achieved. A concrete implementation for

integer multiplication is described in [52] and is centered around the decomposition

$$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{k,\ell=0}^{n-1} x_k y_\ell 2^{k+\ell} = \sum_{k=0}^{n-1} 2^k (x_k \boldsymbol{y} + y_k \boldsymbol{x} - x_k y_k) \tag{3.2.6}$$

of the product to be computed. The corresponding phase multiplication consists of the phase addition/subtraction of all $3n$ terms in the most right-hand side of (3.2.6). The respective power of two enters directly into the rotational gates' applied relative phases while the bit-valued pre-factor can be realized by additionally controlling all associated rotational gates on the respective qubit. These $3n$ operations can be further condensed to $2n$ phase additions by simply leaving out the addition of $2^k x_k y_k$ when adding either $2^k x_k \boldsymbol{y}$ or $2^k y_k \boldsymbol{x}$ and therefore accounting for the negative term. Furthermore, I may ignore all rotational gates which would carry a relative phase larger than πi as this directly corresponds to overflows. Sandwiching the just described circuit between the `QFT` and inverse `QFT` on the third register implements the unitary which acts on products of CB states as $|\boldsymbol{x}\rangle |\boldsymbol{y}\rangle |\boldsymbol{z}\rangle \mapsto |\boldsymbol{x}\rangle |\boldsymbol{y}\rangle |\boldsymbol{z} + \boldsymbol{x}\boldsymbol{y}\rangle$. Initializing the third register with zero therefore writes the integer multiplication of $\boldsymbol{x}$ and $\boldsymbol{y}$ in the third register.

Lastly, I consider integer division and the modulo operation. Both operations can again not be made reversible with only two outputs, since none of them is initially defined when the second operand is zero and choosing any artificially set value will eventually collide with the result of the regular division and modulo operation. Therefore, one extra output register is necessarily required for reversibility. When calculating the quotient and remainder simultaneously, an entirety of four registers is necessary. Regarding the case where the second operand is zero, the usual choice is to simply return the original dividend as quotient and some large value for the remainder. There are several works on quantum versions of integer division and modulo operation; in the following I will focus on the quantized restoring division algorithm as proposed in [53]. The classical version works on four n-bit registers: The first one holds the dividend, the second one holds the divisor, the third register is initialized to zero and will hold the resulting quotient, and the fourth register is initialized with the dividend and will hold the remainder. The following steps are repeated n times: Bit-shift the combined remainder-dividend register to the left by one, subtract the divisor from the remainder register and if the result is negative restore the former value by adding the divisor to the remainder register; otherwise set the least significant bit of the quotient register to one. This completes the algorithm. Note that, if the divisor is zero then the quotient will be equal to the dividend and the remainder will be set to $011\ldots1$, i.e. the largest positive number representable in the two's complement. If dividend and divisor differ in their signs, the final quotient has to be adjusted to hold a negative value. When working with unsigned inputs, the subtractions and subsequent checks for negativity have to be substituted with direct integer comparisons and conditional subsequent subtraction to avoid underflows in the unsigned format. Basically all parts of the

classical restoring division algorithm are directly quantizable with the available tools. Namely, I have already introduced quantum circuits for addition and subtraction, and checking whether the result of a subtraction is negative can be implemented by a quantum control on the leading qubit in the respective register. A corresponding quantum circuit for in-place division on inputs in the two's complement (without sign adjustment) is shown in Fig. 3.3. For unsigned input, adding one qubit to each register allows for applying the regular restoring division algorithm on $(n+1)$-qubit integers.

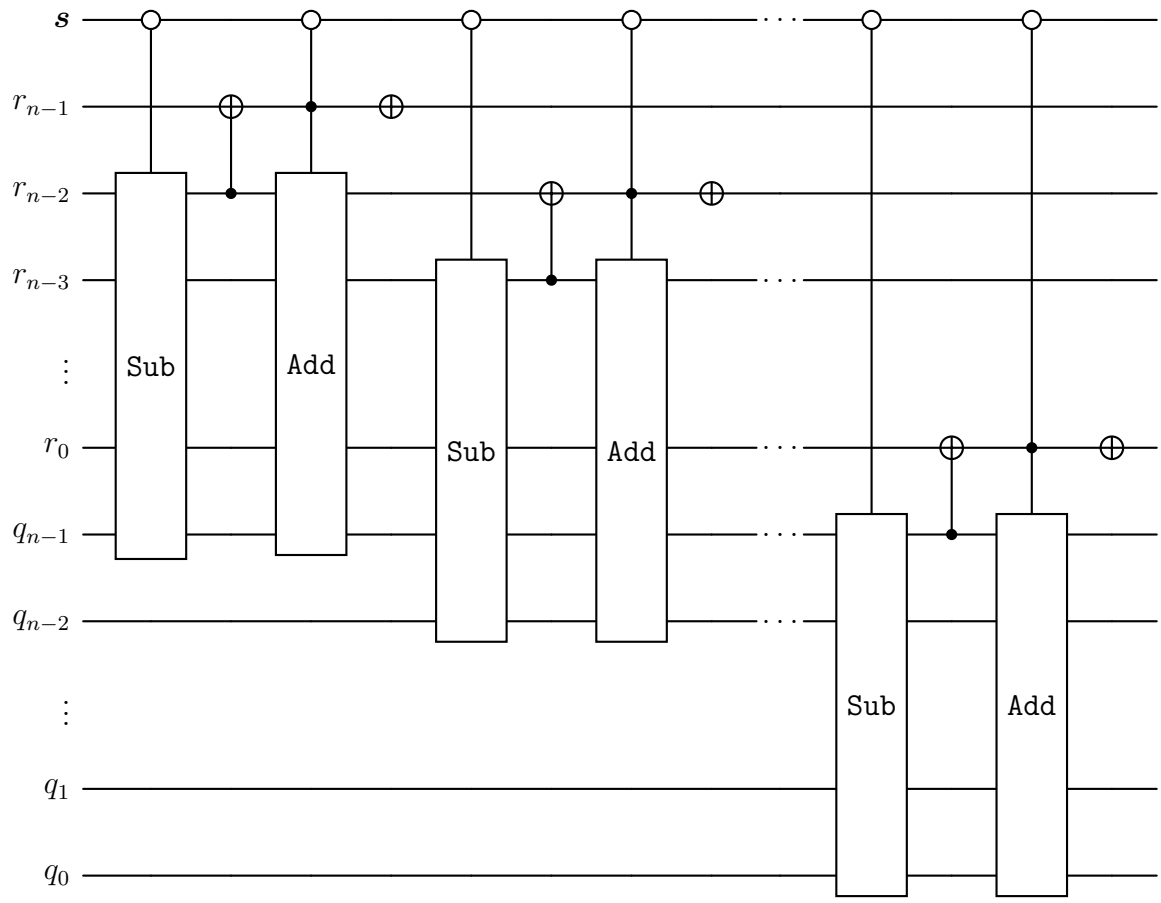


Figure 3.3: Quantum circuit diagram for division and modulo operation. The qubit register $\boldsymbol{s}$ holds the divisor's value and is used for in-place addition and subtraction on the compound register $q_0, \ldots, q_{n-1}, r_0, \ldots, r_{n-1}$ where the first n qubits represent the dividend and the last n qubits are initialized to zero. Between the cascade of alternating subtractions and additions with the divisor which are shifted towards the beginning of the compound register by one qubit per step, CNOT and `X` gates account for correctly setting the remainder register's qubits.

3.3 Boolean Expressions as Conditions

Now that I have detailed how to quantize various bitwise and arithmetic operations, I wish to discuss how to quantize conditional operations. The underlying idea is to take the quantization of the respective unconditional operation and to control it on an ancilla flag qubit which was previously flipped to the $|1\rangle$ state whenever the respective condition is fulfilled. This should happen coherently, i.e. without measurements so that the flag qubit also has to be manipulated with suitable controlled quantum gates. In the following I may always assume that the operation acts (at least partly) on the same registers on which also the condition is checked, enforcing the introduction of flag qubits to store the result of checking the condition. Note, however, that if the controlled operation and the condition affect disjoint registers, the operation may occasionally be controlled directly on the qubits of the registers on which the condition is to be evaluated, avoiding the need for additional qubits.

A scheme for quantizing general logical formulae can be given inductively: The simplest case arises when an operation is conditioned directly on a single (qu-)bit being set to one. Here, the flag qubit, being initialized in the $|0\rangle$ state, is conditionally flipped using the CNOT gate Eq. (3.1.2) with the conditioning qubit as control qubit. Given a formula φ and a quantum circuit realizing the unitary

$$\mathtt{C}_\varphi(\mathtt{X}_\mathrm{a}) \coloneqq \sum_{\varphi(\boldsymbol{x})=0} |\boldsymbol{x}\rangle\langle\boldsymbol{x}| \otimes \mathbb{1}_\mathrm{a} + \sum_{\varphi(\boldsymbol{x})=1} |\boldsymbol{x}\rangle\langle\boldsymbol{x}| \otimes \mathtt{X}_\mathrm{a} \tag{3.3.1}$$

which flips the ancilla qubit for all CB states which fulfill the formula φ, the quantum circuit implementing the control on the negated formula $\neg\varphi$ is simply given by flipping the ancilla qubit unconditionally before applying the control on the unnegated formula, that is

$$\mathtt{C}_{\neg\varphi}(\mathtt{X}_\mathrm{a}) = \mathtt{C}_\varphi(\mathtt{X}_\mathrm{a})(\mathbb{1} \otimes \mathtt{X}_\mathrm{a}). \tag{3.3.2}$$

Controlling on the exclusive OR of two formulae φ and ψ is again relatively straight forward. Assuming access to the unitaries $\mathtt{C}_\varphi(\mathtt{X}_\mathrm{a})$ and $\mathtt{C}_\psi(\mathtt{X}_\mathrm{a})$, the control on $\varphi \oplus \psi$ simply arises from executing both unitaries in sequence since the ancilla qubit is flipped twice (and therefore brought back to the $|0\rangle$ state) whenever both formulae are fulfilled:

$$\mathtt{C}_{\varphi\oplus\psi}(\mathtt{X}_\mathrm{a}) = \mathtt{C}_\psi(\mathtt{X}_\mathrm{a})\mathtt{C}_\varphi(\mathtt{X}_\mathrm{a}). \tag{3.3.3}$$

For controlling on the logical OR/AND of two formulae φ and ψ, I have to introduce two ancilla qubits a_0, a_1. In both cases, one of the formulae, say φ, is checked and the result stored on $\mathrm{a}_0/\mathrm{a}_1$. Secondly, the remaining formula is checked for. In case of the logical OR, a_1 is only flipped if additionally a_0 has not been flipped previously

such that a_1 is only set to $|1\rangle$ if ψ is fulfilled, but φ is not. Afterwards, a_0 is flipped controlled on a_1. If φ is fulfilled, a_0 already is in the $|1\rangle$ state and the final flip is not applied as a_1 has not been flipped in the second step. If φ is not fulfilled, but ψ is, this final CNOT gate flips a_0 into the $|1\rangle$ state. Thus, in summary, the described gate sequence implements the check for $\varphi \vee \psi$ and stores the result on a_0, that is

$$\mathtt{C}_{\varphi\vee\psi} = \Big(\mathbb{1} \otimes \mathtt{C}_{\mathrm{a}_1}(\mathtt{X}_{\mathrm{a}_0})\Big)\mathtt{C}_{\psi}\overline{\mathtt{C}}_{\mathrm{a}_0}(\mathtt{X}_{\mathrm{a}_1})\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}_0}). \tag{3.3.4}$$

In case of the logical AND, flipping a_0 is controlled on both the qubit a_1 and on whether ψ holds true, ensuring that a_0 is only set to $|1\rangle$ if both φ and ψ are fulfilled, yielding the gate sequence

$$\mathtt{C}_{\varphi\wedge\psi} = \mathtt{C}_{\psi}\mathtt{C}_{\mathrm{a}_1}(\mathtt{X}_{\mathrm{a}_0})\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}_1}) \tag{3.3.5}$$

This concludes the treatment of the most fundamental logical operations. More complex logical formulae can always be decomposed into some combination of variables, negations, exclusive ORs, logical ORs, and logical ANDs e.g. in the common conjunctive or disjunctive normal form (see [54]). A summary of all quantized control structures is also given in Table 3.5.

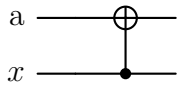
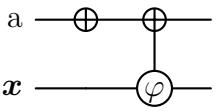
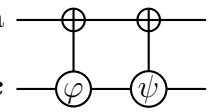
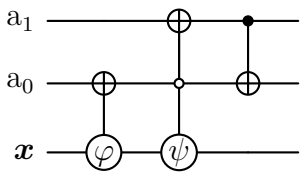
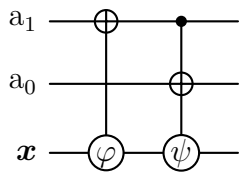
formula	unitary	quantum circuit
x	$\mathtt{C}_x(\mathtt{X}_{\mathrm{a}})$	
$\neg\varphi$	$\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}})(\mathbb{1} \otimes \mathtt{X}_{\mathrm{a}})$	
$\varphi \oplus \psi$	$\mathtt{C}_{\psi}(\mathtt{X}_{\mathrm{a}})\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}})$	
$\varphi \vee \psi$	$\Big(\mathbb{1} \otimes \mathtt{C}_{\mathrm{a}_1}(\mathtt{X}_{\mathrm{a}_0})\Big)\mathtt{C}_{\psi}\overline{\mathtt{C}}_{\mathrm{a}_0}(\mathtt{X}_{\mathrm{a}_1})\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}_0})$	
$\varphi \wedge \psi$	$\mathtt{C}_{\psi}\mathtt{C}_{\mathrm{a}_1}(\mathtt{X}_{\mathrm{a}_0})\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}_1})$	

Table 3.5: Quantum control flow for logical formulae. This set of elementary quantization schemes inductively defines a valid approach for arbitrary logical formulae.

