RWTH AACHEN UNIVERSITY
Chair of Computer Science 2
Software Modeling and Verification

Master Thesis

Compilation of Quantum Programs with Control Flow Primitives in Superposition

Sascha Thiemann
Matr.-No.: 406187
Study Program: Computer Science M.Sc.
September 23, 2024

Supervisors: apl. Prof. Dr. Thomas Noll
Chair for Software Modeling and Verification
RWTH Aachen University

Prof. Dr. rer. nat. Dominique Unruh
Chair for Quantum Information Systems
RWTH Aachen University

Contents

1 Introduction **1**

2 Background **2**

2.1 Quantum Computing . . . 2
2.1.1 Superposition . . . 3
2.1.2 Entanglement . . . 4
2.1.3 Quantum Gates . . . 4
2.1.4 Measurement . . . 6
2.1.5 Relevant Algorithms . . . 6
2.1.6 Circuit optimization . . . 7
2.2 Quantum Control Flow . . . 9
2.2.1 Branching . . . 9
2.2.2 Iteration . . . 10
2.2.3 Limitations . . . 10
2.3 Quantum Languages . . . 11
2.3.1 Quantum Control Machine . . . 12
2.3.2 OpenQASM Language . . . 15
2.4 Compilation . . . 16
2.4.1 Lexer (Lexical Analysis) . . . 17
2.4.2 Parser (Syntax Analysis) . . . 17
2.4.3 Semantic Analysis . . . 19
2.4.4 Code Generation . . . 19
2.4.5 Optimization . . . 20
2.4.6 Tools . . . 21

3 Concept **22**

3.1 Language Overview . . . 22
3.1.1 Blocks and Scopes . . . 22
3.1.2 Data Types . . . 22
3.1.3 Basic Operations . . . 23
3.1.4 Control Flow . . . 23
3.1.5 Expressions . . . 23
3.1.6 Composite Gates . . . 23
3.2 Error Handling . . . 23
3.2.1 Warnings . . . 23
3.2.2 Critical Errors . . . 24

3.3 Optimization 24
3.3.1 Circuit Graph 24

4 Implementation **25**
4.1 Grammar 25
4.2 Semantic analysis 25
4.3 Code Generation 25
4.3.1 Expressions 25
4.3.2 Composite Gates 25
4.4 Optimization 25
4.4.1 Circuit Graph 26
4.4.2 Optimization Rules 26
4.4.3 Optimization Algorithm 27
4.5 Testing and Continuous Integration 27

5 Conclusion and Future Work **28**

References **28**

1 Introduction

2 Background

In the following section, we introduce and discuss different concepts that are referenced in later parts of this thesis. Firstly, we give a general introduction into quantum computing with some basic background knowledge on how quantum computers work, which quantum mechanical principles are essential for them, and some more specific knowledge about quantum algorithms and optimizations. Next, we discuss quantum control flow in more detail; it includes the formal definitions and its limitations. Then, we review existing quantum programming languages in general and some specific examples. Lastly, we give an overview on the topic of compilation and the different phases of a compiler.

2.1 Quantum Computing

While computers are prevalent and important in today's society, there are many relevant problems which classical computers can currently and perhaps will never realistically be able to solve. Quantum Computing (QC) is gaining more momentum as the technology that could solve at least some of these problems. For example, Quantum algorithms like Shor's algorithm [Shor97] could provide a significant improvement for prime factorization given sufficient technology. Therefore, it is estimated to be a valuable market with many of the largest technology companies as well as governments investing billions in the research and development of quantum technology [RDB*22, Pres18]. In the following section, we take a look at the basic concepts of a quantum computer and the core principles it relies on.

Classical Computers are based on simple operations executed on bits, like `and`, `or`, and `not`. These bits can either have a value of 0 or 1. Similarly, at their core, quantum computers apply simple operations, like `controlled not`, and `Hadamard`, on quantum bits (qubits). On a higher level, a classical computer executes operations on a register, consisting of multiple bits while a quantum computer operates on quantum registers, consisting of multiple qubits. In contrast to classical bits, quantum computers use the unique properties of quantum mechanics to enable qubits to have not just one value of either 0 or 1 but a combination of both. The phenomenon, where a particle or qubit exists in a combination of both states, is called *superposition*. Additionally, quantum computers also use the idea of *entanglement* to their advantage. Two qubits entangled when the value of one is dependent on the value of the other. The combination of superposition and entanglement enable quantum computers to solve specific problems more efficiently than classical computers [RDB*22].

Models for Quantum Computers can be divided into three main categories, the *analog model*, the *measurement-based model*, and the *gate-based model*. The analog model uses smooth operations to evolve a quantum system over time such that the resulting system encodes the desired result with high probability. It is not clear whether this model allows for universal quantum computation or quantum speedup [DiCh20b]. Instead of smoothly evolving a system, the measurement-based model starts with a fixed quantum state, the cluster-state. The computation is accomplished by measuring qubits of the system, possibly depending on the results of previous measurements. While there are different measurement-based models, one technique to apply gates is to leverage quantum teleportation, so called gate teleportation [Jozs05]. The result is a bit-string of the measurement results [DiCh20b, Niel06]. Lastly, the gate-based model uses a digitized, discrete set of qubits that are manipulated by a sequence of operations represented by quantum gates. The result is obtained by measuring the qubits at the end of the computation. Although digital quantum computation is more sensitive to noise than analog computations, the digitization can also be used for quantum error correction [DMN13] and to mitigate the increased noise [DiCh20b]. Furthermore, because qubits are actively manipulated and not passively evolved, digital quantum computers are more flexible than analog ones [RDB*22]. Therefore, the gate-based model is the most common model and this thesis will mainly focus on it.

Possible section on: no cloning/deleting [WoZu82, KuBr00]

2.1.1 Superposition

The first important property of quantum mechanics used by quantum computers is the idea of superposition. Qubits in superposition are often informally described as simultaneously having a value of 0 and 1 until their state is measured. However, a qubit in superposition is more formally a linear combination of its basis states. The basis states are the states where the qubit has a value of 0, written $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$, and 1, written $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ [DiCh20a]. Furthermore, the state can be reduced to a simple vector. Therefore, a state ψ in superposition can be written as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}.$$

The factors α and β are the amplitudes of the basis states and are complex numbers. The factors must also satisfy the condition $|\alpha|^2+|\beta|^2 = 1$. This is a result of the relation between the amplitudes and the probability to which basis state the state will collapse when measure, described in Sec. 2.1.4.

Beside $|0\rangle$ and $|1\rangle$, there exist more relevant short hands for quantum state. For example, $|+\rangle$ and $|-\rangle$ are states in uniform superposition, i.e. both basis state are equally likely, and often used when discussing quantum state und transformations. They are defined as follows:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \quad \text{and} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}.$$

2.1.2 Entanglement

Another important quantum mechanical concept is entanglement. Simply said, two qubits are entangled when their values depend on each other. An example would be a quantum system where two qubits are in superposition and equally likely to collapse to either 0 or 1; whichever value one qubit collapses to when measured, the second one will also collapse to the same values. Additionally, changes to one of the qubits can also affect the other one. This happens independent of the locations of the two qubits [RDB*22, HHHH09].

