Formal Verification of Quantum Stabiliser Codes By Stabiliser Formalism

by

Qiuyi Feng

Student Number: 1431319

Supervised by Dr. Christine Rizkallah and Prof. Udaya Parampalli.

A research thesis submitted in fulfilment for the degree of

Master of Computer Science

in the

Department of Computing and Information Systems

THE UNIVERSITY OF MELBOURNE

2025

Abstract

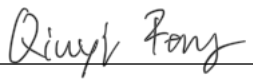
Quantum information is prone to errors. They might disrupt quantum computing processes, rendering the result meaningless. *Quantum error correction* (QEC) theory can mitigate this by encoding quantum information into *quantum codes*. Quantum codes are coding schemes that detect or correct errors. Hence, the correctness of quantum codes is crucial for a trustworthy foundation of fault-tolerant quantum computing. However, verifying quantum error correction codes is challenging and under-explored. A major obstacle in existing verification efforts is the exponential growth of the quantum state space, which hinders scalability.

To efficiently verify quantum codes, we formalise a theoretical framework of quantum error correction – the quantum stabiliser code formalism – in the proof assistant Coq. This development, named Coq-QECC, supports formal reasoning and verification of key program properties of quantum codes. We demonstrate the utility of Coq-QECC through case studies on a three-qubit quantum code and a nine-qubit quantum code. We verify certain key program properties of interest including detectable and correctable errors. Compared to existing efforts, our verification is more principled and reusable. This is achieved by building on an established algebraic reasoning framework, enabling higher-level automation and abstraction in the verification process. In addition, by leveraging algebraic methods, we partially address the scalability challenges and are able to verify program properties that remain unverified in prior work.

Declaration

I, Qiuyi Feng, declare that this thesis and the work presented in it are my own. I confirm that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- This thesis did not require clearance from the University's ethics committee.
- The thesis is approximately 25917 words in length (excluding text in figures, tables and bibliographies).

Signed: 

Date: 09 Jun 2025

Acknowledgement

I would like to thank my supervisors, Dr. Christine Rizkallah and Prof. Udaya Parampalli for guiding me through this research project. Your support and timely advice have steered me away from many wrong turns. And I'm Particularly thankful for you to support me to present my work-in-progress on CoqPL'25 workshop.

I am also truly grateful to my fellow students who offer me help. Thanks to Zoe Orlanda Elwyn for helping me understand MathComp, and to Dr. Peiyong Wang from CSIRO for your valuable insights into the theoretical aspects of quantum computing.

I would also express my gratitude to A.Prof. Robert Rand from the University of Chicago for providing key insights on SQIR, QuantumLib, and formal verification of quantum programs in general.

And I appreciate audience at the CoqPL'25 workshop for their constructive feedback, which helped shape the later stages of this work.

Lastly, I am deeply thankful to my family and friends for their unwavering emotional support throughout this journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Quantum computing promises to solve certain complex problems exponentially faster than classical computers [2]. Various quantum programming languages (QPLs) have been proposed to support quantum programming [23]. However, quantum programming is hard to be made correct [39]. Beyond the counter-intuitive quantum mechanics that easily confuse programmers, quantum programs are also difficult to test and debug due to their fundamental nature [33]. For example, quantum programs are inherently non-deterministic and probabilistic. This property makes straightforward example-based tests inapplicable, that is, tests by matching fixed and repeatable outputs for given inputs. Therefore, *formal verification* becomes more critical in quantum programming than in classical settings [8].

This thesis investigates the verification techniques for quantum programs. In particular, we focus on *quantum error correction (QEC)* codes [46] (abbreviated as *quantum codes*). Quantum codes can detect or correct certain errors, thus protecting quantum information from being corrupted. They are thought to be the key to achieving *fault-tolerant quantum computing*, the next milestone in the field [42]. Like other quantum programs, quantum codes need to be carefully implemented through programming. Hence, verifying them provides a trustworthy foundation in fault-tolerant quantum computing.

Particularly, we want to know can we find a rigorous yet efficient method to verify quantum error correction codes? Inspired by the pen-and-paper framework *stabiliser formalism* proposed by Gottesman [19], we combine it with state-of-the-art *computer-assisted theorem proving* techniques to mechanise the verification process. We believe our work serves as a solid and practical foundation for static verification of quantum error correction codes, and provides insights on the broader context of formally verifying quantum software.