In addition to formulae involving only logical, i.e. boolean, variables, I wish to also quantize comparisons of two integers, say $\boldsymbol{x}$ and $\boldsymbol{y}$. A uniform approach, already used in the (quantum) division algorithm (see Fig. 3.3), is to subtract $\boldsymbol{y}$ from $\boldsymbol{x}$ via the subtraction method described in Fig. 3.2 and to subsequently check whether...

$\boldsymbol{x}$ `==` $\boldsymbol{y}$: ...all qubits are in the $|0\rangle$ state.

$\boldsymbol{x}$ `!=` $\boldsymbol{y}$: ...at least one qubit is in the $|1\rangle$ state.

$\boldsymbol{x}$ `<` $\boldsymbol{y}$: ...the most significant qubit is in the $|1\rangle$ state (negative number).

$\boldsymbol{x}$ `>=` $\boldsymbol{y}$: ...the most significant qubit is in the $|0\rangle$ state (non-negative number).

For checking $\boldsymbol{x}$ `>` $\boldsymbol{y}$, one may subtract $\boldsymbol{y}+1$ instead and then perform the same check as for $\boldsymbol{x}$ `>=` $\boldsymbol{y}$. Analogously, checking $\boldsymbol{x}$ `<=` $\boldsymbol{y}$ can be implemented by subtracting $\boldsymbol{y}-1$ and then performing the same check as for $\boldsymbol{x}$ `<` $\boldsymbol{y}$. For unsigned n-qubit integers, I may again introduce a single ancilla qubit in order to perform the subtraction on a signed $(n+1)$-qubit integer. The entire check may be computed in-place in which case it has to be uncomputed via a subsequent addition by $\boldsymbol{y}$ (or $\boldsymbol{y} \pm 1$ for strict inequalities). Alternatively, one may introduce an ancilla register, initialized in $|\mathbf{0}\rangle$ and copy the relevant information via a layer of CNOT gates.

Lastly, I wish to detail how to quantize computational branches, appearing e.g. in `if-else`-constructions. A single `if`-statement is already covered by the discussions above; All operations in the `if`-body have to be controlled on the flag qubit previously manipulated with a quantum control flow $\mathtt{C}_{\varphi}(\mathtt{X}_{\mathrm{a}})$ for the `if`-condition φ. When appending an `else`-statement, all operations within the `else`-branch have to be explicitly controlled on the negated flag qubit in order to ensure that only one of the two computational branches is executed on a given CB state. Similarly, when appending one or multiple `else if`-statements, the operations within an `else if`-branch have to be controlled not only on the respective `else if`-condition, but also on all previously checked conditions to be not fulfilled. This requires to store the evaluation of each condition on a separate flag qubit and to control the i-th `else if`-condition on the i-th qubit being in the $|1\rangle$ state and the zeroth to $(i-1)$-th flag qubits being in the $|0\rangle$ state. The final `else`-branch, if present, has to be controlled on all flag qubits being in the $|0\rangle$ state.

CHAPTER 4

The Design of CQ

In this chapter, I detail the construction of CQ. The first section gives a collection of CQ's vocabulary, i.e. of all valid "words"; these words are already partially equipped with some semantical meaning to motivate their selection. Furthermore, I comment on the implementation of the lexer, that is, the software which inputs a stream of characters (a potential CQ-source code file) and tokenizes the input into valid words. In the second section, I introduce the entire grammar of CQ while keeping semantical interpretation to a minimum. However, in order to give insights into the language's inner workings, some comments on the intended semantics are necessary. For each set of grammar rules, I display the corresponding source code for the parser, that is, the software that takes the tokenized output of the lexer and verifies that it complies to the given grammar rules. Lastly, in the third section, I complete the discussion of CQ's semantic. The previously grammar rules are extended to action rules in order to construct the abstract syntax tree (AST). I design a one-pass compiler that can check syntactical and semantical correctness simultaneously. This means, the AST does not have to be traversed again in order to check a program's semantical correctness. The excellent online tutorial by drifter1 [55] about compiler construction helped me enormously to get myself familiar with the relevant software tools. All the code shown in this chapter can be found under [32].

4.1 Vocabulary and Lexer

As in most other programming languages, identifier for variable or function names may start with a letter of the Latin alphabet (lower- or uppercase) or an underscore and be followed up by an arbitrary amount of Latin letters, underscores, and digits. Special combinations are reserved for the language's 21 keywords which are summarized in Table 4.1. For example, the tokens `false` and `true` are reserved for boolean truth-values, i.e. are boolean literals. Integer literals are sequences purely comprised of digits, optionally preceded by a minus-sign. These three rules are summarized in Fig. 4.1

```
1 ID                    [a-zA-Z_][a-zA-Z0-9_]*
2 BCONST                "false"|"true"
3 ICONST                "-"?[0-9]+
```

Figure 4.1: Lexer code snippet for defining allowed identifiers as well as boolean and integer literals.

The remaining 19 keywords are tokenized similarly to `false/true`. For example, the two qualifier keywords `const` and `quantum` are translated to internal objects `CONST` and `QUANTUM`, respectively (see Fig. 4.2).

```
1 const                 { return CONST; }
2 quantum               { return QUANTUM; }
```

Figure 4.2: Lexer code snippet for recognizing the two qualifier keywords `const` and `quantum`.

In addition to identifiers, integer literals, and keywords, there are symbols and sequences of symbols reserved for operations and delimiters. The four logical operations incorporated into CQ are NOT (`!`), AND (`&&`), OR (`||`), and XOR (`^^`). Further logical elements are the comparisons (`<`, `<=`, `>`, and `>=`) as well as equality (`==`) and inequality (`!=`). The nine possible integer operations are comprised of four bitwise operations and five arithmetic operations: bitwise-INVERT (`~`), bitwise-AND (`&`), bitwise-OR (`|`), and bitwise-XOR (`^`), as well as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`). All these operations, except the unary INVERT, also come as in-place assignment operators (`|=`, `^=`, `&=`, `+=`, `-=`, `*=`, `/=`, and `%=`), complementing the ordinary assignment operator (`=`). Lastly, CQ uses as delimiters colons (`:`), commas (`,`), semicolons (`;`), parentheses (`(` and `)`), square brackets (`[` and `]`), and braces (`{`, `}`).

The code snippets in Figs. 4.1 and 4.2 are taken from my implementation of the lexer using the C-based flex-software. In this framework, the prescribed vocabulary is easily accessible from the lexer source code; language elements are simply listed together with some actions (expressed as C-code) which should be executed upon reading a given pattern from the input file. If characters in the input file do not match any prescribed pattern, the lexer throws an error, indicating invalid CQ-source code. The lexer could also be run independently from the subsequent parser, simply performing an analysis of the vocabulary.

CQ-**Keyword**	**Purpose**
`bool`	primitive type
`break`	classical control flow
`case`	classical and quantum switch-statements
`continue`	classical control flow
`const`	type qualifier, for classical variables only
`default`	classical and quantum switch-statements
`do`	classical control flow
`else`	classical and quantum control flow
`false`	boolean truth-value
`for`	classical control flow
`if`	classical and quantum control flow
`int`	primitive type
`measure`	quantum measurement
`phase`	address phase of quantum variable
`quantum`	type qualifier for quantum variables
`return`	classical and quantum control flow
`switch`	classical and quantum control flow
`true`	boolean truth-value
`unsigned`	primitive type
`void`	no-return functions
`while`	classical control flow

Table 4.1: CQ-Keywords. 18 out of the 21 keywords of CQ are directly imported from C, while `measure`, `phase`, and `quantum` are completely new keywords which are introduced for purely quantum syntax. CQ features the two qualifiers `const` and `quantum`, and the four types `bool`, `int`, and `unsigned`, and `void`. The keywords `break`, `continue`, `do`, `for`, and `while` are for classical control flow only, while `case`, `default`, `else`, `if`, `return`, and `switch` can be used in the quantum context as well.

4.2 Grammar and Parser

Conducting the lexical analysis of the input with the lexer introduced in the previous section ensures a stream of valid tokens. The next step is to define the grammar of CQ which has these tokens as terminal symbols. The grammar should be deterministic context-free in order to admit a sufficiently efficient parser [56]. For the implementation of the parser, I use GNU's bison [31] that accepts production rules in Backus-Naur form and constructs the corresponding parser from it. I will display all following grammar rules already as bison source code.

By convention, the starting symbol of the grammar is `program`. In the global scope I only wish to allow for variable declaration and definition as well as function definitions; other statements may only appear inside the body of a function. This is implemented by producing a declaration list from `program` which, in turn, can yield arbitrarily many declarations via left-recursion; it always contains at least one declaration, thus prohibiting an empty program. Finally, a declaration in this sense can either be a variable declaration, a variable definition, or a function definition. These three product rules complete the construction of the global scope and are summarized in Fig. 4.3.

```
1 program: decl_l ;
2
3 decl_l: decl
4     | decl_l decl ;
5
6 decl: var_decl
7     | var_def
8     | func_def ;
```

Figure 4.3: Parser code snippet for the global scope. The declaration list (`decl_l`) is derived from the starting symbol `program`. The former, in turn, can be expanded into at least one, but otherwise arbitrarily many declarations (`decl`) which may either be a variable declaration (`var_decl`), a variable definition (`var_def`), or a function definition (`func_def`). All appearing semicolons are not part of the production rules themselves. They are used by bison to mark the respective production rule's end.

A variable declaration always consists of a type specifier, determining the primitive type and array dimensions, the variable's name and a subsequent semicolon. This sequence may optionally preceded by the `quantum`-qualifier while the `const`-qualifier is not permitted. In contrast, a variable definition may start with any qualifier (none, `const`, or `quantum`), followed by the type specifier, the variable's name, the assignment symbol, an initializer, and a closing semicolon. The initializer may either be a single expression or an initializer list enclosed by braces. Such an initializer list is then expanded into at least one expression; arbitrarily many additional elements together with a comma may be appended via left-recursion. Notably, in the context of initialization, also superposition-calls to functions are allowed. The syntax for such a call is to enclose the function's name with square brackets and to omit the argument, as it is determined by the variable to be defined on the left-hand side. The type specifier which is present both in variable declarations as well as in variable definitions is eventually expanded into one of the three primitive types `bool`, `int`, and `unsigned`. Arbitrarily many array dimensions, consisting of a square brackets-enclosed expression, may be appended to the primitive type via left-recursion. The rules for variable declaration and definition as well as for the initializer and the type specifier are summarized in Fig. 4.4.

```
var_decl: QUANTUM type_specifier declarator SEMICOLON
    | type_specifier declarator SEMICOLON ;

var_def: QUANTUM type_specifier declarator ASSIGN init SEMICOLON
    | CONST type_specifier declarator ASSIGN init SEMICOLON
    | type_specifier declarator ASSIGN init SEMICOLON ;

init:
    lor_expr
    | LBRACKET ID RBRACKET
    | LBRACE init_elem_l RBRACE ;

init_elem_l: lor_expr
    | LBRACKET ID RBRACKET
    | init_elem_l COMMA LBRACKET ID RBRACKET
    | init_elem_l COMMA lor_expr ;

type_specifier: BOOL
    | INT
    | UNSIGNED
    | type_specifier LBRACKET or_expr RBRACKET ;

declarator: ID ;
```

Figure 4.4: Parser code snippet for variable declaration and variable definition. Declared variables may be declared with the `quantum`-qualifier or with no qualifier; they cannot be `const`. Defined variables, in turn, can also be `const`. Variable definitions include an assignment symbol (`=`) and a right-hand side (`init`) which is either a single expression or an initializer list (`init_elem_l`) of arbitrary length. Variables are uniquely determined by their qualifier, their type (`type_specifier`), and their name (`ID` in `declarator`). Both, variable declaration and variable definition are closed with a trailing semicolon.

Function definitions are comprised of a description of their return type, the function's name, a parentheses-enclosed comma-separated list of parameters (type specifiers and names) as well as the actual function body. A function's return may have none or the `quantum`-qualifier, and any primitive type as well as the `void`-type, where `void` is a substitute for the combination of qualifier and ordinary type specifier, i.e. neither the `quantum`-qualifier, nor array dimensions are allowed whenever the function definition involves `void`. The function parameters may have none or the `quantum`-qualifier, and any primitive type as well as arbitrarily array dimensions, implemented via the ordinary type specifier. The parameter list itself can hold arbitrarily many elements via left-recursion, thus it may also be empty. Lastly, the braces-enclosed function body gives rise to a sub-program. All productions related to function definitions are summarized in Fig. 4.5.

```
1  func_def: QUANTUM type_specifier declarator func_head func_tail
2      | type_specifier declarator func_head func_tail
3      | VOID declarator func_head func_tail ;
4
5  func_head: LPAREN par_l RPAREN
6      | LPAREN RPAREN ;
7
8  par_l: par
9      | par_l COMMA par ;
10
11 par: QUANTUM type_specifier declarator
12     | type_specifier declarator ;
13
14 func_tail: LBRACE sub_program RBRACE ;
```

Figure 4.5: Parser code snippet for function definition. These start with specifying the return-type of the function. A function may not return anything (`void`) or return either classical (unqualified) or quantum values which can be arrays of arbitrary dimensions. The function head (`func_head`) holds a (possibly empty) parameter list (`par_l`) with unqualified or quantum parameters (`par`), which can also be array-valued. Eventually, the function tail (`func_tail`) is given by a braces-enclosed sub-program (`sub_program`).