A more formal definition for an entangled state uses the definition of a composite system. Two separate quantum system can be represented as a single system with the tensor product of both systems. For example, the combined state $|\psi\rangle$ of the separate states $|0\rangle$ and $|1\rangle$ can be represented as:

$$|\psi\rangle = |0\rangle \otimes |1\rangle = |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

When a quantum state cannot be expressed as a tensor product of two states, the state is entangled. The previous example is a case of a maximally entanglement Bell state [DiCh20a, MHH19], often denoted β_{00}, and can be expressed as the following:

$$\beta_{00} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The entanglement of states is used by leveraging the effect of the qubits on each other to collaborate to calculate the result. Although this can be simulated on classical computers, it cannot be achieved "natively" because all classical bits are independent of each other. Moreover, quantum algorithms not using entangled states can often be simulated efficiently on classical computers [MHH19]. Therefore, entanglement is at the core of quantum computing but it can also have unintended consequences one needs to be aware of when designing quantum algorithms.

To calculate specific functions or intermediate values, quantum algorithms may need to use additional qubits or registers whose state can, in turn, be entangled with the main data of the algorithm. If this entanglement is not resolved in time by, e.g., uncomputing the changes to the qubit or register, it can interfere with future calculations or measurements and cause the results to be invalid. This effect is called *disruptive entanglement* [YVC24].

Uncomputing as a concept was not introduced before

Cannot find literature besides [YVC24] which calls this effect disruptive entanglement, use anyway?

2.1.3 Quantum Gates

In gate-based quantum computer, the transformations applied to the quantum data are represented by *quantum gates*. Similar to quantum states, which can be represented by linear combinations of basis states, or vectors, quantum gates can be formulated as linear transformations of these combinations, or a matrix. Because the result of such a transformation also needs to be a valid quantum state, the transformation needs to be norm-preserving, or *unitary* [DiCh20a]. The most relevant and often used unitary gates are depicted in Tab. 2.1

	Gates	Matrix	Ket-notation
Pauli gates	X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\|0\rangle \mapsto \|1\rangle$ $\|1\rangle \mapsto \|0\rangle$
	Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\|0\rangle \mapsto \|i\rangle$ $\|1\rangle \mapsto -\|i\rangle$
	Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto -\|1\rangle$
Hadamard gate	H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\|0\rangle \mapsto \|+\rangle$ $\|1\rangle \mapsto \|-\rangle$
Phase gate	$P(\lambda)$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto e^{i\lambda} \cdot \|1\rangle$
Controlled-NOT gate	CX	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	$\|00\rangle \mapsto \|00\rangle$ $\|01\rangle \mapsto \|01\rangle$ $\|10\rangle \mapsto \|11\rangle$ $\|11\rangle \mapsto \|10\rangle$
Toffoli gate	CCX	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	$\|000\rangle \mapsto \|000\rangle$ $\|001\rangle \mapsto \|001\rangle$ $\|010\rangle \mapsto \|010\rangle$ $\|011\rangle \mapsto \|011\rangle$ $\|100\rangle \mapsto \|100\rangle$ $\|101\rangle \mapsto \|101\rangle$ $\|110\rangle \mapsto \|111\rangle$ $\|111\rangle \mapsto \|110\rangle$

Table 2.1: List of relevant quantum gates in matrix representation as as functions in ket-notation.

A matrix U is unitary if it has an inverse matrix which is equal to its conjugate transpose $U^\dagger$, i.e. the following must hold:

$$UU^\dagger = I.$$

Therefore, all transformations applied to quantum states in a gate-based quantum computer must be reversible by definition. This limitation does not apply to classical computers where non-reversible transformations, e.g. mapping an arbitrary bit to a specific value, are easily implementable.

To design a useful quantum computer or language, the set of gates should be *universal*. A set of gates is universal if any gate can be simulated by a combination of the gates from the set with arbitrary accuracy [BrBr01]. An example for a universal set of gates is the combination of the Toffoli gate together with the Hadamard gate [DiCh20a].

Add paragraph on implicit measurement

2.1.4 Measurement

For quantum computer to be of any use, we need a way to read out information about its state. However, the information we can obtain from a quantum system is limited by the quantum measurement postulate. The postulate states that the only way, to gain any information from a quantum system, is to measure it. When measuring a quantum state, the state irreversibly collapses to one of its basis states. Furthermore, this is a probabilistic transformation and the original state in superposition cannot be recovered from the result. For a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the measurement collapses the state to $|0\rangle$ with a probability of $|\alpha|^2$. Correspondingly, the state will collapse to $|1\rangle$ with a probability of $|\beta|^2$ when measured [DiCh20a].

Measurement can be represented as a measurement basis set $\{M_i\}_i$ which requires the following condition:

$$\sum_i M_i^\dagger M_i = I.$$

The probability that outcome i is obtained when measuring a state $|\psi\rangle$ is equivalent to $|M_i|\psi\rangle|^2$. After the measurement of outcome i, the state $|\psi'\rangle$ will be equivalent to

$$|\psi'\rangle = \frac{M_i|\psi\rangle}{|M_i|\psi\rangle|} = \frac{M_i|\psi\rangle}{\sqrt{\Pr[\text{observe } i]}}.$$

In contrast to all other transformations, measurements are neither unitary nor reversible and, therefore, are able to "destroy" information on the quantum state before the measurement [DiCh20a].

2.1.5 Relevant Algorithms

Since quantum computers differ greatly from classical computer not only in their technology but also in the concepts they use for calculation, they cannot function without specially designed algorithms. These algorithms needs to exploit the special quantum properties of qubits to achieve *quantum advantage*, i.e. a better complexity than any classical algorithm. One of the first algorithms to show its quantum advantage was the Deutsch–Josza algorithm [DeJo92]. Deutsch et al. define a problem that can be solved in exponential time on classical computer and present a quantum algorithm which can solve the problem in polynomial time. The Bernstein-Vazirani algorithm [BeVa93] is another example with shown quantum advantage, resulting in a polynomial speed up. However, currently, there does not exist a use case for either of the algorithms and, therefore, they are only of limited theoretical interest [DiCh20c].

An algorithm with more potential for practical use is Shor's algorithm [Shor97]. It presents an more efficient, polynomial-time quantum implementation for the discrete logarithm, i.e. find r for a given a, x, p such that $a^r = x \mod p$. The algorithm is of special interest because Shor also provides a reduction of prime factorization to order finding; order finding is a special case of the discrete logarithm where $x = 1$. Modern cryptography is often based on the complexity of factoring large prime numbers, e.g. the commonly used RSA cryptosystem [RSA78]. Therefore, an advanced quantum computer could brake these systems with Shor's algorithm [MVZJ18]. Not only does this prospect provide a practical use-case for QC but it also results in the research field of *post-quantum cryptography* [BeLa17].

Another relevant algorithm or transformation is the quantum Fourier transform (QFT) [Copp02]. Beside being used as a subroutine in Shor's algorithm, it is also relevant for other algorithm, e.g. addition of quantum registers [Drap00]. Similar to the discrete Fourier transform [Wino78] which operates on vectors, the QFT_{2^n} operates on the quantum equivalent of vector, quantum registers, of size n. Registers of size n consist of n qubits. From the register, the QFT extracts the present periodic features. Then, other algorithms can use these features for their calculations.