1.1 Motivation and Research Question

This study sets the broader background in formal verification of quantum programs. Classical programs are mostly made correct through testing and debugging. However, directly importing these techniques is challenging in quantum settings. This is mainly because of the probabilistic and nondeterministic nature of quantum information. Besides, the computational resources are scarce and expensive, making experiments on quantum computers much more costly [12]. As a consequence, *formal verification* (Section 2.5) has been seen as an alternative to traditional testing strategies [8]. Formal verification allows quantum programs to be statically analyzed by their formal semantics, without the need to run the program on a quantum computer. A few quantum verification frameworks have been proposed over the past decade. Two notable ones are SQIR [25] and CoqQ

[55]. These two foundational frameworks demonstrate the feasibility of applying formal verification to basic quantum programs.

Specifically, this study focuses on the *quantum error correction codes* (short as quantum codes). Quantum error correction codes are coding schemes used to detect or correct errors in quantum information. Quantum codes have been seen as a foundational component towards a fault-tolerant quantum computing, since *quantum noise* presents a major challenge for building any useful quantum programs [42]. Besides their crucial position in fault-tolerant quantum computing, they also have rich program properties and complex structures. This makes verifying them challenging, hence by applying different verification techniques, we can examine and compare the effectiveness of each method.

Verifying quantum programs is challenging, and a major reason is the scalability issue. Quantum programs are known to feature non-deterministic semantics, and the quantum state space grows exponentially due to *superposition* of states. This not only hinders formal verification, but also makes pen-and-paper reasoning difficult. For quantum code, one pen-and-paper technique that physicists used to reduce the difficulty in reasoning problem is the *quantum stabiliser code formalism* (short as *stabiliser formalism*). The stabiliser formalism, originally introduced by Gottesman [19], provides a framework for reasoning about quantum error correction codes. Using stabiliser formalism, the reasoning is made easy by taking advantage of algebraic methods. Having this rationale, we aim to integrate stabiliser formalism with computer-assisted formal verification (the *proof assistant*) for a efficient means of formal verification. Specifically, we ask *whether formal verification of quantum error correction codes can be efficiently achieved by stabiliser formalism?*

1.2 Contributions

We presented our investigation of the research problem by developing a formalisation *Coq-QECC* in the proof assistant Coq.[1] Coq-QECC formalize general Quantum Error Correction Codes based on the *stabiliser formalism*, a celebrated theoretical framework proposed by Gottesman in 1997 [19]. Coq-QECC takes advantage of the state-of-the-art verification frameworks of quantum computing and formal mathematics, namely SQIR [25], QuantumLib[2] and MathComp [35]. To show the usability of Coq-QECC, we apply it to verify some key properties of a selection of quantum codes and compare the verification with prior efforts. These examples demonstrate that Coq-QECC can efficiently verify quantum codes that are otherwise hard to handle by existing approaches. This validates our hypothesis that stabiliser formalism can not only be a theoretical framework, but also help achieve efficient formal verification.

Coq-QECC comes with an open source repository,[3] which is readily reusable in the proof assistant Coq. While this thesis is intended to be self-contained, we strongly recommend that readers download and inspect the Coq-QECC source code to better understand the topics discussed.

[1]We acknowledge that at the time when this thesis is compiled, Coq is renaming into "Rocq". However, we stick on the old name as most literature still use it.

[2]QuantumLib is a dependent library of QWIRE [40] and open source on https://github.com/inQWIRE/QuantumLib

[3]https://github.com/ExcitedSpider/Coq-QECC/tree/mcs-thesis-68

1.3 Thesis Outline

We begin by presenting the background of this research in Chapter 2, where we discuss quantum computing, quantum programming, and formal verification. This not only establishes the terminology and notations, but also highlights the broader significance of formally verifying quantum programs. Chapter 2 also includes the theory of quantum error correction codes and stabiliser formalism as the theoretical foundations to support the formal development.

With the general background established, we will turn our attention to work that is closely related to our research question (Chapter 3). We highlight a selection of state-of-the-art verification frameworks for quantum programs, namely SQIR [25] and CoqQ [55]. By comparing these frameworks, we clarify the rationale behind our choice of verification tools. In addition, we examine a prior efforts to formally verify quantum error correction codes using SQIR, which we adopt as the baseline for our evaluation.