Similar to the global `program`, a sub-program may be expanded into a statement list with at least one statement. This rule is again implemented via left-recursion. Variable declarations and definitions are also valid within sub-programs while function definitions are not. In addition, statements may be one of the restricted statements: phase-shift, measurement, function call, if-else-construction, switch-construction, do-while-loop, while-loop, for-loop, break, continue, return, and assignment. A restricted sub-program, in turn, may only contain restricted statements and therefore neither declarations nor definitions. In the following, I will allow for general sub-programs as bodies of functions and loops, while if-else-branches and switch-cases are merely restricted sub-programs. The product rules for (restricted) sub-programs and statements are also depicted in Fig. 4.6.

The left-hand side of a phase-shift-statement is given by the `phase`-keyword, followed by a parentheses-enclosed reference (a variable's name, optionally with array indexing). This is followed up by either an addition-assignment or subtraction-assignment. The right-hand side consists of an expression as well as a trailing semicolon. In contrast, the measurement-statement has only one possible derivation. It consists of the `measure`-keyword, a parentheses-enclosed reference, and a trailing semicolon. The two possible expansions for the phase-shift-statement and the production rule for the measurement-statement are depicted in Fig. 4.7.

```
sub_program: stmt_l ;

stmt_l: stmt
    | stmt_l stmt ;

stmt: decl_stmt
    | res_stmt ;

decl_stmt: var_decl
    | var_def ;

res_sub_program: res_stmt_l ;

res_stmt_l: res_stmt
    | res_stmt_l res_stmt ;

res_stmt: phase_stmt
    | measure_stmt
    | func_call_stmt
    | if_stmt
    | switch_stmt
    | do_stmt
    | while_stmt
    | for_stmt
    | break_stmt
    | continue_stmt
    | return_stmt
    | assign_stmt;
```

Figure 4.6: Parser code snippet for a sub-program. A sub-program is expanded into a statement list (`stmt_l`) which contains at least one statement (`stmt`), but otherwise arbitrarily many statements. A statement, in turn, may either be a declaration statement (`decl_stmt`) or a restricted statement (`res_stmt`) which is either a phase-shift (`phase_stmt`), a measurement (`measure_stmt`), a function call (`func_call_-stmt`), an if-statement (`if_stmt`), a switch-statement (`switch_stmt`), a do-while-loop (`do_stmt`), a while-loop (`while_stmt`), a for-loop (`for_loop`), break (`break_-stmt`), continue (`continue_stmt`), a return-statement (`return_stmt`), or an assignment (`assign_stmt`). Declaration statements are either variable declarations or definitions, but no function definitions. A restricted sub-program (`res_sub_program`) is expanded into a list of restricted statements (`res_stmt_l`).

Function calls can either be direct or inverted in which case the actual function call is preceded by the INVERT-operator. The atomic function call is comprised of either the function's name or by the function's name enclosed in square brackets, and a (possibly empty) parentheses-enclosed argument list. The arguments themselves can be general expressions. These three production rules are also listed in Fig. 4.8.

```
phase_stmt: PHASE LPAREN ref_expr RPAREN ASSIGN_ADD lor_expr
    SEMICOLON
     | PHASE LPAREN ref_expr RPAREN ASSIGN_SUB lor_expr SEMICOLON ;

measure_stmt: MEASURE LPAREN ref_expr RPAREN SEMICOLON ;
```

Figure 4.7: Parser code snippet for a phase-shift- and a measurement-statement. The phase-shift-statement (`phase_stmt`) is expanded into the `phase`-keyword, a subsequent reference (`ref_expr`) enclosed by parentheses, either a addition- or subtraction-assignment, and an expression as right-hand side. The measurement-statement (`measure_stmt`) is given by the `measure`-keyword followed by an expression enclosed by parentheses.

```
func_call_stmt: INV func_call SEMICOLON
     | func_call SEMICOLON ;

func_call: ID LPAREN arg_expr_l RPAREN
     | ID LPAREN RPAREN
     | LBRACKET ID RBRACKET LPAREN arg_expr_l RPAREN ;

arg_expr_l: lor_expr
     | arg_expr_l COMMA lor_expr ;
```

Figure 4.8: Parser code snippet for a function call statement. Function call statements (`func_call_stmt`) consist of a function call (`func_call`) and a trailing semicolon, where the function call may optionally be inverted via a leading INVERT-operator. The actual function call is given by a function's name, possibly enclosed by square brackets and a list of arguments (`arg_expr_l`). If the function call is regular (without square brackets), the argument list may also be empty. In any case, there is not upper bound on the number of arguments, and the individual arguments may be separated with commas.

If-else-statements are initialized by the `if`-keyword, followed by a parentheses-enclosed expression: the if-condition. The if-branch, in turn, is given by a braces-enclosed restricted sub-program. In addition, the if-branch may be complemented by an arbitrary amount of else-if-branches, implemented via left-recursion. These are introduced with a combination of the `else`-keyword and the `if`-keyword, again followed by a parentheses-enclosed expression, the corresponding else-if-condition. As for the if-branch, the else-if-branches are restricted sub-programs, enclosed by braces. Eventually, an optional trailing else-branch may complete the if-else-statement, consisting only of the `else`-keyword and a braces-enclosed restricted sub-program, the else-branch. The entire construction of if-else-statements is also summarized in Fig. 4.9.

```
1 if_stmt: IF LPAREN lor_expr RPAREN LBRACE res_sub_program RBRACE
      optional_else
2     | IF LPAREN lor_expr RPAREN LBRACE res_sub_program RBRACE
      else_if optional_else ;
3
4 else_if: ELSE IF LPAREN lor_expr RPAREN LBRACE res_sub_program
      RBRACE
5     | else_if ELSE IF LPAREN lor_expr RPAREN LBRACE
      res_sub_program RBRACE ;
6
7 optional_else: ELSE LBRACE res_sub_program RBRACE
8     | /* empty */ ;
```

Figure 4.9: Parser code snippet for an if-else-statement. An if-statement (`if_stmt`) starts with the `if`-keyword holds an parentheses-enclosed expression and a braces-enclosed restricted sub-program followed by an arbitrary amount of else-ifs (`else_if`) and an optional else (`optional_else`). Else-ifs consist of the combination of the `else`- and `if`-keyword, an expression and a restricted sub-program; A simple `else` has to be followed by a restricted sub-program.

Switch-statements begin with the `switch`-keyword, followed by a parentheses-enclosed expression, the switch-expression, and a list of cases, enclosed by braces. The case list must at least hold one case, but, using left-recursion, there is no upper limit on the number of provided cases. A single case may either start with the `case`-keyword and a subsequent constant, or with the `default`-keyword. Both variants are continued by a colon and a restricted sub-program. Note that, unlike for the if-else-construction, no enclosing braces are required. The grammar rules for the switch-statement and cases are also shown in Fig. 4.10.

```
1 switch_stmt: SWITCH LPAREN lor_expr RPAREN LBRACE case_stmt_l
      RBRACE ;
2
3 case_stmt_l: case_stmt
4     | case_stmt_l case_stmt ;
5
6 case_stmt: CASE const_val COLON res_sub_program
7     | DEFAULT COLON res_sub_program ;
```

Figure 4.10: Parser code snippet for a switch-statement. Such a switch-statement is introduced with the `switch`-keyword, followed up by a parentheses-enclosed expression and a braces-enclosed list of cases (`case_stmt_l`). A case list, in turn, can hold arbitrarily many case-statements (`case_stmt`). Lastly, a case-statement consists of a colon and a restricted sub-program, preceded by either the `case`-keyword followed by a constant (`const_val`) or the `default`-keyword.

The three types of loops (do-while-loop, while-loop, and for-loop) are introduced with their respective keyword. In case of the do-while-loop, the initial `do`-keyword is followed up by a braces-enclosed sub-program, the `while`-keyword, a parentheses-enclosed expression and a trailing semicolon. The order of expression and sub-program is reversed for while-loops: The initial `while`-keyword is followed by a parentheses-enclosed expression and then by the braces-enclosed sub-program. Finally, for-loops start with the `for`-keyword. Their parentheses-enclosed head is comprised of a variable definition or an assignment, an expression, and a subsequent assignment. The for-loop's body is, as for the other two loops, given by a braces-enclosed sub-program. The production rules for all three loops are again summarized in Fig. 4.11.

```
1 do_stmt: DO LBRACE sub_program RBRACE WHILE LPAREN lor_expr RPAREN
      SEMICOLON ;
2
3 while_stmt: WHILE LPAREN lor_expr RPAREN LBRACE sub_program RBRACE
      ;
4
5 for_stmt: FOR LPAREN for_first lor_expr SEMICOLON assign_expr
     RPAREN LBRACE sub_program RBRACE ;
6
7 for_first: var_def
8     | assign_stmt ;
```

Figure 4.11: Parser code snippet for loop statements. Do-while-loops (`do_stmt`) and while-loops (`while_stmt`) have similar syntax: Both consist of a sub-program enclosed by braces and the `while`-keyword followed by a parentheses-enclosed expression. Merely the order of these elements and the fact that the do-while loop additionally start with the `do`-keyword and ends with a semicolon are different. In contrast, the for-loop (`for_stmt`) is initiated with the `for`-keyword followed by a parentheses-enclosed triple of either an assignment or a variable definition (`for_first`), an expression, and an assignment expression (`assign_expr`). As for the other loops, the body of the for-loop is a sub-program enclosed by braces.

The three possible jump-statements in CQ are break-statements, continue-statements, and return-statements. The first two merely consist of the respective keyword (`break` and `continue`) and a semicolon while the latter may optionally enclose an expression between the leading `return`-keyword and the trailing semicolon. These production rules are also listed in Fig. 4.12.

There are nine possible assignment-statements in CQ. The different types of assignment-statements exactly correspond to the different types of assignment-operators that I already introduced in the previous section. In any case, the statement consists of a reference as left-hand side, the respective assignment-operator, an expression as right-hand side, and a trailing semicolon. These rules are again listed in Fig. 4.13.

```
break_stmt: BREAK SEMICOLON ;

continue_stmt: CONTINUE SEMICOLON ;

return_stmt: RETURN SEMICOLON
    | RETURN lor_expr SEMICOLON ;
```

Figure 4.12: Parser code snippet for jump-statements. Both the break-statement (`break_stmt`) and the continue-statement (`continue_stmt`) merely consist of the respective keyword and a semicolon. A return-statement (`return_stmt`) may, in addition to the `return`-keyword and a closing semicolon, contain an expression.

```
assign_stmt: assign_expr SEMICOLON ;

assign_expr: ref_expr ASSIGN lor_expr
    | ref_expr ASSIGN_OR lor_expr
    | ref_expr ASSIGN_XOR lor_expr
    | ref_expr ASSIGN_AND lor_expr
    | ref_expr ASSIGN_ADD lor_expr
    | ref_expr ASSIGN_SUB lor_expr
    | ref_expr ASSIGN_MUL lor_expr
    | ref_expr ASSIGN_DIV lor_expr
    | ref_expr ASSIGN_MOD lor_expr ;
```

Figure 4.13: Parser code snippet for an assignment-statement. The assignment-statement (`assign_stmt`) consists of an assignment-expression (`assign_expr`) and a trailing semicolon. The assignment-expression, in turn, is comprised of a reference as left-hand side, one of the nine assignment operators, and an expression for right-hand side.

CQ's grammar has several production rules for dealing with expressions. The following production rules are heavily inspired by [57] which provides exemplary bison code for some of C's syntax rules. First and foremost, a fixed ordering of different expressions accounts for the different precedences of logical, bitwise, and arithmetic operators. Whenever expressions occurred in previous production rules, they were logical OR expressions; correspondingly, the logical OR has the lowest precedence among all operators. These may either be expanded into a single logical XOR expression or a logical XOR expression preceded by another logical OR expression and the corresponding operator. Similarly, a logical XOR expression may yield a logical AND expression, optionally preceded by a logical XOR expression and the operator. The logical AND operator has the highest precedence among all binary logical operators; a logical AND expression eventually gives rise to a comparison expression, preceded by an arbitrary amount of logical AND expressions and logical AND operators. The three production rules for binary logical operators are also depicted in Fig. 4.14.

```
lor_expr: lxor_expr
    | lor_expr LOR lxor_expr ;

lxor_expr: land_expr
    | lxor_expr LXOR land_expr ;

land_expr: comparison_expr
    | land_expr LAND comparison_expr ;
```

Figure 4.14: Parser code snippet for logical expressions. The order of the logical expression resembles the precedence of the three binary logical operators, that is, the logical OR has the lowest precedence, followed by the logical XOR while the logical AND has the highest precedence among these three operators. A logical AND expression (`land_expr`) may eventually give rise to a comparison expression (`comparison_expr`).

The four comparison operators have equal precedence, stronger than the one of any binary logical operator, but weaker than the one of the equality and inequality operator. Accordingly, a comparison expression may directly yield an (in-)equality expression or another comparison expression followed by one of the comparison operators and an (in-)equality expression. Similarly, the equality and inequality operator have the same precedence which, in turn, is below the precedence of every binary integer operator. An (in-)equality expression therefore may be expanded into an OR expression, preceded by arbitrarily many combinations of (in-)equality expressions and either the equality or inequality operator. These rules are also shown in Fig. 4.15.

```
comparison_expr: equality_expr
    | comparison_expr GE equality_expr
    | comparison_expr GEQ equality_expr
    | comparison_expr LE equality_expr
    | comparison_expr LEQ equality_expr ;

equality_expr: or_expr
    | equality_expr EQ or_expr
    | equality_expr NEQ or_expr ;
```

Figure 4.15: Parser code snippet for comparison and (in-)equality expressions. Unlike the binary logical operators, the four comparison operators all share the same precedence, implemented by alternative productions rather than consecutive ones. A comparison expression may eventually yield an (in-)equality expression (`equality_expr`). Again, the equality and inequality operator share the same precedence which is superseded by that of integer operators. Therefore, an (in-)equality expression may eventually give rise to the integer operator expression of lowest precedence: the bitwise OR expression.