2.1.6 Circuit optimization

Despite the expansive theoretical foundations for QC, the current state of the art for its technology is limited. However, the technology is nearing its first milestone towards useable quantum computers with the advent of prototypes with noisy intermediate-scale quantum (NISQ) technology [BFA22]. Nevertheless, the technology is still far away from fault-tolerant quantum computers and, by definition, limited in the number of available qubits. Furthermore, the gate count of NISQ era quantum computers is limited by the inherent noise which is increased with each additional transformation [Pres18]. Therefore, attributes such as the gate count of a quantum algorithm are an important metric for its utility. To improve the utility of an algorithm, its quantum circuit can be optimized with different techniques and rules.

There exist many kinds of optimization techniques for quantum circuits. They are mostly concerned with optimizing the gate count of quantum circuits with the use of peephole optimizations, as described in Sec. 2.4.5. These techniques can range from general rules [GaCh11, LBZ21], that can be applied to all quantum circuits, to hardware-specific optimizations [KMO*23]. Furthermore, machine learning based optimization frameworks for quantum circuits are also gaining popularity [FNML21, LPM*24, RLB*24].

The most simple general optimizations are so called *null gates* [GaCh11]. They are gate combinations or gates under specific conditions that are equivalent to the identity gate I. Therefore, any occurrence of such a null gate can be removed from the circuit. The most basic example for null gates is the double application of a self-inverse gate, i.e. a gate which is its own inverse. These include the H, X, Y, and Z gates. The identities are also depicted in Fig. 2.1. Furthermore, the same holds for any controlled version of a self-inverse gate such that the rule can also be applied to $CNOT$ and similar gates.

The second kind of null gates are gates that do not have an effect under specific conditions. For example, a controlled gate U is not applied if we know that the control

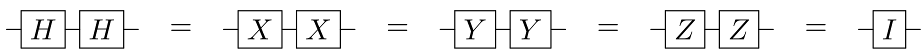

Figure 2.1: Null gates of self-inverse gates.

is $|0\rangle$. Similarly, the X gate does not have an effect on a qubit in the $|+\rangle$ state. Hence, the two circuits depicted in Fig. 2.2 are null gates and semantically equivalent to an identity gate.

Figure 2.2: Null gates for gate in specific conditions.

Another class of optimizations are called *control reversal*. Control reversal describes gate combination equalities based on the symmetry of the controlled Z gate. For the controlled Z gate, it is semantically equivalent to apply the Z gate to the second wire with a control on the first and to apply the Z gate on the first wire with a control on the second one. Based on this and together with the equalities $HZH = X$, and $HXH = Z$, a controlled X gate surrounded by H gates on both wires can be represented as the reversed X gate. Both equalities are depicted in Fig. 2.3 and Fig. 2.4.

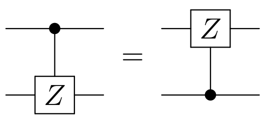

Figure 2.3: Control reversal of the controlled Z gate.

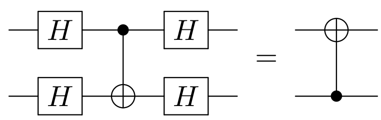

Figure 2.4: Control reversal of CX.

In contrast to general optimization rules, hardware-specific optimizations are mostly not concerned with the reduction of gates based on mathematically equal gate combinations; however, they exploit either the specific properties of the hardware for optimizations or replace gates with cheaper equivalents on the specific hardware. For example, a shuttling-based trapped-ion quantum computer operates by physically moving ions to segments in the hardware where operations can be applied. Since ions can and must be freely moved, swapping qubits can easily be accomplished by physically changing the position of their hardware equivalent. On the software side, this can be achieved by removing the swap-gate and swapping all following instances of both qubits. [KMO*23].

2.2 Quantum Control Flow

The idea of quantum control flow was first used by Altenkirch et al. [AlGr05] when defining a functional programming language with quantum control flow elements. The language uses an if-statement in superposition, `if°`, which is used to, e.g., defined the Hadamard gate as a function *had* instead of a matrix. The *had* function takes a qubit as an input. If the qubit is true, i.e. the value is one, the function returns a uniform superposition of true and false, where true has a negative sign. Correspondingly, for a false input, a uniform superposition with both signs positive is returned.

$$\begin{aligned} had &: Q \to Q \\ had &: x \mapsto \ \texttt{if}^{\circ} x \\ & \qquad\quad \texttt{then}\ \{false \mid -true\} \\ & \qquad\quad \texttt{else}\ \{false \mid true\} \end{aligned}$$

Quantum control flow can be divided into *quantum branching* and *iteration* [YVC24]. In the following, we will discuss both branching and iteration in superposition as well as the limitations of quantum control flow.

2.2.1 Branching

Based on the work presented by Altenkirch et al. [AlGr05], the concept of quantum control flow, more specifically quantum branching, was expanded on and formally defined by Ying et al. [YYF12]. They introduce two different types of quantum branching, quantum guarded commands, and quantum choices as a special case of guarded commands. The definition of quantum guarded commands is based on Dijkstra's guarded commands [Dijk75]. Guarded commands concern the nondeterministic execution of functions based on Boolean expressions, where the nondeterminism derives from the possible overlapping of the guards. In contrast, quantum branching allows for execution of functions based on a value in superposition. The functions are executed such that the result may be a superposition of the results of the individual functions [YVC24]. Quantum branching is, e.g., used in simulation algorithms like [BGB*18] and [LoCh19]. Furthermore, many basic concepts such as controlled gates can be represented as quantum branching or even single qubit gates as seen in the previous example of the Hadamard implementation.

The formal definition for classical guarded commands is given by:

$$\square_{i=1}^{n} b_i \to C_i$$

where C_i is a command guarded by a Boolean expression b_i. The command can only be executed if the expression is true. Similarly, quantum guarded commands map to a set of quantum programs P_i. Further, a set of qubits or quantum registers and a corresponding orthogonal basis $|i\rangle$ is given. However, the set of qubits guarding the program must be disjoint from the set of qubits used in the program. Without this condition, the resulting operation may not be unitary. For example, an X-gate that is executed if the wire it operates on is 1 always results in a values of 0; therefore, the operation can be neither reversible nor unitary. The resulting quantum guarded command is of the following form:

$$\square_{i=1}^{n} \bar{q}, |i\rangle \rightarrow P_i.$$

The quantum programs are guarded by the basis states and the control flow results from the superposition of these basis states [YYF12].

2.2.2 Iteration

Quantum iteration can be implemented either as quantum recursion or quantum loops. While some languages implement loops based on the measurement of qubits or registers [Ying11], the concept of quantum iteration requires the body of the loop to be executed in superposition based on a guard in superposition [YYF12].

While classical iteration takes an operation and repeats it on a classical register for k iterations, quantum iteration is dependent on a value k' in superposition and, correspondingly, returns a quantum register in superposition. Moreover, it is a special case of quantum branching and heavily restricted by the limitations of quantum computers [YVC24].

2.2.3 Limitations

While quantum control flow is often based on the corresponding control flow primitives on classical computers, it is restricted by multiple limitations imposed by quantum computers. Therefore, many control flow primitives that are used in classical programs can either by not used at all or in a limited capacity. There are two main limitations for quantum programs. Firstly, all gate-based quantum computers need to adhere to *reversibility*. Secondly, programs need to follow the *synchronization* principle for them to return any useful results [YVC24].