Then, Chapter 5 and Chapter 6 discuss the formalism of Coq-QECC in the proof assistant Coq. We first formalize the critical algebraic structure in stabiliser formalism — the Pauli groups (Chapter 5). Based on Pauli groups, we provide other components of the stabiliser formalism: Pauli observables, Pauli operators, and structures of the error detecting code and correcting codes (Chapter 6).

To evaluate Coq-QECC, we select two quantum codes from the literature and verify their key properties (Chapter 7). We compare our verification with the existing ones using only SQIR. This shows the benefits of Coq-QECC. Based on the evaluation, we further analyse current limitation of Coq-QECC and propose them as future work in Chapter 8.

Chapter 2

Background

This chapter provides a background on quantum computing, quantum error correction codes and formal verification. Although not all topics are directly related to our contributions, they offer useful context for understanding how this work fits into the larger research landscape.

We first focus on the theoretical foundations of quantum computing and quantum error correction codes. Section 2.1 provides a compact introduction to quantum computing, where we establish the notations throughout this thesis. Then, Section 2.2 discusses the basics of the verification target — quantum error correction codes. Following it, we explain the stabiliser formalism, a celebrated theoretical framework for quantum error correction codes (Section 2.3). We believe it can benefit formal verification since this framework greatly simplifies the pen-and-paper reasoning of quantum codes.

Subsequently, we turn to the implementation and verification aspects. Section 2.4 discusses quantum programming languages, which are the essential tools to implement quantum algorithms. Next, we highlight several representative techniques in formal verification to illustrate their key characteristics in Section 2.5.

The topics discussed in this chapter support a few important insights that motivated this research in Section 2.6:
1. Debugging and testing are extremely complicated for Quantum Programs.
2. Quantum programs are amenable to formal verification.
3. The verification of quantum error correction codes is important.
4. Stabiliser formalism might be beneficial in the verification of quantum codes.

We assume only a general background in linear algebra and theoretical computer science, without requiring prior knowledge of quantum computing or formal methods. Readers with expertise in either area may skip the corresponding background sections and proceed directly to the conclusion of background (Section 2.6).

2.1 An Introduction to Quantum Computing

Since our research aims to verify quantum programs, a basic understanding of quantum computing is essential for discussing the verification process. This section is a compact introduction to quantum computing. We also establish the notations that are used throughout this thesis. Most notations in this section are from Nielsen and Chuang's celebrated textbook [37]. However, for measurements, we introduce our notation in Section 2.1.2, as there are no universally established conventions in the literature. Hence, readers familiar with quantum computing may skip this section but refer directly to Section 2.1.2.

2.1.1 Hilbert Space and Quantum States

The vector space that used to describe quantum computing is referred to as the *Hilbert space*, which is **a complex vector space equipped with an inner product**.[4] We begin by introducing Hilbert spaces from a linear algebraic perspective and establishing the notational conventions used throughout this thesis.

Let $\mathcal{H}_n$ be the Hilbert space of dimension n, and $\mathbb{C}$ be complex numbers. The *inner product* $\langle u, v\rangle$ is a function that takes two vectors u and v in a vector space and produces a complex number as output.

$$\begin{aligned} &\langle u, v\rangle : \mathcal{H}_n \times \mathcal{H}_n \to \mathbb{C} \\ &\langle u, v\rangle := u^\dagger v \end{aligned} \tag{1}$$

In which the $u^\dagger$ denotes the conjugate transpose of u. Many other concepts are based on inner product:

- Vectors u and v are *orthogonal* if their inner product is 0.
- The norm $\|v\| : \mathbb{C} \to \mathbb{C}$ of a vector v is defined as :

$$\|v\| = \sqrt{\langle v, v\rangle} \tag{2}$$

- A unit vector is a vector v such that $\|v\| = 1$.

In quantum mechanics, quantum states are represented just as vectors in Hilbert space. Hence, physicists extends the basic notation of inner product into what's known as the *bra–ket notation* [37].