The eight different binary integer operators demand several production rules for properly enforcing correct precedence. They are ordered in precedence according to the following scheme: Bitwise operators have lower precedence than arithmetic operators. Internally, the bitwise operators' precedences are ordered in the same way as the ones of their logical counterparts, that is bitwise OR has the lowest, bitwise XOR the second lowest, and bitwise AND the highest precedence. Addition and subtraction have the next higher precedence. Finally, multiplication, division and modulo share the highest precedence among all binary operators. The corresponding production rules for integer expressions as shown in Fig. 4.16 follow the given order of precedence, allowing for different expansions whenever two or more operators tie in terms of their precedence.

```
1  or_expr: xor_expr
2      | or_expr OR xor_expr ;
3
4  xor_expr: and_expr
5      | xor_expr XOR and_expr ;
6
7  and_expr: add_expr
8      | and_expr AND add_expr ;
9
10 add_expr: mul_expr
11     | add_expr ADD mul_expr
12     | add_expr SUB mul_expr ;
13
14 mul_expr: unary_expr
15     | mul_expr MUL unary_expr
16     | mul_expr DIV unary_expr
17     | mul_expr MOD unary_expr ;
```

Figure 4.16: Parser code snippet for integer expressions. The order of logical expressions mirrors the precedence of the eight binary integer operators: first the three binary bitwise operators OR, XOR, AND, followed by addition and subtraction and completed by multiplication, division, and modulo. A multiplicative expression (`mul_expr`) eventually gives rise to a unary expression (`unary_expr`).

The last set of production rules is concerned with unary operators, measurements, function calls, references with array indexing, and constants. First, a unary expression may either yield a postfix expression, another unary expression preceded by the logical NOT-operator or the INVERT-operator, or a measurement of another parentheses-enclosed unary expression. A postfix expression, in turn, may either be a primary expression, a function call, or a reference. References are identifier of variables followed by an arbitrary amount of square brackets-enclosed OR expressions, representing the C-style of array indexing. Lastly, a primary expression may either be a constant or a logical OR expression, enclosed by parentheses, thus establishing prioritization of operations via parentheses. These production rules are again shown in Fig. 4.17.

```
unary_expr: postfix_expr
    | NOT unary_expr
    | INV unary_expr
    | MEASURE LPAREN unary_expr RPAREN ;

postfix_expr: primary_expr
    | func_call
    | ref_expr ;

ref_expr: ref ;

ref: ID
    | ref LBRACKET or_expr RBRACKET ;

primary_expr: const
    | LPAREN lor_expr RPAREN ;

const: BCONST
    | ICONST ;
```

Figure 4.17: Parser code snippet for unary expressions. These include negated or inverted expressions as well as measured quantities. Of higher precedence than these unary operators/keywords are function calls, references with C-style array indexing, and parentheses-enclosed expressions.

4.3 Abstract Syntax Tree and Semantic Analysis

Combining all previously given code snippets would give a perfectly valid parser which only checks for syntactical correctness. However, these grammar rules do not prohibit code like `int a = true;` While this example is a syntactically valid variable definition, it is semantical nonsense to assign the truth value `true` to an integer variable. These and many other cases have to be excluded with additional rules which are either applied alongside the production rules or subsequently. For the CQ, I chose to implement the first option as it is more performant than the second while still being easily maintainable. The general strategy is to enrich the production rules with additional semantical checks, expressed as C-code. In order to avoid the definition of an integer with a truth value, the parser additionally compares the declared type (in this case `int`) with the type of the supplied right-hand side (in this case `true` which has the type `bool`) and raises an error whenever these do not match. In addition, the parser creates a tree structure (the AST) out of the different parts of the source code. For the variable definition, it creates a corresponding definition-node with the right-hand side of the definition as child node. The constant-node for the truth value `true` has no further child nodes such that the tree for the exemplary variable definition is completed.

Referring to Fig. 4.3, the root node of the entire AST corresponds to the whole `program`. It can have arbitrarily many child nodes which correspond to global variable declarations, variable definitions, and function definitions. The nodes for variable declarations, in turn, do not have any child nodes. When constructing them, the parser checks whether the variable has already been declared or defined previously; this step is implemented with a global symbol table that stores variable and function names, their appearances, and additional information such as their type. The same verification is performed when encountering variable definitions. Additionally, these nodes carry all elements of the right-hand side as child nodes (compare Fig. 4.4). For each child node, the parser performs a type check: matching qualifier (assigning quantum to classical is prohibited), matching primitive types (only `bool` to `bool`, `int` to `int`, `unsigned` to `unsigned`, and `int` to `unsigned` are allowed), and matching array dimensions (comparing depth and size of each dimension). If an initializer list is used, its elements have to be arrays of depth zero, and their total number must not exceed the product of all dimensions' sizes of the declared variable. For `const`-declared variables, each element on the right-hand side also has to be constant.

Function definitions (compare Fig. 4.5) have, by far, the most complicated semantical restrictions. First, in analogy to variable declarations/definitions, the availability of the function's name is checked via the global symbol table. Upon entering the function's parameter list, the scope is increased and all parameters are declared within this narrower scope, allowing for potential name collisions with previously defined variables or functions. The parameters' type information are directly stored within the function definition-node while the body of the function enters as a child node. This child node, in turn, holds arbitrarily many child nodes which correspond to all statements within the `sub_program` defined by the function body. Such a statement list-node carries additional metadata which has to be compared to the function's declared properties. Namely, each function must eventually return with the type specified in its declaration (the type check is the same as being performed for the variable definition). In order to monitor this, a statement list itself carries information about its return type which is simply compared to the return type specifier and qualifier of the function. If no return statement is present, this is interpreted as returning `void` which raises an error if the function is declared with a non-void return. Furthermore, a subprogram can include loops, `if-else`-statements, and `switch`-statements which may not return in every case. The recursive rules for determining whether a statement list has a conditional or unconditional return are the following: A loop never returns unconditionally, an `if-else`-statement only return unconditionally if an `else`-branch exist and all branches return unconditionally, a `switch`-statement only returns unconditionally if the `default`-case exist and all case branches return unconditionally, and finally, a statement list returns unconditionally if and only if all its statements return unconditionally. As soon as multiple return statements appear within, e.g., a `switch`-statement, their return types are compared upon creating the respective statement, raising an error if they

do not match. Therefore, a statement list always carries a unique return type which then can be compared to the function's return type. In addition to its return type and its parameters, a function may have one of the two properties: being quantizable or being unitary. A function is quantizable if it returns classically or `void`, only inputs classical parameters, and has a quantizable function body. In comparison, a function is unitary if it has a quantum or no return and has a unitary function body. Whether the function body is quantizable or unitary is, analogously to its return type, determined recursively: A statement list is quantizable/unitary if and only if every statement is quantizable/unitary. Variable declarations, variable definitions, loops, and measurements are neither quantizable nor unitary. Phase shifts are never quantizable and only unitary if their right-hand side is unitary. Assignments of classical/quantum variables are never unitary/quantizable; they are quantizable/unitary if and only if the assignment operator is not `=` and the right-hand side is quantizable/unitary. Function calls are quantizable/unitary if the respective function as well as all parameter expressions are quantizable/unitary. A function call also counts as unitary if the function is quantizable and is supplied with quantum parameters which are themselves unitarily computed. Array indexing is quantizable and unitary if and only if all indices are constant. Logical and binary operations are quantizable/unitary of all operands are quantizable/unitary. Return-statements in the top scope of the function body are both quantizable and unitary. If-else-statements are quantizable/unitary if and only if all expressions and branches are quantizable/unitary, and switch-statements are quantizable/unitary if and only if the switch-expression and all case-branches are quantizable/unitary. The rules for determining whether an if-/if-else-/else-/case-branch is quantizable/unitary are the same as for the function body with the exception that return-statements destroy both properties.

Sub-programs and restricted sub-programs are represented as statement list-nodes with arbitrarily many child nodes just like the global `program`. The child nodes correspond to the possible statements within the list (compare Fig. 4.6). When creating a node for a phase-shift-statement (see Fig. 4.7), it has to be checked whether the left-hand side actually refers to a quantum variable while the right-hand side has to be classical; both sides enter as child nodes. Similarly, when creating a node for a measurement-statement, the referenced variable has to be quantum; it is appended as child node.

Function calls (compare Fig. 4.8) require again more semantical checks. When creating a corresponding node in the AST the function's name has to be looked up in the symbol table to infer its properties such as return type, (number of) parameters, and whether it is quantizable or unitary. The provided parameter expressions enter as child nodes of the function call-node, comparing the number of child nodes to the declared number of parameters and raising an error if both numbers do not match. Furthermore, for each parameter a type check is performed following the rules detailed for the variable definition. However, there are some special treatments involving quantizable functions.

If the function in question is declared to input only classical parameters, but quantum expressions are provided, it is checked whether the function is quantizable. If this is the case, all provided parameter expressions must be quantum; otherwise, or if the function is not quantizable, an error is raised. If the function's name is enclosed with square brackets, the parameters necessarily have to be quantum, the function has to be quantizable, and has to return a single classical bool. If the function call is preceded by an INVERT-operator, the function either has to be unitary or must have been invoked with quantum parameters while being quantizable and, additionally, must have no return.

If-else-expressions can be either classical or quantum, but must always be single bools. As soon as one expression in an if-else-statement is quantum, all others have to be quantum too; for the mixed case, an error is raised. Furthermore, in the quantum case, all branches have to be unitary restricted sub-programs. For the classical case, there is no such restriction. An if-else-node contains the if-expression, the if-branch, and the optional else-branch as child nodes (compare Fig. 4.9). Additionally, each else-if-block enters as an additional child node which, in turn, carries the corresponding expression and branch as child nodes.

Switch-statements (compare Fig. 4.10) may incorporate a classical or quantum expression which, however, must always be a single value; arrays are not allowed. Each case-constant has to pass a type check, comparing its primitive type to the expression's type. If the switch-expression is quantum, all case-branches, including the optional default-branch, have to be unitary restricted sub-programs. None of the case-constants or the `default`-keyword may be repeated. The switch-node has the expression as well as each individual case (including `default`) as child nodes. A case-node, in turn, carries the respective case-branch as child node.

Since loops of any kind (compare Fig. 4.11) are purely classical control structures in CQ, there semantics come without additional enhancements: Do-loop-nodes as well as while-nodes have the sub-program contained in the loop-body and the loop-condition as child nodes; only the order is reversed between both types of loops. While there are no restrictions on the sub-program, the expression constituting the loop-condition has to be a single, classical bool. A for-loop-node additionally possesses a variable definition or assignment as well as a (second) assignment as child nodes. Both structures may be of any qualifier. The restriction on the loop-condition is the same as for the other two loops, that is a single, classical bool is required. Furthermore, break-statements and continue-statements (compare Fig. 4.12) may only appear inside a loop's body. A return-statement is allowed everywhere except for the global scope. While break-node and continue-node do not have any child nodes, the return-node has the return-expression, if existing, as child node.

Assignment-nodes also have two child nodes corresponding to the left-hand side and the right-hand side of the assignment (compare Fig. 4.13). Two additional semantical rules are executed upon constructing the nodes: First, the primitive type of the left-hand side has to be compatible with the respective assignment-operator. Since the assignment-operators for the logical and bitwise operation are comprised of the same symbols the only incompatibilities arise when the left-hand side's primitive type is bool while the assignment-operator is arithmetic. Second, left-hand side and right-hand side have to pass the same type check as incorporated in variable definitions.

Since CQ overloads all binary operators to allow for classical, quantum, or mixed operands, all binary expressions (compare Figs. 4.14 to 4.16) comply to the similar semantical checks: Both operands' types have to be compatible with the respective operator and further pass the already established type check. Logical operators only accept operands of type bool, (in-)equality operators have no restriction on the operands (comparing, e.g., a bool to an integer is, nevertheless, prohibited by the subsequent type check), and comparison operators, bitwise as well as arithmetic operators only accept (unsigned) integer operands. For unary operators (compare Fig. 4.17), only compatibility of operator and operand has to be established; the logical NOT-operator may only precede an expression of primitive type bool while the INVERT-operator may only precede expressions which are (arrays of) signed or unsigned integers. Lastly, array indices may only be classical expressions and never be quantum.

This completes the description of CQ's semantics. The parser provided along this thesis already includes the whole semantical analysis. I omitted the source code here due to its sheer length; all relevant semantical rules have been elaborated on anyway. The current version of the parser constructs the whole AST as described within this section and allows for saving it in a text file after compilation.

CHAPTER 5

Grover's Algorithm in CQ

My main motivation when designing CQ was to give a compact and natural formulation of Grover's famous search algorithm. In this chapter, I explain first the quantum algorithm and its scope on a conceptual level. This follows the mathematical language introduced in Section 2.2. Subsequently, I iterate through the algorithmic blueprint and discuss how the different steps of the algorithm are matched with CQ-functionalities, resulting in the CQ-source code for Grover as presented in Fig. 5.1. Finally, I compare CQ's version of Grover's algorithm with implementations found in established quantum programming languages.