Reversibility

As introduced in Sec. 2.1.3, any sequence of instructions on gate-based quantum computers, excluding measurements, is required to be reversible by definition as they are all unitary transformations. Therefore, any quantum control flow is also required to adhere to this principle. A resulting limitation, that is not present on classical computer, is that any guards for guarded commands need to be immutable in the commands themselves. For example, if a qubit's state is flipped when its value is 0, the resulting command will always return value of 1. When a program returns the same result regardless of which statements where executed, the program cannot be reversible. This limitation is also inherit in the definition of quantum guarded commands, as described in Sec. 2.2.1. Moreover, control flow, as implemented in classical computers, is also not possible. At the most basic software level, modern computers use jump and conditional jump instructions to implement branching and loops. However, any classical jump instruction is inherently irreversible. Not only can a jump go to a section of code that is accessible without any jumps, multiple jumps can also lead to the same line of code. Therefore, a reversed program cannot know which path was taken in original program [YVC24].

A simple solution seems to be offered by the *Landauer Embedding* [Land61]. Fundamentally, the idea of the embedding is to turn a non-reversible function into a reversible one by not only returning the output but also the input of the function. For example, for a domain D and a codomain D', any non-reversible function $f : D \to D'$ can be given as a reversible function $g : D \to D' \times D$ with $g(x) = (f(x), x)$. In the case a quantum program with, e.g., jump instructions based on the Landauer embedding, the output would be the result of the program and a complete history of which path was taken through the program. However, because the quantum data depends on the program history, they become entangled. This leads to disruptive entanglement, as described in Sec. 2.1.2, causing invalid results [YVC24].

Synchronization

As previously discussed, reversibility alone is not the only limiting factor on quantum control flow. When handling control flow, similar to the classical implementation, with a program counter in superposition, the program counter can become entangled with the data and result in disruptive entanglement leading to an invalid result. To avoid this issue, the program must not only be reversible but also adhere the the principle of *synchronization*. It states that control flow must become independent from the data. Further, because any quantum program needs to be synchronized to return any useful results, while loops dependent on a value in superposition need to be bound by a classical value [YVC24].

2.3 Quantum Languages

With the emergence of quantum computing, many quantum languages were introduced. Most languages focus on a lower level representation of quantum circuits. An example is the popular Open Quantum Assembly Language (OpenQASM) [CBSG17]. OpenQASM consists mainly of quantum and classical registers that can be manipulated by predefined and composite gates. Additionally, some classical control flow is possible with if-statements depending on classical bits or measurements. As its name suggests, the language is designed for low level interactions with quantum computers and mostly used to directly describe a quantum circuit. In Sec. 2.3.2, OpenQASM is discussed in more detail.

In contrast to the low level circuit descriptions of OpenQASM, there are also languages with a focus on high level interactions. One such language is Tower [YuCa22]. It does not only allow for basic qubits and registers in superposition but also abstract data structures such as lists. Another example is the language Silq [BBGV20] which allows for the automatic and safe uncomputation of registers after they have been used for, e.g., intermediate calculations. What both languages have in common is the restriction to quantum data while using only classical control flow.

Although quantum control flow was formally defined by Ying et al. [YYF12], as described in Sec. 2.2, over ten years ago, only very few languages have incorporated the principle. One example is the functional programming language proposed by Altenkirch et al. [AlGr05] where quantum branching is used to define, e.g., the Hadamard gate. Only recently was the Quantum Control Machine with quantum control flow at its core proposed by Yuan et al. [YVC24]. It presents an instruction set similar to classical assembly languages but for quantum computers and discusses the resulting limitations for the language. In the following section, we will discuss the quantum control machine in more detail.

2.3.1 Quantum Control Machine

The Quantum Control Machine (QCM), proposed by Yuan et al. [YVC24], is an instruction set architecture that does not only allow for data in superposition but also quantum control flow. The architecture is designed around the limitations of control flow in superposition.

The syntax and logic of the QCM are both heavily influenced by classical assembly languages. Similar to classical computers, the language provides a finite set of quantum registers which are all initiated to a value of zero. The instruction set of the architecture does not only provide limited gate transformations and swap operations but also more classical operations on registers such as get-bit operations and simple arithmetical operations like addition and multiplication. However, what makes the QCM stand out are the jump instructions that enabled quantum control flow.

The gates of the architecture are limited to the X and Hadamard gate H. However, since the QCM machine enables quantum branching, any gate can become a controlled gate such that the X gate can easily be used in combination with quantum branching to create a Toffoli gate. Together with the Hadamard gate, the gate set is therefore universal, as described in Sec. 2.1.3.

There are three kinds of jump instructions. The first is a simple jump based on a given offset, the second is a conditional jump that performs a basic jump when a given register is 0, and, lastly, an indirect jump which is based on the value of a given register. Although the jump instructions are based on jumps in classical computers, they are limited by the restriction of unitary gates and must adhere to *reversibility* and *synchronization* [YVC24], as described in Sec. 2.2. An overview of some QCM instructions is depicted in Tab. 2.2.

When quantum computers are based on unitary gates, all their operations need to be unitary and, therefore, reversible as well. This limits quantum jump instructions and prohibits them to work like their classical equivalent. However, the problem of a reversible architecture and instruction set is not unique to quantum computers but was also taken into consideration for classical architecture to, e.g., increase energy efficiency

[1] After all operations, the instruction pointer is increased by the value of the branch control register.

Operation	Syntax	Semantics[1]
No-op	`nop`	Only increases instruction pointer by the branch control register.
Addition	`add` ra rb	Adds register rb to rb.
Multiplication	`mul` ra rb	Multiplies register ra by rb.
Jump	`jmp` p	Increases branch control register by p.
Conditional Jumps	`jz` p ra	Increases branch control register by p if ra is 0.
	`jne` p ra rb	Increases branch control register by p if ra is not equal to rb.

Table 2.2: An excerpt of the QCM instruction set with instructions used in later examples.

of classical computers [AGY07, TAG12]. To enable reversible jumps, the QCM adapts the *branch control register* from the reversible Bob architecture [TAG12]. Instead of directly changing the instruction pointer of the machine, the branch control register specifies how much the instruction pointer advances after each instruction.

The branch control register can then be manipulated reversibly by, e.g., adding or subtracting from it. To jump by a given *distance*, the branch control register needs to be increased to *distance*. However, after the instruction pointer has reached the desired location, the register needs to be decreased to its original value. Otherwise, the pointer would continue to jump in larger increments and any further jumps, i.e. modification to the branch control register, would not jump to the correct location. Since the jump instructions are defined to be reversible, the instruction set also includes a reverse jump instruction which instead decreases the branch control register by a given offset. Therefore, a jump instruction always requires a reverse jump instruction to reset the program counter. Similarly, other operations can also be represented as the reverse operation of an existing one. For example, subtraction can be implemented as reverse addition. Further, to make the code easier to read and write, the QCM also allows for named labels, which can be used for jump instructions instead of offsets. The offset to the given label can then be computed at compile time.

An example of a classical program and the reversible equivalent can be seen in Fig. 2.5 and Fig. 2.6 respectively. Both programs calculate x^y for two registers x and y. While the first example has classical jumps that are not reversible, the second example uses reversible jump instructions and their reverse counterpart to create a reversible algorithm.