5.1 Grover's Algorithm

Before explaining how Grover's algorithm works, let me first clarify for which kind of problems it has been designed initially. Suppose to be given a collection of N items of which $M \leq N$ items are marked. The task is to find one of the M marked items among the totality of N items. In order to verify that an item is marked, I may assume that there is a boolean *oracle* function $f : \{1, \ldots, N\} \to \mathbb{b}$, returning one for (the index of) a marked item and zero otherwise. Without any further structure or knowledge about relations between the items, the simplest approach to solving this problem classically is also optimal: Linear search iterates through the collection of items, evaluates the oracle function for each item, and stops as soon as the function evaluates to one, returning its input. This algorithm clearly has a best-case performance of $O(1)$, an average performance of $O(N/M)$, and a worst-case performance of $O(N - M)$.

Assume for simplicity that $N = 2^n$ for some $n \in \mathbb{N}$ such that the problem can be mapped to n (qu-)bits and assign a CB state $|\boldsymbol{x}\rangle$ to each item, e.g. by encoding the item's index in some list as a bit string. On a high level, Grover's algorithm may be summarized as follows: Create a uniform superposition of all CB states, then alternately apply a quantum oracle, which flips the phase of all states corresponding

to a solution, and a *diffusion* operator which applies a phase flip in all direction orthogonal to the uniform superposition. Due to the irrelevance of global phases, this is physically equivalent to applying a phase flip only in direction of the uniform superposition. Finally, measure the outcome state to obtain a candidate for a solution. By convention, qubits are initialized in the $|0\rangle$ state. In order to create the initial uniform superposition, it suffices to apply a layer of Hadamard gates (3.2.3) to each qubit. In addition, the quantum oracle is typically implemented with the aid of an ancilla qubit in a state whose phase can be flipped easily. One common approach is to initialize the ancilla qubit in the $|1\rangle$ state and to apply controlled `Z` gates whenever the phase has to be flipped conditionally. Therefore, the initialization step of Grover's algorithm reads

$$\left(\mathtt{H}^{\otimes n} \otimes \mathtt{X}_{\mathrm{a}}\right)(|\mathbf{0}\rangle \otimes |0\rangle_{\mathrm{a}}) = |+\rangle \otimes |1\rangle_{\mathrm{a}} \coloneqq \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} |k\rangle \otimes |1\rangle_{\mathrm{a}}\,, \tag{5.1.1}$$

where the existence of the ancilla register is optional and depends on whether the quantum oracle can be implemented in-place or not. With the choice made for the ancilla qubit's initial state, the general quantum oracle consists of applying a `Z` gate to the ancilla qubit whenever the main register's state fulfills the oracle. How this control is implemented highly depends on the classical oracle function. In Chapter 3, I have detailed how to quantize elementary building blocks of classical functions. Following these strategies, every oracle function can, at least in principle, be quantized to give rise to a valid quantum oracle. As an example, consider the oracle that returns true for every odd number and false otherwise. A possible classical implementation is given by `x` $\mapsto$ `(x & 1) == 1`, i.e. it is checked whether the bitwise AND of the input and the number one (whose binary representation has $n-1$ leading zeros and its least significant bit set to one) yields one. It is readily sufficient to simply check whether the least significant qubit of the input is set to one (odd number) or to zero (even number) and to control a phase flip on the former. As highlighted previously, the `Z` gate only introduces a phase flip when applied to the $|1\rangle$ state. Therefore it is sufficient to apply a `Z` gate directly to the least significant qubit without the need for an ancilla qubit or any additional control structure. This ancilla-free construction is, however, rarely possible. Lastly, the diffusion operator is usually implemented by sandwiching a check for the $|\mathbf{0}\rangle$ state between two layers of Hadamard gates. Since a layer of Hadamard gates (also called Hadamard transform) maps $|\mathbf{0}\rangle$ to $|+\rangle$ and vice versa, this readily gives the correct operation. This operation can again be implemented without requiring any ancilla qubit: Instead of checking for the $|\mathbf{0}\rangle$ state, sandwich the check for the all-one state between two layers of `X` gates which readily translate between both states. This check can now be implemented in-place by applying a multi-controlled `Z` gate to any of the main registers qubits, controlled on every other qubit in the main register. Only if all qubits are in the $|1\rangle$ state, the `Z` gate is executed and introduces a phase flip. Otherwise, the execution may not be triggered or the gate itself does not have any effect on the $|0\rangle$ state.

In order to verify the correctness of Grover's algorithm and to infer its runtime, let me first rewrite the uniform superposition as

$$\begin{aligned} |+\rangle &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} |k\rangle = \cos\left(\tfrac{\theta}{2}\right) |\alpha\rangle + \sin\left(\tfrac{\theta}{2}\right) |\beta\rangle \text{ with} \\ \theta &\coloneqq 2 \arccos\left(\sqrt{\tfrac{N-M}{N}}\right), \\ |\alpha\rangle &\coloneqq \frac{1}{\sqrt{N-M}} \sum_{f(\boldsymbol{x})=0} |\boldsymbol{x}\rangle \,, \text{ and} \\ |\beta\rangle &\coloneqq \frac{1}{\sqrt{M}} \sum_{f(\boldsymbol{x})=1} |\boldsymbol{x}\rangle \,. \end{aligned} \tag{5.1.2}$$

The definition of θ may appear arbitrary but will play a fundamental role in the following derivation. More intuitively, the state $|\alpha\rangle$ is the uniform superposition of all CB states that do not correspond to solutions (i.e. on which the oracle evaluates to false) while the state $|\beta\rangle$ is the uniform superposition of all solution states (i.e. on which the oracle evaluates to true). Accordingly, note that the normalization factors in $|\alpha\rangle$ and $|\beta\rangle$ precisely match (the inverse square root of) the number of infeasible states $(N - M)$ and feasible states (M), respectively. Correspondingly, the quantum oracle may generally be written as

$$U_f = \mathbb{1} \otimes |0\rangle\langle 0|_{\mathrm{a}} + \Big(\mathbb{1} - 2\,|\beta\rangle\langle\beta|\Big) \otimes |1\rangle\langle 1|_{\mathrm{a}} \,, \tag{5.1.3}$$

although only the second term is of relevance since the ancilla qubit stays within the state $|1\rangle$ throughout the entire routine. For the concrete example, illustrated previously, the ancilla qubit may indeed be dropped entirely. The diffusion operator with its naturally ancilla-free implementation possesses the abstract form

$$U_{\overline{\mathbf{0}}} = \Big(2\,|+\rangle\langle +| - \mathbb{1}\Big) \otimes \mathbb{1}_{\mathrm{a}} \tag{5.1.4}$$

where again only the second term is relevant for understanding the algorithm's functionality. I may therefore omit the ancilla qubit from all the following calculations. The following lemma and its elementary proof establish the fundamental design principle which fuels Grover's algorithm and various other related quantum algorithms.

Lemma 5.1. Applying the Grover iteration $U_{\overline{\mathbf{0}}} U_f$ k-times, $k \in \mathbb{N}$, to the initial state (5.1.2) yields

$$\Big(U_{\overline{\mathbf{0}}} U_f\Big)^k |+\rangle = \cos\left(\Big(\tfrac{1}{2} + k\Big)\theta\right) |\alpha\rangle + \sin\left(\Big(\tfrac{1}{2} + k\Big)\theta\right) |\beta\rangle \,. \tag{5.1.5}$$

Proof. For $k = 0$ the claim is true by construction of θ. Assume now that (5.1.5) holds for a given $k \in \mathbb{N}$. Then

$$\begin{aligned}
\left(U_{\overline{\mathbf{0}}} U_f\right)^{k+1} |+\rangle &= \left(U_{\overline{\mathbf{0}}} U_f\right)\left(\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\alpha\rangle + \sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\beta\rangle \right) \\
&= U_{\overline{\mathbf{0}}}\left(\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\alpha\rangle - \sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\beta\rangle \right) \\
&= 2\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|+\rangle\langle+|\alpha\rangle - \cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\alpha\rangle \\
&\quad - 2\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|+\rangle\langle+|\beta\rangle + \sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\beta\rangle \\
&= 2\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)\cos\left(\tfrac{\theta}{2}\right)|+\rangle - \cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\alpha\rangle \\
&\quad - 2\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)\sin\left(\tfrac{\theta}{2}\right)|+\rangle + \sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\beta\rangle \\
&= 2\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)\cos^2\left(\tfrac{\theta}{2}\right)|\alpha\rangle - 2\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)\sin\left(\tfrac{\theta}{2}\right)\cos\left(\tfrac{\theta}{2}\right)|\alpha\rangle \\
&\quad - \cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\alpha\rangle + 2\sin\left(\tfrac{\theta}{2}\right)\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)\cos\left(\tfrac{\theta}{2}\right)|\beta\rangle \\
&\quad - 2\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)\sin^2\left(\tfrac{\theta}{2}\right)|\beta\rangle + \sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)|\beta\rangle\,.
\end{aligned}$$

Using trigonometric product-to-sum formulae (see, e.g., [58]), the first, second, fourth, and fifth term may be further expanded into terms consisting only of a single trigonometric function:

$$\begin{aligned}
&4\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)\cos^2\left(\tfrac{\theta}{2}\right) \\
&\quad = \cos\left(\left(\tfrac{3}{2}+k\right)\theta\right) + \cos\left(\left(\tfrac{1}{2}-k\right)\theta\right) + 2\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right) \\
&4\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)\sin\left(\tfrac{\theta}{2}\right)\cos\left(\tfrac{\theta}{2}\right) \\
&\quad = -\cos\left(\left(\tfrac{3}{2}+k\right)\theta\right) + \cos\left(\left(\tfrac{1}{2}-k\right)\theta\right) \\
&4\sin\left(\tfrac{\theta}{2}\right)\cos\left(\left(\tfrac{1}{2}+k\right)\theta\right)\cos\left(\tfrac{\theta}{2}\right) \\
&\quad = \sin\left(\left(\tfrac{3}{2}+k\right)\theta\right) + \sin\left(\left(\tfrac{1}{2}-k\right)\theta\right) \\
&4\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right)\sin^2\left(\tfrac{\theta}{2}\right) \\
&\quad = -\sin\left(\left(\tfrac{3}{2}+k\right)\theta\right) + \sin\left(\left(\tfrac{1}{2}-k\right)\theta\right) + 2\sin\left(\left(\tfrac{1}{2}+k\right)\theta\right).
\end{aligned}$$

Comparing the signs of the terms as they appear in initial calculation, one readily verifies that all terms containing an angle of $\left(\frac{1}{2}-k\right)\theta$ cancel each other out, and that

all terms containing the angle $\left(\frac{3}{2}+k\right)\theta$ sum up to the expected pre-factors. Lastly, all the remaining terms and the unexpanded terms in the initial calculation containing an angle of $\left(\frac{1}{2}+k\right)\theta$ eventually cancel each other out. Therefore, I have just shown that

$$\begin{aligned}\left(U_{\overline{\mathbf{0}}}U_f\right)^{k+1}|+\rangle &= \cos\left(\left(\tfrac{3}{2}+k\right)\theta\right)|\alpha\rangle + \sin\left(\left(\tfrac{3}{2}+k\right)\theta\right)|\beta\rangle \\ &= \cos\left(\left(\tfrac{1}{2}+(k+1)\right)\theta\right)|\alpha\rangle + \sin\left(\left(\tfrac{1}{2}+(k+1)\right)\theta\right)|\beta\rangle\end{aligned}$$

and the claim follows for all $k \in \mathbb{N}$ by induction. □

From (5.1.5) one can see that a Grover iteration performs a rotation by θ within the two-dimensional subspace spanned by $|\alpha\rangle$ and $|\beta\rangle$. Repeatedly performing a Grover iteration therefore rotates the initial vector closer to $|\beta\rangle$ such that a subsequent measurement will yield with high probability a solution from the superposition $|\beta\rangle$. However, by setting the number of Grover iterations too high, it is also possible to overshoot and to obtain a diminished success probability. More concretely, given a total rotation angle of $x\theta$, $x \in \mathbb{R}$, the probability of measuring a solution state is given by

$$P_{\text{succ}} = \left|\cos\left(\left(\tfrac{1}{2}+x\right)\theta\right)\langle\beta|\alpha\rangle + \sin\left(\left(\tfrac{1}{2}+x\right)\theta\right)\langle\beta|\beta\rangle\right| = \left|\sin^2\left(\left(\tfrac{1}{2}+x\right)\theta\right)\right|. \quad (5.1.6)$$

The success probability is maximized when the sine's argument becomes $k\pi/2$ for any $k \in \mathbb{Z}$. Since the rotation is counterclockwise, the first encounter will be at $\pi/2$. This yields the following ideal rotation:

$$\begin{aligned}\left(\frac{1}{2}+x\right)\theta = \frac{\pi}{2} \Leftrightarrow x = \frac{\pi-\theta}{2\pi} &= \left(\frac{\pi}{2} - \arccos\left(\sqrt{\tfrac{N-M}{N}}\right)\right)/\theta = \arcsin\left(\sqrt{\tfrac{N-M}{N}}\right)/\theta \\ &= \arcsin\left(\sqrt{1-\tfrac{M}{N}}\right)/\theta = \arccos\left(\sqrt{M/N}\right)/\theta. \quad (5.1.7)\end{aligned}$$

Choosing the integer k_x closest to the optimal x, i.e. it holds that $|x-k_x| \leq 1/2$, as number of Grover iterations produces the desired state $|\beta\rangle$ with an angle that is at most $\theta/2$ away from the optimal angle $\pi/2$. The case where $M > N/2$, i.e. more than half of the items are marked, shall not be of interest as classical linear search will give, on average, a solution after two steps, that is independently of the problem size N.[1] For the more interesting case of $M \leq N/2$, the following derivation is possible:

$$M \leq \frac{N}{2} \Leftrightarrow \frac{N-M}{N} \geq \frac{1}{2} \Rightarrow \frac{\theta}{2} = \arccos\left(\sqrt{\tfrac{N-M}{N}}\right) \leq \arccos\left(\sqrt{\tfrac{1}{2}}\right) = \frac{\pi}{4}. \quad (5.1.8)$$

[1] This reasoning is, of course, only valid if it is known in advance that $M > N/2$. By introducing an additional qubit and considering the search problem over now $N' = 2N$ states where a state is considered a solution if and only if it was a solution before and has a zero at the additional position, one can artificially achieve that $M' = M \leq N = N'/2$ such that the subsequent derivation is generally valid.

Therefore, the angle after k_x Grover iterations is at most $\pi/4$ away from $\pi/2$, i.e., within the interval $[\pi/4,\ 3\pi/4]$ on which $\sin^2$ only takes values $\geq 1/2$. Grover's algorithm thus succeeds with probability at least $1/2$. Furthermore, since the $\arccos(y) \leq \pi/2$ and $\sin(y) \leq y$ for all $y \geq 0$ it holds that

$$\left.\begin{array}{l} k_x \leq \left\lceil \arccos\left(\sqrt{M/N}\right)/\theta \right\rceil \leq \left\lceil \frac{\pi}{2\theta} \right\rceil \\ \frac{\theta}{2} \geq \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}} \end{array}\right\} \Rightarrow k_x \leq \left\lceil \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rceil . \tag{5.1.9}$$

(5.1.9) truly is a remarkable result. It entails that the average runtime of Grover's algorithm is in $O(\sqrt{N/M})$ which has to be contrasted with the average runtime of $O(N/M)$ provided by classical linear search. Surely, the quantum oracle is usually more expensive to implement than its classical counterpart, but this is solely due to hardware challenges. In principle, both circuits can be executed with a similar number of gates such that Grover's algorithm indeed achieves a quadratic speed-up over linear search. While unstructured search is rather rare in real world application as, e.g., databases are usually sorted, the scope of Grover's algorithm has already been extended to a variety of problems such as solution counting [59], non-uniform search [13], and discrete optimization [60]. All these extensions follow the blueprint laid out by Grover's algorithm and manifested in Lemma 5.1: Prepare an initial superposition that has non-zero overlap with at least one of the solution states and alternately apply a quantum oracle, flipping the phase of all solution states, and a diffusion operator which flips the phase in all directions orthogonal to the initial superposition. Together, the quantum oracle and adjusted diffusion operator function as a rotation within the two-dimensional subspace spanned by the feasible portion of the initial state and the superposition of solution states. The geometric intuition gained from the previous considerations remains valid in the more general case; the larger the overlap of initial state and solution state, the fewer rotations are required, and the closer the optimal rotation angle is to a multiple of the initial angle, the higher is the probability of measuring a solution state at the end.

In practice, the construction of a suitable quantum oracle can often be easily derived from the classical problem description at hand while designing a favorable initial superposition which is also efficiently preparable is typically more challenging. Conversely, if a favorable initial superposition can be found, there is often a classical probabilistic algorithm that samples from the same probability distribution corresponding to the superposition and thereby already achieves good results. Accordingly, whether Grover's algorithm and derived routines will ever offer practical speed-ups compared to classical algorithms has been the topic of debate for several years (see, e.g., [15, 17, 61]).

5.2 Implementation

The first step to implementing Grover's algorithm is to specify a concrete search problem. For simplicity, I may assume a problem size of $N = 2^{64}$ and model each possible item with a 64-bit integer. A suitable oracle function should input a candidate, i.e. an integer, and return a boolean truth value: `true` for marked items, `false` for unmarked items. Furthermore, the oracle function should be quantizable as described in the previous chapter. An example for such an oracle function is the parity check introduced earlier which returns `true` for even numbers and `false` for odd numbers. In this case, exactly half of the candidates are solutions, which yields an optimal number of Grover iterations of roughly `GROVER_ITER = 2`.

For creating the initial uniform superposition, an approach could have been to introduce this frequently used operation as an extra keyword or a macro-like function. However, using CQ's syntax for superposition, this can be compactly implemented without the need for enriching the language's vocabulary. As a reminder, in CQ, defining `quantum int a = [f]` with a quantizable function of signature `bool f(int)` initializes the quantum integer `a` in a uniform superposition of all CB states $|\boldsymbol{x}\rangle$ such that `f`, given $\boldsymbol{x}$ as its classical input, evaluates to `true`. In order to create the uniform superposition of all CB states, I may simply introduce a function `all_true` which returns `true` regardless of its input, thereby establishing a short implementation for this very important subroutine. By altering the quantizable function `f`, the very same syntax may be employed to create arbitrary uniform superpositions.

After properly initializing a 64-qubit quantum integer with the aforementioned method, the heart of Grover's algorithm consists of a repeated application of the quantum oracle and the diffusion operator. The repetition is controlled by a classical `for`-loop, running `GROVER_ITER`-many times. The quantum oracle is required to flip the phase of all CB states $|\boldsymbol{x}\rangle$ for which the classical oracle returns `true` on input $\boldsymbol{x}$. The feasibility check can be naturally implemented in CQ by calling the oracle with the quantum integer as input inside an `if`-condition. As the classical oracle function is required to be quantizable, inputting a quantum integer instead of a regular one is permitted and returns a quantum bool. This, in turn, means that the `if`-statement will be quantized as well and all statements within the `if`-body are implemented as quantum circuits controlled on the outcome of the quantized oracle function. A generally valid implementation of this construction is to introduce an ancilla quantum integer, copy the original quantum integer via a layer of CNOT gates to the ancilla register, and to subsequently execute the quantized oracle function on the copy. Additional qubits may be required for the quantized routine such as, e.g., a qubit for storing the result of the calculation, where $|0\rangle$ is associated with `false` and $|1\rangle$ with `true`. In this particular case, the function can be evaluated without altering the input or the need for an

ancilla qubit to store the result at. The `if`-body consists of adding one to the phase of the original quantum integer. The statement `phase (a) += b` is to be understood as applying a phase shift of $\pi/$`b` to `a`, i.e. `b` $= 1$ realizes a full phase flip. The phase shift can be implemented in full generality by introducing an additional qubit (initialized in $|0\rangle$), flipping it with an `X` gate and subsequently applying a rotational gate `R` as in (3.2.3) with a properly set angle. The entire `if`-statement therefore precisely matches the usual implementation of the quantum oracle, but expressed in logical abstraction and in terms of the classical problem rather than a seemingly unmotivated sequence of quantum gates.

The diffusion operator is implemented in three steps: First, apply the same quantum circuit which has been used for creating the initial superposition. In CQ, this is achieved via the extended syntax for function calls which allows for statements of the form `[f](a)` and follows the same semantics as the previously discussed superposition syntax for variable definitions. Second, a phase flip has to be applied to all CB states except the $|\mathbf{0}\rangle$ state which can be elegantly formulated via a quantized `if`-statement, containing the corresponding phase shift in its body. The `if`-condition reads `a != 0` since the inequality-operator is overloaded for quantum to classical values. This check can again be implemented without altering the input `a` such that no quantum copy of `a` is necessary. In a concrete implementation, checking whether `a == 0` is more straightforward, and a generally valid approach to utilize this fact is to simply check for the negated condition, write the result to an ancilla qubit, and to subsequently flip the ancilla qubit with an `X` gate. Another generally valid approach is to first apply all operations in the `if`-body uncontrolled and to apply their inverse controlled on the negated condition. In this concrete case of a single phase flip to be executed, another implementation, as already highlighted in the previous section is also suitable: sandwiching a `Z` gate on any of the target register's qubits, controlled on all other qubits of the target register, between two layers of `X` gates. Lastly, the superposition-creating circuit has to be applied again which follows the exact same syntax as before.

The final step in Grover's algorithm is to measure the processed quantum state in the CB in order to obtain a bit string, representing (the index of) a marked item with high probability. Accordingly, the last line of CQ-code consists of the `measure` keyword being applied to the quantum integer which indeed returns a classical integer. Fig. 5.1 depicts the just discussed implementation of Grover's algorithm in CQ as a minimum working example, constituting Grover's algorithm as a single executable which returns the item index. In principle, the outcome of Grover's algorithm may be processed further within a broader program. A promising example would be to substitute linear search routines in a complex C program with the CQ implementation of Grover's algorithm.

```
const unsigned GROVER_ITER = 2; // number of iterations

// exemplary quantizable oracle function
bool oracle(int x) {
    return (x & 1) == 1;
}

// quantizable function used for state preparation
bool all_true(int x) {
    return true;
}

int main() {
    quantum int state = [all_true]; // create superposition

    for (unsigned i = 0; i < GROVER_ITER; i += 1) {
        // quantum oracle
        if (marker(state)) {
            phase (state) += 1;
        }

        // apply layer of Hadamard gates
        [all_true](state);

        if (state != 0) {
            phase (state) += 1;
        }

        // apply layer of Hadamard gates
        [all_true](state);
    }

    return measure (state);
}
```

Figure 5.1: Grover's algorithm in CQ. The quantizable oracle function returns `true` for odd numbers and `false` for even numbers. The quantum integer `state` is initialized in uniform superposition via the quantizable helper function `all_true` which returns `true` for every input. Subsequently, a classical `for`-loop with `GROVER_ITER` many iterations contains the alternating application of the quantum oracle and the diffusion operator. The former is comprised of a quantized `if`-statement, controlling a phase flip on `state` on whether the (quantized) oracle returns `true` while the latter consists of sandwiching a quantized `if`-statement,which flips the phase of `state` conditioned on `state != 0`, between two applications of the superposition-creating circuit. Finally, `state` is measured and the classical outcome is returned.

5.3 Comparison to Existing Frameworks

In this section, I compare the implementation of Grover's algorithm with the parity check oracle and the standard diffusion operator in CQ to its implementations in Quipper, QCL, OpenQASM, Q#, and Silq. Qrisp would have been also an interesting candidate to compare to, but its focus on quantum floats rather than quantum integers renders a fair comparison rather difficult. As a rule of thumb, the higher-level the language the longer the source code for this specific example. This is mainly due to the fact that the quantum oracle takes up one line of code when expressed in terms of elementary quantum gates: a `Z` gate applied to the least significant qubit in the main register. The only exception to this rule is Quipper which, despite being relatively low-level, requires several lines of code to implement the quantum oracle.

In essence, I find that only Silq provides the possibility of quantizing a classical function and `if`-statements for usage within a quantum routine. In all other languages, the quantum oracle has to be given in terms of quantum gates. Additionally, Silq is the only other language which provides native support of quantum integers like CQ does. In all other cases, one has to use arrays of qubits of suitable length. The diffusion operator looks very similar in Quipper, QCL, and OpenQASM, being a sequence of elementary quantum gates applied to the different qubits of the main register. Q#, in turn, offers a feature that lets me express the sandwiching layers of Hadamard and `X` gates as a conjunction of operators which perfectly matches their higher-level meaning. Only in Silq, I have been able to express the core part of the diffusion operator, the phase flip for all CB states except $|\mathbf{0}\rangle$, as a conditional statement. However, even in Silq, the enclosing layers of Hadamard gates, in contrast to CQ, have to be specified directly. Given the language-specific implementations of the quantum oracle and the diffusion operator, the skeleton for Grover's algorithm looks nearly identical in all languages, including CQ: After initializing the target in uniform superposition, a simple `for`-loop with a prescribed number of iterations alternately applies the quantum oracle and the diffusion operator to the target state. Subsequently, the state can be measured via the language's respective keyword.

Admittedly, Grover's algorithm looks very similar in CQ and Silq with the main exception lying in CQ's extended syntax for creating equal superpositions via function calls rather than quantum gate sequences. The feature of quantized `if`-statements is indeed analogously implemented in both languages. However, both languages follow a somewhat different design idea: While CQ enforces the programmer to explicitly declare whether a variable is quantum, Silq assumes this by default and has to track whether a variable remains in a CB state throughout the program. Furthermore, CQ automatically checks whether a provided function is quantizable or unitary, while this has to be annotated explicitly in Silq with keywords such as `qfree`, `lifted`, and

`mfree`.

```
-- exemplary oracle function
oracle :: [Qubit] -> Circ [Qubit]
oracle qubits = do
    let lsb = last qubits
    gate_Z_at lsb
    return qubits

-- diffusion operator
diffuse :: [Qubit] -> Circ [Qubit]
diffuse qubits = do
    hadamard_all qubits
    qnot_all qubits
    gate_Z_at (qubits !! 0) `controlled` (tail qubits)
    qnot_all qubits
    hadamard_all qubits
    return qubits

-- Grover's algorithm
grovers_algorithm :: (Int, Int) -> Circ Int
grovers_algorithm (n, iter) = do
    qubits <- qinit (replicate n False)
    hadamard_all qubits
    forM_ [1..iter] $ \_ -> do
        qubits <- oracle qubits
        qubits <- diffuse qubits
    result <- qmeas qubits
    return (from_bitlist result)
```

Figure 5.2: Grover's algorithm in Quipper. Quipper is a functional programming language, realized as Haskell extension. Support for quantum resources is enabled via additional keywords such as `Qubit` (a quantum bool in CQ) and various low-level quantum gates such as `X` `Y`, and `Z` gates (via `qnot`, `gate_Y`, and `gate_Z`) or Hadamard gates (via `hadamard`). Quantum operations can be controlled on qubits, but not on more complex conditions. The exemplary oracle function has to be given as a quantum circuit which applies a `Z` gate to the least significant qubit in the main register. Similarly, the diffusion operator has to be written as a function that performs a multi-controlled `Z` gate on the zeroth qubit qubit of the main register, controlled on the remaining main qubits and sandwiched between two layers of Hadamard gates and `X` gates, respectively. Lastly, Grover's algorithm consists of preparing the main register and the flag qubit, iteratively applying the previously defined oracle and diffusion functions, and to return the measured main register. While Quipper encapsulates all relevant quantum operations with Haskell-like syntax, classical and quantum resources are always treated separately, not supporting analogies of e.g. classical and quantum control flows.

```
1  operator diffuse(qureg q) { // diffusion operator
2      H(q);
3      Not(q);
4      CPhase(pi, q);
5      !Not(q);
6      !H(q);
7  }
8
9  procedure grover(int n, int iter) { // Grover's algorithm
10     int x;
11     int i;
12     qureg q[n];
13     reset;
14     H(q);
15     for i = 1 to iter {
16         Z(x[0]); // exemplary oracle function
17         diffuse(q);
18     }
19     measure q, x;
20     print "measured", x;
21     reset;
22 }
```