Although such a program counter addresses the issue of reversibility, it can become entangled with data registers when in superposition. This can lead to disruptive entanglement where the output of the program becomes invalid [YVC24]. To prevent any disruptive entanglement of the data and control registers, QCM programs must adhere to the principle of synchronization, as described in Sec. 2.2. It requires that the control flow is separated from the data at the end of execution. However, this is

```
    add   res $1
    add   r1  y
l1: jz    l2  r1
    mul   res x
    radd  r1  $1
    jmp   l1
l2: nop
```

Figure 2.5: A non-reversible exponentiation algorithm.

```
     add   res $1
     add   r1  y
l1:  rjne  rl1 r1  y
rl2: jz    l2  r1
     mul   res x
     radd  r1  $1
rl1: jmp   l1
l2:  rjmp  l2
```

Figure 2.6: Reversible exponentiation algorithm.

not the case for the reversible example program in Fig. 2.6 which, therefore, is not a valid QCM program.

The issue that the loop in the reversible example encounters is the *tortoise and hare* problem. Given a superposition of two different values a and b in the y register, the loop will execute a and b times respectively. Therefore, one of the two loops will finish before the other. Since we must adhere to synchronization, the instruction pointer needs to become independent of the two values again. However, because the branch with the faster execution of the loop cannot simple wait, the other branch cannot catch up and the instruction pointer cannot become independent of the data values. Consequently, the program does not adhere to synchronization. To prevent this issue, the program must include padding operations which are executed instead of the main loop. Furthermore, the loop also needs to be bounded by a classical value, as described in Sec. 2.2.3. The results in an algorithms, as depicted in Fig. 2.7, that calculates $x^{\min(y, max)}$. Here, *max* is a classical bound to the number of loop iterations, as required.

```
       add   res  $1
       add   r1   max
l1:    rjne  rl1  r1   max
rl2:   jz    l2   r1
rl3:   jg    l3   r1   y
       mul   res  x
rl4:   jmp   l4
l3:    rjmp  rl3
       nop                  ; padding
l4:    rjle  rl4  r1   y
       radd  r1   $1
rl1:   jmp   l1
l2:    rjmp  rl2
```

Figure 2.7: A synchronized, reversible exponentiation algorithm.

2.3.2 OpenQASM Language

The Open Quantum Assembly Language (OpenQASM) 3 [CJA*22] is the successor of the OpenQASM 2 [CBSG17] language. Both languages are imperative and machine independent quantum languages. They are low level quantum languages and, thereby, concretely describe a quantum algorithm in the form of a circuit. OpenQASM 2 developed into a de facto standard and is often used as an intermediate for different quantum tools [CJA*22]. OpenQASM 3 was developed to fit the changing needs of current quantum research and hardware while being mostly backwards compatible except for some uncommon cases. For example, some keywords were added of changed for the successor such that identifiers of OpenQASM 2 circuits by be invalid in the successor language. Sine OpenQASM 3 is the new and improved standard, we will focus on its features in the following section.

OpenQASM 3 requires the header to indicate the language in the circuit header for any top-level circuit. This is achieve by adding `"OpenQASM 3.0";` to the beginning. Additionally, the language supports the inclusion of other source files which can be include with the `include` keyword.

Similar to other quantum languages, OpenQASM operates in 2 basic data types. The first is the classical bit while the second is the qubit. Both primitives can also be used in registers with a fixed size. Additionally, OpenQASM 3 also supports further classical data types such as angles and signed and unsigned integers. In contrast to its predecessor where any identifiers have to start with a lowercase letter, in OpenQASM 3, identifiers can start with a range of unicode characters with some exception.

The basic operations of the language can be divided into unitary and non-unitary operations. In OpenQASM 3, all unitary operation are based on the unitary $U(a, b, c)$ where a, b, c are angular parameters. While OpenQASM 2 supported a controlled-NOT gate natively, the successor requires the gate to be defined with, e.g., the NOT gate and a control modifier. The control modifier can be used to turn any arbitrary unitary gate into a controlled gate with an arbitrary number of control qubits. Therefore, the formally predefined gate CX must now be defined by the programmer of represented by a NOT gate with a control modifier, e.g. `ctrl @ x`. Lastly, the non-unitary operations are `measure` and `reset`. While the `measure` operation measures the state of a qubit and saves it to a classical bit, the `reset` operation discards the value of a qubit and replaces it with the $|0\rangle$ state.

The programmer can not only use the operations and modifiers provided by OpenQASM 3 but can also define custom gates. These user-defined gate are defined with an identifier for the gate and a fixed number of single qubit arguments and angular parameters. In the body of the gate definition, the user can apply a sequence of gates to the qubit arguments with the given angular parameters. Additionally, the language also provides implicit iteration. This means that the application of a single qubit gate to a quantum register will be interpreted as separate applications of the gate to all qubits in the register.

In Fig. 2.8, a example circuit, written in OpenQASM 3, is depicted. The circuits takes two qubits, brings them into an entangled superposition, measures their state

```
"OpenQASM 3.0"; /* Indicate language in circuit header. */

gate x a { U(pi,0,pi) a; }       /* Define x  gate. */
gate cx a, b { ctrl @ x a, b; } /* Define cx gate. */
gate h a { U(pi/2, 0, pi) a; }  /* Define h  gate. */

qubit[2] reg; /* Definition of quantum register. */
bit[2] res; /* Definition of resical register. */

h reg[0]; /* Apply h gate to fist qubit in register. */
cx reg[0], reg[1]; /* Apply cx gate to the qubits. */

res[0] = measure reg[0]; /* Measure qubit and save to bit. */
res[1] = measure reg[1]; /* Measure qubit and save to bit. */
```

Figure 2.8: Code for an OpenQASM 3 example circuit.

and saves the result to a classical register. In the beginning of the circuit definition, the circuit header indicates the language and the X, CX, and H gate are defined based on the predefined unitary U. Then, the quantum and classical register, both with a size of 2, are defined. Next, the Hadamard gate H is applied to the first qubit in the quantum register followed with the application of a controlled-NOT gate to both qubits. Lastly, the state of both qubits is measure and the result is saved to the classical bits.

2.4 Compilation

The execution on a computer is controlled by a program. This program is written in a specific language unique to the hardware of the computer, machine code. However, this language is often neither human readable nor suitable for writing complex systems. Therefore, most programs are written in a more accessible language. The program can then be translated to the machine code with a *compiler*.

A compiler translates a program written in a source language to a program in a target language. The compilation process can be divided into multiple steps. The first step is the *lexical analysis* to transform the source code into a sequence of tokens. Next, the syntactic structure of the code is analyzed be the *parser*. Then, the code is *semantically analyzed* to find semantic errors and infer information for the following phases. Lastly, the *code generation* step generates the code in the target language. Additionally, the compiler may perform optimizations on the code before generating the target code or it may *optimize* the resulting target code [Oliv07, VSSD07]. In the following, we be discuss the different steps of a compiler individually.

2.4.1 Lexer (Lexical Analysis)

The lexical analysis of the source program takes the character stream and groups together associated characters producing a sequence of tokens [Oliv07]. Therefore, the step is also referred to as *tokenization* [Gref99]. The process can be divided into the *scanning* and *screening* of the character and token sequence [DeRe74].