Figure 5.3: Grover's algorithm in QCL. QCL can treat most classical functionality in a C-like manner. In order to also handle quantum variables and functions, QCL introduces additional keywords like `operator` (comparable to unitary functions in CQ), `qufunct` (similar to quantizable functions in CQ), and `qureg` (would be a quantum boolean array in CQ). The displayed source code is an adaptation of the example given in the latest QCL release [62]. The exemplary oracle function can be expressed with one line of code (line `16`) and is therefore not introduced as an additional function/operator. The diffusion operator (lines `1`–`7`) reads similar to its implementation in Quipper (see Fig. 5.2); a `Z` gate on the zeroth qubit in the main register is controlled on all other qubits and explicitly sandwiched between two layers of Hadamard gates and `X` gates, respectively. Grover's algorithm (lines `9`–`22`) consists of initializing an `n`-qubit main register `q`, preparing the main register in uniform superposition, iteratively performing the exemplary phase oracle and the diffusion operator `iter`-times, measuring the final state, and storing the result in the classical integer `x`. Afterwards, all qubits are reset. While the extended utility of QCL allows to properly treat classical and quantum resources at the same time, support for straight-forward quantization of classical primitives is missing. Ultimately, QCL offers a proper interface to integrate explicit quantum gate sequences into larger hybrid programs. However, it does not offer many simplifications when writing pure quantum code.

```
1  define n_qubits(n: int);
2  define iterations(iter: int);
3
4  qubit[n] q;
5  bit[n] result;
6
7  // diffusion operator
8  gate diffuse q {
9      h q;
10     x q;
11     ctrl @ z q[0:n_qubits - 1];
12     x q;
13     h q;
14 }
15
16 // Grover's algorithm
17 h q;
18 for i in [0:iterations - 1] {
19     z q[n_qubits - 1]; // exemplary oracle function
20     diffuse q;
21 }
22 measure q -> result;
```

Figure 5.4: Grover's algorithm in OpenQASM. OpenQASM is the currently most prominent "high-level" quantum programming language, compatible with e.g. IBM's Qiskit and Xanadu's Pennylane. The fundamental quantum data type in OpenQASM is the `qubit` (a quantum bool in CQ) and a `gate` is a unitary instruction acting on one or several `qubit` registers/arrays (corresponds to a unitary function in CQ). The exemplary oracle function can again be expressed with one line of code (line 19) and is therefore not introduced as a stand-alone `gate`. The diffusion operator (lines 8–14) reads similar to the implementations in Quipper (see Fig. 5.2) and QCL (see Fig. 5.3); applying a layer of Hadamard and `X` gates to all qubits in the main register, applying a `Z` gate on the zeroth qubit, controlled on all other qubits, and applying another layer of `X` and Hadamard gates on all qubits. Grover's algorithm (lines 17-22) then consists of applying a layer of Hadamard gates to the `n_qubits`-qubit register `q`, a subsequent iterative application of the exemplary quantum oracle and the previously defined `diffusion gate`, and a final measurement of `q` to store the classical outcome in the `n_qubits`-bit register `result`. While OpenQASM combines classical operations and control structures with basic quantum utility, it does not provide quantization schemes of any kind; quantum analogues of integers to not exist and logical operations as well as control structures are not overloaded for qubits. In contrast, OpenQASM popularity mainly stems from its advanced compatibility with several quantum hardware architectures. It should therefore ultimately serve as a possible intermediate step when compiling higher-level languages like CQ or Silq down to hybrid machine code.

```
operation Main() : Result[] { // Grover's algorithm
    let n = 64;
    let iter = 2;
    use qubits = Qubit[n];
    UniformPrep(qubits);
    for _ in 1..iter {
        Z(qubits[0]); // exemplary oracle function
        Diffuse(qubits);
    }
    return MResetEachZ(qubits);
}

operation UniformPrep(inputQubits : Qubit[]) : Unit is Adj + Ctl {
    for q in inputQubits {
        H(q);
    }
}

operation Diffuse(qubits : Qubit[]) : Unit { // diffusion operator
    within {
        Adjoint UniformPrep(qubits);
        for q in qubits {
            X(q);
        }
    } apply {
        Controlled Z(Most(qubits), Tail(qubits));
    }
}
```

Figure 5.5: Grover's algorithm in Q#. Q# is the high-level quantum programming language behind Microsoft's quantum development kit. The displayed source code is an adaptation of the example given in the official Q# algorithms section. Similar to OpenQASM, the fundamental quantum data type in Q# the `Qubit`. An `operation` is any function that affects a qubit register. These can be annotated as adjoinable/unitary (`Adj`) and controllable (`Ctl`) as has been done for the uniform state preparation (`UniformPrep`). Using the `within`-keyword, an `operation` can be executed between another `operation` and its adjoint which is a convenient shortcut for the sandwiching with Hadamard and `X` gate layers in the diffusion operator. Grover's algorithm has the typical form of first initializing an `n`-qubit register in uniform superposition, then iteratively applying the exemplary quantum oracle and the diffusion operator, and returning the result of measuring the qubit register in the CB. Although subroutines can be nicely packaged into functions, one ultimately has to supply low-level quantum instructions on a gate level in order to implement Grover's algorithm in similar procedures in Q#.

```
GROVER_ITER := 2: !ℕ;
n := 64: !ℕ;

def oracle(const cand: int[n]) lifted: 𝔹 { // oracle function
    return (cand & 1) == 1;
}

def diffuse(cand: int[n]) mfree: int[n] { // diffusion operator
    for k in [0..n) {
        cand[k] := H(cand[k]);
    }
    if cand != 0 {
        phase(π);
    }
    for k in [0..n) {
        cand[k] := H(cand[k]);
    }
    return cand;
}

def main() { // Grover's algorithm
    cand := 0: int[n];
    for k in [0..n) {
        cand[k] := H(cand[k]);
    }
    for k in [0..GROVER_ITER) {
        if oracle(cand) {
            phase(π);
        }
        cand := diffuse(cand);
    }
    return measure(cand);
}
```

Figure 5.6: Grover's algorithm in Silq. Silq is a proper high-level quantum programming language which offers rich support for quantization of classical functions. In Silq, variables are considered quantum by default (unless annotated with a leading exclamation mark), but realized as classical variables if they are not brought into superposition. For example, the exemplary oracle function can be written as in CQ, but additionally has to be annotated as `lifted`, indicating that no superpositions are created and that all input variables are held constant. The diffusion operator has to implement the Hadamard gates explicitly, but can describe the conditional phase flip via quantized `if`-statement like in CQ. Similarly, Grover's algorithm has to explicitly apply a Hadamard gate to each qubit of the quantum integer `cand`. Subsequently, the exemplary oracle function can be used with quantum input inside an `if`-condition, controlling the phase flip. Finally, the quantum state `cand` is measured and the measurement outcome is returned as classical integer.

CHAPTER 6

Conclusion and Outlook

In this thesis, I have proposed a high-level programming language for expressing integer-based quantum-classical programs: CQ. The main feature of CQ is to provide several quantization schemes for classical concepts such as logical, bitwise, and arithmetic operations as well as `if`- and `switch`-statements, and to address both the classical functionality and its quantization with the same syntax. For this purpose, I have detailed explicit quantization strategies for all the aforementioned concepts in Chapter 3. While the literature on quantum arithmetic is quite rich, little work has been presented on quantizing classical control structures such as `if`-statements. However, all issues addressed in this thesis admit a short and elegant solution which are probably implemented in the same or a similar way in frameworks like QCL or Silq. Building on these quantization schemes, I have introduced the vocabulary, grammar, and semantics of CQ in Chapter 4 alongside a prototypical implementation of the parser (available under [32]). Finally, I have given a fairly detailed introduction to Grover's algorithm. The proof to Lemma 5.1 is a more rigorous, but also lengthier alternative to proofs found in the literature. The subsequent comparison of implementations of Grover's algorithm in various quantum programming languages (CQ, Quipper, QCL, OpenQASM, Q#, and Silq) provides an important benchmark set for assessing a language's expressibility.

As already mentioned several times, the provided parser (and the design of CQ in general) are prototypical. My main intent was to deliver a working parser software for a compact set of rules and functionalities. Due to this reason, I decided to develop the parser independently of the existing software stack for C compilers, to avoid the enormous initial task of sufficiently understanding these massive code libraries, although CQ is meant to integrate well with C. In the following, I wish to comment on plausible next steps which would elevate CQ to a promising bridge between classical and quantum computing with real-world applications. The next step clearly has to be introducing CQ as a proper extension of C: This task involves cloning the source code of one's favorite C compiler and building the extended functionality of CQ on top of C's scope. While the introduction of the three new keywords `measure`, `phase`, and `quantum` and their additional syntactical rules should be relatively straight-forward,

implementing the checks for quantizable and unitary functions might necessitate diving deep into the original source code. Furthermore, on the one hand, some design choices I have made such as Java-style array annotations have to be reverted in order to comply with the C syntax. On the other hand, some C concepts (most notably pointers, floating point numbers, and a printing mechanics) are missing in my prototype and have to be thoroughly aligned with CQ's functionality. I assume the first task to not require any intellectual overhead, since this would already amount to changing only a few conventions within the current parser source code. Also for the latter task, I am quite optimistic that the current state of research allows for finding quick solutions for the quantization of each C concept or to argue why it should not be lifted to a quantum operation.

Preprocessor directives: Since the preprocessing of source and header files has nothing to do with the actual language features, all preprocessor directives may be adopted without any changes. This also includes the macro-specific keyword `_Generic` as in introduced in C11. Additional directives addressing quantum hardware details could prove themselves useful in the future.

Linker: The linking process for a hybrid architecture should be similar to the one on purely classical computers. Therefore, the C keywords `extern` and `static` as well as `inline` as introduced in C99 which communicate storage management with the linker may be adopted without any changes.

Goto: The only control structure I did not take into consideration so far is `goto`. Similar to the other jump statements `break` and `continue`, `goto` may only be allowed in a classical context and render the enclosing function non-quantizable and non-unitary.

Primitive types: Following the quantization schemes implemented in Qrisp, the primitive types `float` and `double` may also carry the `quantum` qualifier and arithmetic operations may also be overloaded to accept quantum floating point numbers as operands. Via the usual implementation of complex numbers as two floating point numbers, quantum versions of the `_Complex` and `_Imaginary` keywords introduced in C99 as well as complex operations and functions may also possess quantum versions. Other primitive types (`signed` and `unsigned` versions of `char`, `long`, and `short`) which are all integer-like may as well carry the `quantum` qualifier; quantum versions of bitwise and arithmetic operations with these operands are essentially the same as the ones presented in this thesis. In C23, the `_BitInt` and `unsigned _BitInt` types are introduced for bit-precise integers. This perfectly matches the functionality of declaring bit/qubit registers of difference sizes (as, e.g., in OpenQASM) and such bit-precise integers may also carry the `quantum` qualifier. Bitwise and arithmetic quantum operations readily generalize for bit/qubit-precise operands. In addition, C23 introduced decimal floating point numbers of various precision via the keywords `_Decimal32`, `_Decimal64`, and `_Decimal128`. Although there is, to my knowledge, yet no literature on quantizations of decimal floating point arithmetic, I am convinced that adapting the classical circuits in a similar fashion to ordinary floating point arithmetic should be sufficient in order to allow also for quantum versions for this new data type and operations with it.

Compound types: Maintaining a type system will almost certainly remain a purely classical task since it is an already well established concept, is typically not a computational bottleneck, and requires precise, deterministic checks. Correspondingly, C's functionality for creating custom types via `enum`, `struct`, and `union` may be adopted only with small adjustments. An `enum` may also have a quantum version. Currently, I do not see any problem with allowing both classical and quantum members in a `struct`, even at the same time, while all members of a `union` may either be purely classical or purely quantum since it is otherwise not possible for them to share the same memory. Note, however, that introducing these more complex types will require more complicated checks for assessing whether a function using those types is quantizable or unitary. Other type-specific features (`sizeof` and `typedef` as well as the operators `typoeof` and `typeof_unqual` as in introduced in C23) may be adopted straight-forwardly.

Pointers and memory management: Pointers are undoubtedly one of the most powerful features of C and properly implementing them as well as associated concepts (`restrict` from C99 and `nullptr` from C23) could be done in two fundamentally different ways. Which variant to use highly depends on future decisions regarding the management of quantum memory. The first option is to allow only for classical addresses. This means that the address of any variable, regardless on whether variable is classical or quantum, can only be classical, i.e. `quantum int * p;` would be a valid declaration of a pointer to a quantum integer, while `int * quantum p;` would be an illegal declaration of a quantum pointer to a classical integer. Alternatively, one could also allow for addresses being stored in superposition as proposed in [63]. In this case, also the second declaration would be valid. While the second option definitely is more general and allows for more powerful and interesting programs, it also complicates the type system further as soon as quantum pointers to classical memory are allowed. In general, memory management on quantum architectures is a broad topic and different design choices will influence which classical concepts to transfer to the quantum case. Concretely, in C there exists the (legacy) `auto` specifier, indicating that the annotated variable shall be destroyed upon exiting its scope. By default all variables have automatic memory management in C. This is in contrast to dynamically allocated memory (via `malloc` and similar functions) where the memory has to be freed by the programmer. Depending on which quantum storage class is implemented, one of these functionalities may not be applicable to quantum variables. Furthermore, classical variables may exist only on a register, annotated via the `register` specifier, and do not have an address. Depending on the available/chosen architecture, the `register` keyword might not be applicable to quantum variables either. Another low-level functionality addressed in C (via `_Alignas` and `_Alignof` since C11 and their successors `alignas` and `alignof` since C23) is alignment requirement which is the number of bytes between successive addresses at which objects of the respective type can be allocated. Whether these keyword may also be used for quantum variables highly depends on the actual implementation of the quantum memory's address space.