The scanning process groups together substrings into textual elements, or tokens. In contrast to the characters and substrings, these tokens have defined meanings and may have additional attributes. For example, they may include identifiers, operator, comments, and spaces. In the case of the identifier token, an additional attribute could be the string value of the identifier. They can be specified with the help of a regular grammar or regular expression [DeRe74, VSSD07].

[VSSD07] is extensive book, cite specific chapter somehow?

After being divided into a sequence of tokens, the screening step drops any characters or sequences of characters not relevant to the compilation from the program code. These may include characters such as spaces and tabs, or white space in general, and character sequences such as comments. Further, is may also recognize additional special symbols, such as keywords, and map them to a designated token. For example, a identifier with a value of “while” could be mapped to the corresponding token of the `while`-toke.[DeRe74].

Some example regular expressions for a lexical analysis are depicted in Fig. 2.9. The code depicts regular expressions for integers, identifiers, comments, and white space in ANTLR syntax. The integer can either be an arbitrary sequence of characters between zero and nine without a leading zero or just zero with a length of at least one. Similarly, an identifier is a sequence of lower and upper case alphabetical characters, numbers, and underscores with a length of at least 1 and without a leading number. In contrast, a comment is any string starting with a double slash until the line break and white space is any white space characters. Additionally, the comment and white space also define a scanning step where both are discarded.

add reference to section discussing ANTLR

```
INTEGER    : [1-9] [0-9]* | '0' ;

IDENTIFIER : [a-zA-Z_] [a-zA-Z_0-9]*;

COMMENT    : '//' ~[\r\n]*    -> skip;

SPACE      : [ \t\r\n\u000C] -> skip;
```

Figure 2.9: An example of a regular grammar for the lexical analysis.

2.4.2 Parser (Syntax Analysis)

The lexical analysis of the compiler yields a sequence of tokens with a known meaning; the structure of the program, however, is not apparent in the token sequence. For example, an operator-token does not indicate what the operands are. To gain knowledge of the structure of the program, the parser step analyzes the syntactic structure of the source program and creates a parse tree from it. The compiler can then use the tree by, e.g., walking over it to generate the target code. This step should also detect and report any syntactical errors, like a missing closing parentheses [VSSD07].

While the lexical analysis can be achieved with regular expressions, the syntactic structure of a program must be represented by, at least, a context-free grammar. Since regular languages are a subset of context-free languages, the parsing step can also perform the lexical analysis. However, there are multiple reasons why the lexical and syntax analysis are separated. Firstly, the separation of both analysis makes the compiler more modular and extensible. Furthermore, using regular expression for the lexical analysis prevents it from being more complex than necessary with a context-free grammar. Lastly, the lexer can be more efficient when generated from regular expressions instead of a context-free grammar [VSSD07].

citation needed? (Chomsky, "Three models for the description of language")

There exists two main kinds of parsing a grammar, either top-down or bottom-up. Top-down parsing creates a parse tree based on an input sequence of tokens starting from the root and creating the nodes in a depth-first approach. It yields a left-most derivation for the input sequence and can be implemented as a recursive-descent parser. The most common form of top-down parsing is *LL*-parsing, where the input is read from *l*eft to right, yielding a *l*eftmost derivation. To improve the efficiency of parsers, the context-free grammar is often restricted such that it can be parsed without backtracking with a fixed length *lookahead* onto the token sequence. Such grammar are called $LL(k)$ grammars where k is the length of the lookahead [VSSD07, PaFi11].

In contrast, bottom-up parsing builds the parse tree from the leaves up to the root. Furthermore, instead of yielding a left-most derivation, it produces a right-most derivation. Similar to top-down parsing, the most common bottom-up parsers scan the input from left to right which are therefore LR-parsers. Moreover, they can also be implemented more efficiently when restricting the grammar to a maximum lookahead. These grammar the $LR(k)$ grammars [VSSD07, PaFi11].

An example grammar for parsing simple integer expressions is depicted in Fig. 2.10. Similar to the regular expressions in Fig. 2.9, the grammar is given in ANTLR syntax. An expression is either the sum of another expression and a term or just a term. In turn, a term is either the product of a term and a factor or just a factor. Lastly, a factor is either an expression in parenthesis or an integer. Here, the definition of an integer is omitted. However, it can be seen in the previous example. The grammar is defined such that a generated parse tree inherently adheres to the order of operations.

```
exp    : exp '+' term    | term;

term   : term '*' factor | factor;

factor : '(' exp ')'     | INTEGER;
```

Figure 2.10: An example of a context-free grammar for parsing simple expressions.

2.4.3 Semantic Analysis

The parser analyzes the syntactic structure of a program with a context-free grammar; however, an analysis without any context is not sufficient for an analysis of non-syntactic, i.e. semantic, constraints of the program. This step is performed by the semantic analysis. The semantic analysis is used to throw semantic errors that may prevent the program from being compiled such as the use of undefined identifiers. Further, it may also enforce constraints that prevent runtime error such as type checking in a strongly typed language. Additionally, the analysis step may also process and save declarations and similar information to a symbol table which can be used in the code generation or optimization [Oliv07, SWW*88]. Moreover, the semantic analysis may not only throw errors but can also be used to infer additional information for further compilation steps. For example, besides preventing operations on operands with invalid types, the analysis may deduce which operation to apply to the operands based on their type; in the case of two integers, the analysis may infer an integer additions for the "+"-operator while two floating point values require floating point operations [Wait74, VSSD07].

What specifically the semantic analysis does is dependent on the design of the language being analyze. For example, a loosely typed language may have limited type checking, when compared to a strongly typed language, if any at all. Further, the implementation of the analysis can differ greatly. However, all implementations have some common elements. It requires the propagation of attributes through the syntactic structure of the program to enable the analysis. In the case of type checking, the analysis must pass on the type of a variable. Moreover, it does not only need to know the types of variables and constants, i.e. leafs in a parse tree, but also the resulting type of an expression using them. For example, a integer added to a floating point value may result in a floating point value. To infer and propagate these attributes, the parse tree may need to be transverse [Wait74, VSSD07].

2.4.4 Code Generation

After the semantic analysis of the program, which, at this stage, is in the form of a parse tree, the compiler can generate the code. Here, the compiler can either generate the target code, e.g. machine code, directly or translate the parse tree into an intermediate code. The translation of the source code to the intermediate can be thought of as the *frontend* of the compiler, with the translation of the intermediate to the target being the *backend*. While the intermediate code will need to be translated again into the target language, the use of an intermediate representation can increase the modularity and extensibility of a compiler. Additionally, it can also ease the construction of a new compiler. When creating a new compiler from a source language to a target language the front end of an existing compiler for the source can be combined with an existing compiler to the target if both are using the same intermediate language [VSSD07, GFH82].

em dash (—) here?

The most common issues when generation the target code are the evaluation order of expressions, register and storage allocation as well as related issues, context switches, and instruction selection [GFH82]. While these issues are critical for compilers that translate classical languages, i.e. not quantum languages, to machine code, they are mostly not relevant for the translation of quantum computers, since quantum computers do not offer same features and abstractions that classical computer do; they have, e.g., no storage, other than the quantum registers. Therefore, we will not discuss these issues in more detail.

citation needed?