Multiprocessing and multithreading: Similar to implementing a type system, managing processes and threads as well as their associated resources will probably always remain a task for the classical computer. QPU will most likely solely enter as additional computational resources which are assigned and revoked by an operating system, running on the classical computer. Accordingly, C's functionalities for context management (`volatile` as well as `_Atomic` and `_Thread_local` as introduced in C11 with the latter being deprecated and replaced with `thread_local` in C23) may be adopted without any change.
Other keywords: In the above paragraphs, I covered all keywords introduced since C89 except for four keywords which do not belong to any of the broader topics discussed. First, the `_Noreturn` annotation, introduced in C11, for functions tells the compiler that the function does not return to its caller. This can have several reasons with exiting the entire program as one of them. At this point I do not see any chance that a function which is rightfully annotated with `_Noreturn` can be quantizable or unitary since it typically means that a jump statement is involved for which there is no quantum counterpart. Second and third, `_Static_assert`, introduced in C11, and its successor `static_assert` from C23 are compile-time checks for constant expressions. For the latter, the annotation `constexpr` has been introduced in C23. While other languages such as QCL also propose constant quantum variables and expressions, this is not the case within CQ. Therefore, these concepts may remain only be applicable to classical values.

Suitably adopting all the content from the exhaustive list of C keywords would enable CQ as a proper C extension and would also massively increase CQ's practicability. I wish now to discuss further enhancements which are not necessary in order to make C and CQ compatible, but which allow to compactly express more advanced quantum algorithms in CQ.
Quantizable and unitary for-loops: In my prototypical version of CQ, `for`-loops are never quantizable or unitary. However, many `for`-loops in practice are (counter-)controlled loops where the loop variable is not altered within the loop body. If additionally, the number of loop iterations is known before executing the loop, the loop variable may be considered `const` during each iteration of the loop. In this case, an otherwise quantizable/unitary loop body would remain quantizable/unitary. Implementing this feature in a future version of CQ is highly desirable for expressing e.g. more complicated state preparation routines, which apply different gates to each qubit in a register (as in [17]), compactly with a unitary `for`-loop.
Parameterized gates: Currently, only the phase shift is parameterizable via the right-hand side of a phase-shift-statement. This has to be extended in a meaningful way to a broader class of quantum gates and subroutines. First and foremost, [17] and many others utilize parameterized `Ry` gates to bring qubits from the $|0\rangle$ state to a non-uniform superposition between $|0\rangle$ and $|1\rangle$ with amplitudes directly corresponding to the chosen parameters. To some extend, this can be understood as the more general

version of applying a Hadamard gate to $|0\rangle$ which creates a uniform superposition of the two CB states. Since this is such an important ingredient of many quantum algorithms, especially those which tackle classical problems, it may be appropriate to introduce an additional keyword `branch` in a future version of CQ, implementing a parameterized `Ry` gate. Alternatively, the low-level access to single quantum gates may be enabled by introducing elementary gates as functions which expect a quantum input. This would then also enable low-level coding in CQ as already present in the other languages discussed in Section 5.3.

Quantum linear algebra: A powerful alternative to mapping bits to qubits for the quantization of floating point numbers is to encode real and complex numbers of arbitrary precision directly into the continuous state space of qubits (see e.g. [64]). Namely, for a given vector $v \in \mathbb{C}^N$ with $\|v\|_2 = 1$, there exists a state preparation circuit that prepares the $\lceil \log_2 N \rceil$-qubit state $|v\rangle = \sum_{i=1}^{N} |i\rangle$ with depth $O(N/\log N)$, yielding an exponentially reduced encoding compared to the usual encoding. This method only works for normalized vectors v while for unnormalized input there exist approximate schemes using additional ancilla qubits. It is particularly well studied in the context of *quantum linear algebra*, a discipline addressing linear algebraic calculations such as matrix-vector multiplication on a quantum computer. Although conceptually challenging, there might exist also (approximate) versions of arithmetic operations for these compactly encoded real/complex numbers. This definitely requires further research but might be worth tracking in order to eventually provide an additional, presumably more powerful quantization scheme for floating point numbers.

Bibliography

[1] P. Benioff. "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines". In: *J. Stat. Phys.* 22 (1980), pp. 563–591.

[2] D. Deutsch. "Quantum theory, the Church-Turing principle and the universal quantum computer". In: *Proc. Roy. Soc. London A Math. Phys. Sci.* 400 (1985), pp. 97–117.

[3] Umesh Vazirani. "A survey of quantum complexity theory". In: *Proceedings of Symposia in Applied Mathematics*. Vol. 58. 2002, pp. 193–220.

[4] R. P. Feynman. "Simulating physics with computers". In: *Int. J. Theor. Phys.* 21 (1982), pp. 467–488.

[5] I. M. Georgescu, S. Ashhab, and Franco Nori. "Quantum simulation". In: *Rev. Mod. Phys.* 86.1 (2014), pp. 153–185.

[6] Sam McArdle et al. "Quantum computational chemistry". In: *Rev. Mod. Phys.* 92.1 (2020), p. 015003.

[7] Ryan Babbush et al. "Low-Depth Quantum Simulation of Materials". In: *Phys. Rev. X* 8.1 (2018), p. 011044.

[8] Ethan Bernstein and Umesh Vazirani. "Quantum complexity theory". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 11–20.

[9] David Deutsch and Richard Jozsa. "Rapid solution of problems by quantum computation". In: *Proc. Roy. Soc. London A Math. Phys. Sci.* 439.1907 (1992), pp. 553–558.

[10] Daniel R. Simon. "On the Power of Quantum Computation". In: *SIAM J. Comput.* 26.5 (1997), pp. 1474–1483.

[11] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509.

[12] L. K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proc. Annu. ACM Symp. Theory Comput.* 1996, pp. 212–219.

[13] Gilles Brassard et al. “Quantum amplitude amplification and estimation”. In: *Contemp. Math.* (2002), pp. 53–74.

[14] J. Preskill. “Quantum Computing in the NISQ era and beyond”. In: *Quantum* 2 (2018), p. 79.

[15] Chris Cade et al. “Quantifying Grover speed-ups beyond asymptotic analysis”. In: *Quantum* 7 (2023), p. 1133.

[16] Sabrina Ammann et al. *Realistic Runtime Analysis for Quantum Simplex Computation*. 2023. arXiv: 2311.09995 [quant-ph].

[17] Sören Wilkening et al. *A quantum algorithm for solving 0-1 Knapsack problems*. 2024. arXiv: 2310.06623 [quant-ph].

[18] M. Cerezo et al. “Variational Quantum Algorithms”. In: *Nat. Rev. Phys.* 3.9 (2021), pp. 625–644.

[19] Ville Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2022. arXiv: 1811.04968 [quant-ph].

[20] Ali Javadi-Abhari et al. *Quantum computing with Qiskit*. 2024. arXiv: 2405.08810 [quant-ph].

[21] Cirq Developers. *Cirq*. Version 1.4.0. 2024. URL: https://doi.org/10.5281/zenodo.11398048.

[22] Amazon Web Services. *Amazon Braket*. 2020. URL: https://aws.amazon.com/braket/.

[23] Microsoft. *Azure Quantum*. 2021. URL: https://azure.microsoft.com/en-us/services/quantum/.

[24] Andrew Cross et al. “OpenQASM 3: A Broader and Deeper Quantum Assembly Language”. In: *ACM Trans. Quantum Comput.* 3.3 (2022), pp. 1–50.

[25] Krysta Svore et al. “Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL”. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018.

[26] Benoît Valiron et al. “Programming the quantum future”. In: *Commun. ACM* 58.8 (2015), pp. 52–61.

[27] Bernhard Ömer. “Classical Concepts in Quantum Programming”. In: *Int. J. Theor. Phys.* 44 (2005), pp. 943–955.

[28] Benjamin Bichsel et al. “Silq: a high-level quantum language with safe uncomputation and intuitive semantics”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 286–300.

[29] Raphael Seidel et al. *Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers*. 2024. arXiv: 2406.14792 [quant-ph].

[30] Vern Paxson. *flex*. Version 2.6.4. 2017. URL: https://github.com/westes/flex.

[31] GNU Project. *Bison*. Version 3.8.2. 2021. URL: https://www.gnu.org/software/bison.

[32] Lennart Binkowski. *CQ Parser*. Version 1.0.1. 2024. URL: https://github.com/MarkAureli/cq_parser.

[33] Jens Mönig and Brian Harvey. *Snap!* Version 9.2.17. 2024. URL: https://github.com/jmoenig/Snap.

[34] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[35] G. J. Milburn. "Quantum optical Fredkin gate". In: *Phys. Rev. Lett.* 62.18 (1989), pp. 2124–2127.

[36] Emanuel Knill, Raymond Laflamme, and Gerald J Milburn. "A scheme for efficient quantum computation with linear optics". In: *Nature* 409.6816 (2001), pp. 46–52.

[37] Daniel Loss and David P. DiVincenzo. "Quantum computation with quantum dots". In: *Phys. Rev. A* 57.1 (1998), pp. 120–126.

[38] Bruce E Kane. "A silicon-based nuclear spin quantum computer". In: *Nature* 393.6681 (1998), pp. 133–137.

[39] Gavin K. Brennen et al. "Quantum Logic Gates in Optical Lattices". In: *Phys. Rev. Lett.* 82.5 (1999), pp. 1060–1063.

[40] J. I. Cirac and P. Zoller. "Quantum Computations with Cold Trapped Ions". In: *Phys. Rev. Lett.* 74.20 (1995), pp. 4091–4094.

[41] John M Martinis et al. "Rabi oscillations in a large Josephson-junction qubit". In: *Phys. Rev. Lett.* 89.11 (2002), p. 117901.

[42] Charles H Bennett et al. "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels". In: *Phys. Rev. Lett.* 70.13 (1993), p. 1895.

[43] Alain Aspect, Philippe Grangier, and Gérard Roger. "Experimental tests of realistic local theories via Bell's theorem". In: *Phys. Rev. Lett.* 47.7 (1981), p. 460.

[44] Charles H Bennett. "Logical reversibility of computation". In: *IBM J. Res. Dev.* 17.6 (1973), pp. 525–532.

[45] Tommaso Toffoli. "Reversible computing". In: *International colloquium on automata, languages, and programming*. Springer. 1980, pp. 632–644.

[46] Phil Gossett. *Quantum Carry-Save Arithmetic*. 1998. arXiv: quant-ph/9808061.

[47] Richard Peirce Brent and Hsiang Te Kung. "A Regular Layout for Parallel Adders". In: *IEEE Trans. Comput.* C-31.3 (1982), pp. 260–264.

[48] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Math. Comput.* 19.90 (1965), pp. 297–301.

[49] Christos H. Papadimitriou. "Optimality of the Fast Fourier transform". In: *J. ACM* 26.1 (1979), pp. 95–102.

[50] D. Coppersmith. *An approximate Fourier transform useful in quantum factoring*. 2002. arXiv: quant-ph/0201067.

[51] Thomas G. Draper. *Addition on a Quantum Computer*. 2000. arXiv: quant-ph/0008033.

[52] Gregory D. Kahanamoku-Meyer and Norman Y. Yao. *Fast quantum integer multiplication with zero ancillas*. 2024. arXiv: 2403.18006 [quant-ph].

[53] Himanshu Thapliyal et al. "Quantum circuit designs of integer division optimizing T-count and T-depth". In: *IEEE Trans. Emerg. Top. Comput.* 9.2 (2019), pp. 1045–1056.

[54] Dirk Dalen. *Logic and Structure*. Springer Berlin, Heidelberg, 1994.

[55] drifter1. *compiler*. https://github.com/drifter1/compiler. 2020.

[56] Donald E. Knuth. "On the translation of languages from left to right". In: *Inf. Control* 8.6 (1965), pp. 607–639.

[57] Aditya Gupta. *C-Language-Parser*. https://github.com/SilverScar/C-Language-Parser/. 2016.

[58] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Vol. 55. US Government printing office, 1948.

[59] Michel Boyer et al. "Tight Bounds on Quantum Searching". In: *Fortschr. Phys.* 46.4–5 (1998), pp. 493–505.

[60] Christoph Durr and Peter Hoyer. *A Quantum Algorithm for Finding the Minimum*. 1999. arXiv: quant-ph/9607014.

[61] E. M. Stoudenmire and Xavier Waintal. "Opening the Black Box inside Grover's Algorithm". In: *Phys. Rev. X* 14.4 (2024).

[62] Bernhard Ömer. *QCL*. Version 0.6.7. 2022. URL: http://tph.tuwien.ac.at/~oemer/tgz/qcl-0.6.7.tgz.

[63] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum Random Access Memory". In: *Phys. Rev. Lett.* 100.16 (2008), p. 160501.

[64] Naixu Guo et al. *Quantum linear algebra is all you need for Transformer architectures*. 2024. arXiv: 2402.16714 [quant-ph].

List of Figures

List of Tables