2.4.5 Optimization

While the lexical, syntax, and semantic analysis combined with the code generation are the essential parts of a compiler, without which it would not work, the optimization step is also important. It used to apply either machine-independent or machine-dependent optimizations. The optimizations can be applied to the parse tree, a possible intermediate representation, and the generated target code depending on the optimization itself. While the removal of unreachable code, e.g. code after a return statement, can most easily be performed on the parse tree, machine-dependent optimizations can, more appropriately, be performed on the target or intermediate code [Oliv07, VSSD07].

Two collaborating machine-independent optimizations that are often applied by compilers are constant propagation and constant folding. Constant propagation analyzes the code to find variables with constant values throughout all executions and replaces the variables in, e.g., expressions with their corresponding constant value. By itself, constant propagation may only result in marginal improvement, loading a constant literal instead of the values of a variable; however, in combination with constant folding it can result significant improvement. Constant folding evaluates expressions or subexpressions with constant values at compile time, resulting in less calculations at runtime. This can significantly increase the performance of a program especially if large expressions or expressions in loops can be folded. Propagating constant values through the code enables more constant folding and, therefore, can improve its effectiveness [WeZa91].

Another optimization technique, that can work in tandem with constant folding and propagation, is loop unrolling. When executing a loop, each iteration needs to check the halting condition and possible execute an increment statement which can result in significant overhead. Futhermore, since the condition is checked before each iteration, the different executions cannot be executed in parallel. To prevent or reduce the performance overhead from these issues, the loop body can be executed multiple times and the increment statement adjusted accordingly. Further, if the number of iterations is constant, the loop can be removed entirely and replaced by the repeating loop body [HuLe99]. In this case, constant propagation and unrolling can help evaluate the halting condition such that the loop can be unrolled.

- Function inlining [BeDa94]
- Peephole optimization [McKe65]

2.4.6 Tools

- Why use tools
- [PaFi11] states: Parsing is not a solved problem, despite its importance and long history of academic study. Because it is tedious and error-prone to write parsers by hand, researchers have spent decades studying how to generate efficient parsers from high- level grammars
- What are different existing tools?
- flex/bison
- yacc
- ANTLR
- ...

3 Concept

- What are the different aspects of the compiler
- How are they designed
- How do the work
- What is the reason for their implementation...

3.1 Language Overview

- Given an overview of the different feature of the language
- How do they work and what is the reason for implementing them
- Why are some feature (e.g. implicit iteration) *not* implemented

3.1.1 Blocks and Scopes

- Basic structure of Luie
- Consists of blocks and statements
- One main block
- Symbol table that handles scopes
- All blocks have scope

3.1.2 Data Types

- Different data types
  - Register
  - Qubits (Registers with size 1)
  - Iterators, in more detail in Sec. 3.1.4

3.1.3 Basic Operations

3.1.4 Control Flow

3.1.5 Expressions

- Consists of expressions, terms and factors
  - Expressions consist of expression, operator, and term or just a term
  - Term consists of term, operator, and factor or just a factor
  - Factor consists of expression in parentheses, a negated factor, number, identifier or function call
- Inherent order of operations

3.1.6 Composite Gates

3.2 Error Handling

Generally, an important part of a program is error handling; useful and precise error messages are essential for comfortable interactions with the program. This is especially the case for compilers where the user should not only easily understand what the issue is but also where in the source code the error occurred.

Our compiler has two types of errors with different severity. The first type is the warning. A warning from the compiler can indicate issues in the source code that may cause unintended behavior. However, the issues itself does not prevent the compilation of the program and is simply an indication that there my be something wrong. In contrast, the critical error is cause by a flaw in the source program that prevents the correct compilation. In the following, we will discuss the different warnings and critical errors, the compiler my raise and what their meaning is.

3.2.1 Warnings

The compiler can throw two different kinds of warnings. The first is the invalid range warning and the second is the unused symbol warning.

An invalid range warning can occur in the context of for loops. They iterate over a range that is defined by the user. It can be given as either a size n and iterate from 0 to $n-1$ or a start and end index, i_{Start} and i_{End} respectively, and iterate from the start to the end. However, the range iterator is designed to only increase. Therefore a range where $i_{Start} \geq i_{End}$ is invalid. Since the for loop is unrolled at compile time, a range with a size less than or equal to zero can just be ignored. This however may be unintended behavior. Therefore, the compiler warns the user that the range is invalid.

The unused symbol warning is raised when a symbol, e.g. a register of composite gate, is defined in the source code but never used. The unused symbol does not have any negative effect on the compilation and the optimization step can easily remove, e.g., an unused register. Therefore, this is only a warning and the program an be compiled. However, an unused symbol may indicate, that the wrong symbol was used somewhere else or part of the program is no longer used, hence, warning the user of the unused symbol may prevent untended behavior.

3.2.2 Critical Errors

3.3 Optimization

- Describe circuit graphs
- Give formal definition
- Example graph

3.3.1 Circuit Graph

4 Implementation

4.1 Grammar

- Lexing and parsing implemented as ANTLR grammar
- Describe grammar structure
- Different elements of grammar

4.2 Semantic analysis

- What is semantic analysis used for?
- How is it implemented in Luie?
- Different types of semantic analysis
- Errors
  - Types of errors: Critical, warning
  - Different critical errors (Type, undefined, ...)
  - Different warnings (invalid range, ...)

4.3 Code Generation

- How is code generated?
- Important classes and abstractions

4.3.1 Expressions

4.3.2 Composite Gates

4.4 Optimization

The implementation of the compiler does not only translate the custom language to OpenQASM 3 but and allows for optimization on the translated circuit. To apply the optimization to the translated circuit, the circuit description, i.e. the program, is used to build a circuit graph, as described in Sec. 3.3.1. Next, an algorithm iterates over the graph and checks whether a list of optimization rules is applicable to a part of the graph. If a rule is applicable, the rule is applied. The process of iterating over the entire graph is repeated for as long as rules where applied in the previous iteration over the graph. When the optimization of the circuit is completed, the graph is translated back to a programmatic description of the circuit and the result is returned.

In the following, we will discuss the implementation of the different steps in the optimization process. This includes the circuit graph in general, the construction of the graph based on the program, and the translation of the graph back to a circuit. Further, we discuss the implementation of the optimization rules and the optimization algorithm in general.

4.4.1 Circuit Graph

- Basic structure of circuit
- Graph construction
- Graph translation
- Some auxillary constructs ands functions
  - Paths
  - Removal of nodes
  - replacing paths

Graph construction
Graph translation

- Important attributes
  - Gates applied in correct order
  - But when to apply which gate?
  - In many cases arbitrary (example)
- Eager translation
  - Translate each wire as much as possible
  - Switch to other wire only if entirely translated
  - or node can only be translated if other wire is translated first (up to the node)

4.4.2 Optimization Rules

- Rule interface
- Abstract optimization rule
- Describe the different optimization and the general implementation

4.4.3 Optimization Algorithm

- How is the graph iterated
- How are sub-paths used and created

4.5 Testing and Continuous Integration

- Different test categories
- How are they implemented?
- What do they test?
- (Continuous integration)

5 Conclusion and Future Work

- Conclusion to thesis
- Future work
  - how could language be extended

Bibliography

[AlGr05] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258. IEEE, 2005.

[AGY07] Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible machine code and its abstract processor architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science – Theory and Applications*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[BrBr01] Jean-Luc Brylinski and Ranee Brylinski. Universal quantum gates.

[BBGV20] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, New York, NY, USA, 2020. ACM.

[BeDa94] Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In Gerhard Goos, Juris Hartmanis, and Jürg Gutknecht, editors, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 105–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[BFA22] Medina Bandic, Sebastian Feld, and Carmen G. Almudever. Full-stack quantum computing systems in the nisq era: algorithm-driven and hardware-aware compilation techniques. In Cristiana Bolchini, editor, *Proceedings of the 2022 Conference et Exhibition on Design, Automation et Test in Europe*, ACM Conferences, pages 1–6, Leuven,Belgium, 2022. European Design and Automation Association.

[BGB*18] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. Encoding electronic spectra in quantum circuits with linear t complexity. *Physical Review X*, 8(4), 2018.

[BeLa17] Daniel J. Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[BeVa93] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In Rao Kosaraju, David Johnson, and Alok Aggarwal, editors, *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing - STOC '93*, pages 11–20, New York, New York, USA, 1993. ACM Press.

[CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language.

[CJA*22] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[Copp02] D. Coppersmith. An approximate fourier transform useful in quantum factoring.

[DiCh20b] Yongshan Ding and Frederic T. Chong. Introduction. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 3–12. Springer International Publishing, Cham, 2020.

[DiCh20c] Yongshan Ding and Frederic T. Chong. Quantum application design. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 55–70. Springer International Publishing, Cham, 2020.

[DiCh20a] Yongshan Ding and Frederic T. Chong. Think quantumly about computing. In Yongshan Ding and Frederic T. Chong, editors, *Quantum Computer Systems*, Synthesis Lectures on Computer Architecture, pages 13–54. Springer International Publishing, Cham, 2020.

[DeRe74] Franklin L. DeRemer. Lexical analysis. In F. L. Bauer, F. L. de Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, W. M. Waite, and J. Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 109–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974.

[Dijk75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DeJo92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[DMN13] Simon J. Devitt, William J. Munro, and Kae Nemoto. Quantum error correction for beginners. *Reports on progress in physics. Physical Society (Great Britain)*, 76(7):076001, 2013.

[Oliv07] José de Oliveira Guimarães. Learning compiler construction by examples. *ACM SIGCSE Bulletin*, 39(4):70–74, 2007.

[Drap00] Thomas G. Draper. Addition on a quantum computer.

[FNML21] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit optimization with deep reinforcement learning.

[GaCh11] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. Equivalent quantum circuits.

[GFH82] Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 14(4):573–592, 1982.

[Gref99] Gregory Grefenstette. Tokenization. In Nancy Ide, Jean Véronis, and Hans van Halteren, editors, *Syntactic Wordclass Tagging*, volume 9 of *Text, Speech and Language Technology*, pages 117–133. Springer Netherlands, Dordrecht, 1999.

[HHHH09] Ryszard Horodecki, Paweł Horodecki, Michał Horodecki, and Karol Horodecki. Quantum entanglement. *Reviews of Modern Physics*, 81(2):865–942, 2009.

[HuLe99] J. C. Huang and T. Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, pages 244–248. IEEE Comput. Soc, 1999.

[Jozs05] Richard Jozsa. An introduction to measurement based quantum computation.

[KuBr00] Arun Kumar Pati and Samuel L. Braunstein. Impossibility of deleting an unknown quantum state. *Nature*, 404(6774):164–165, 2000.

[KMO*23] Fabian Kreppel, Christian Melzer, Diego Olvera Millán, Janis Wagner, Janine Hilder, Ulrich Poschinger, Ferdinand Schmidt-Kaler, and André Brinkmann. Quantum circuit compiler for a shuttling-based trapped-ion quantum computer. *Quantum*, 7:1176, 2023.

[Land61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[LBZ21] Ji Liu, Luciano Bello, and Huiyang Zhou. Relaxed peephole optimization: A novel compiler optimization for quantum circuits. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 301–314, [S.l.], 2021. IEEE Press.

[LoCh19] Guang Hao Low and Isaac L. Chuang. Hamiltonian simulation by qubitization. *Quantum*, 3:163, 2019.

[LPM*24] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. Quarl: A learning-based quantum circuit optimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):555–582, 2024.

[McKe65] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[MHH19] Gary J. Mooney, Charles D. Hill, and Lloyd C. L. Hollenberg. Entanglement in a 20-qubit superconducting quantum computer. *Scientific reports*, 9(1):13465, 2019.

[MVZJ18] Vasileios Mavroeidis, Kamer Vishi, Mateusz D. Zych, and Audun Jøsang. The impact of quantum computing on present cryptography. 2018.

[Niel06] Michael A. Nielsen. Cluster-state quantum computation. *Reports on Mathematical Physics*, 57(1):147–161, 2006.

[PaFi11] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. *ACM SIGPLAN Notices*, 46(6):425–436, 2011.

[Pres18] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[RDB*22] Roman Rietsche, Christian Dremel, Samuel Bosch, Léa Steinacker, Miriam Meckel, and Jan-Marco Leimeister. Quantum computing. *Electronic Markets*, 32(4):2525–2536, 2022.

[RLB*24] Francisco J. R. Ruiz, Tuomas Laakkonen, Johannes Bausch, Matej Balog, Mohammadamin Barekatain, Francisco J. H. Heras, Alexander Novikov, Nathan Fitzpatrick, Bernardino Romera-Paredes, John van de Wetering, Alhussein Fawzi, Konstantinos Meichanetzidis, and Pushmeet Kohli. Quantum circuit optimization with alphatensor.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Shor97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[SWW*88] V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In R. L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, pages 233–240, New York, NY, USA, 1988. ACM.

[TAG12] Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück. A reversible processor architecture and its reversible logic design. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Alexis de Vos, and Robert Wille, editors, *Reversible Computation*, volume 7165 of *Lecture Notes in Computer Science*, pages 30–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[VSSD07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, techniques, & tools.* Pearson/Addison Wesley, Boston, 2nd ed. edition, 2007.

[Wait74] W. M. Waite. Semantic analysis. In F. L. Bauer, F. L. de Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, W. M. Waite, and J. Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 157–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974.

[Wino78] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.

[WoZu82] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982.

[WeZa91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[YuCa22] Charles Yuan and Michael Carbin. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, 2022.

[Ying11] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems*, 33(6):1–49, 2011.

[YVC24] Charles Yuan, Agnes Villanyi, and Michael Carbin. Quantum control machine: The limits of control flow in quantum programming. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1–28, 2024.

[YYF12] Mingsheng Ying, Nengkun Yu, and Yuan Feng. Defining quantum control flow.

List of Figures

2.1 Null gates of self-inverse gates. 8
2.2 Null gates for gate in specific conditions. 8
2.3 Control reversal of the controlled Z gate. 8
2.4 Control reversal of CX. 8
2.5 A non-reversible exponentiation algorithm. 14
2.6 Reversible exponentiation algorithm. 14
2.7 A synchronized, reversible exponentiation algorithm. 14
2.8 Code for an OpenQASM 3 example circuit. 16
2.9 An example of a regular grammar for the lexical analysis. 17
2.10 An example of a context-free grammar for parsing simple expressions. 18