- When a vector ψ represents a quantum state, it is notated as $|\psi\rangle$ (Read as "Ket")
- The complex conjugate of state $|\psi\rangle$ is notated as $\langle\psi|$ (Read as "Bra").
- The inner product of two quantum states $|\psi\rangle$ and $|\varphi\rangle$ are written as $\langle\psi|\varphi\rangle$ (Read as "Bra-ket")
- The norm of a state vector $|\psi\rangle$ is written as

$$\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle} \tag{3}$$

Another feature of Hilbert space is that a larger-dimensional Hilbert space can be formed from smaller ones using the tensor product:

Definition 2.1. The tensor product, also known as *Kronecker product*, takes an $m \times n$ matrix A and $u \times v$ matrix B and returns an $m * u \times n * v$ matrix:

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix} \tag{4}$$

For example, two 2-dimensional vectors can form a vector in a Hilbert space of dimension 4:

[4] This thesis only discusses finite-dimensional Hilbert spaces.

$$\begin{pmatrix} 0 \\ i \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \cdot 1 \\ 0 \cdot 0 \\ i \cdot 1 \\ i \cdot 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ i \\ 0 \end{pmatrix} \tag{5}$$

We now shift our focus to connect the Hilbert space in the sense of linear algebra to quantum information. The first component is what known as a *qubit*.

Single-Qubit Systems A qubit is the quantum analog of a bit. While a classical bit can be either 0 or 1, a qubit is more powerful — it can be both 0 and 1 simultaneously according to the principle of *quantum superposition*. We have introduced the notation of state vectors using Bra-ket notation. Now we use the notation to define a quantum state in Definition 2.2.

Definition 2.2. A quantum state $|\psi\rangle$ is a vector in Hilbert space that satisfies $\||\psi\rangle\| = 1$.

This condition of norm to be 1 is often known as the *normalisation condition*, and has the physical meaning that *the total probability of states sums to one*.

Now let's visit the basic single qubit state:

$$|0\rangle := \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle := \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{6}$$

An arbitrary single-qubit state can be described through a linear combination in the 2-dimensional vector space,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{7}$$

where α and β are complex numbers, also referred to as *phase*. They encode the probability of the state being in $|0\rangle$ or $|1\rangle$. It is easy to see that the normalisation condition requires that $\alpha^2 + \beta^2 = 1$.

To give a concrete example of a single-qubit state in superposition, consider this state (the plus state):

$$|+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{8}$$

It is a single-qubit state that lives in the equal superposition of $|0\rangle$ and $|1\rangle$. One fact that needs attention is that the relative phase of state components also has physical meaning. Consider the $|-\rangle$ state:

$$|-\rangle := \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{9}$$

Although $|+\rangle$ and $|-\rangle$ are both equal superpositions of $|0\rangle$ and $|1\rangle$, they can be distinguished by measurement due to their differing relative phases. We will discuss measurement as a separate section in Section 2.1.2.

Unitary Evolution A quantum operator U is simply a matrix in the same vector space as the quantum state. Applying an operator to a state $|\psi\rangle$ is a multiplication in the vector space.

$$\text{Apply } U \text{ on } |\psi\rangle \coloneqq U|\psi\rangle \tag{10}$$

Usually, physicists are only interested in *unitary* quantum operators:

Definition 2.3. A $n \times n$ matrix U is *unitary* if $U^\dagger U = UU^\dagger = I$, where I is the $n \times n$ identity matrix.

Theorem 2.4. (Unitary Norm Preservation) If a matrix U is unitary and ψ is a unit vector, then $\|U|\psi\rangle\| = 1$.

remark. A succinct proof can be found in Robert's Thesis [44] on page 15.

Unitary operators are favoured because Theorem 2.4 implies that applying a unitary operator to a state can maintain the norm of a state vector. That is, the *total probability remains 1* after applying a unitary operator. Otherwise, if the total probability were to become less than 1, it would indicate a loss of information; if the total probability exceeds 1, the state would lack physical meaning.

An example of a unitary transformation is the Hadamard operator H:

$$H \coloneqq \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{11}$$

It's straightforward to verify H is unitary by examining $H^\dagger = H$ and $HH = I$. The Hadamard operator is usually used to create an equal superposition:

$$H|0\rangle = |+\rangle, H|1\rangle = |-\rangle \tag{12}$$

Corollary 2.5. A sequence of unitary operations $U = U_1 \times ... \times U_n$is also unitary.

A proof of Corollary 2.5 can be found in [37] on page 81. Applying a sequence of unitary operators to a state is known as *unitary evolution*. A unitary evolution f describes a quantum program without measurement:

$$f : |\psi\rangle \mapsto U_k ... U_2 U_1 |\psi\rangle \tag{13}$$

Quantum computing is built based on a set of postulates of quantum mechanics. Unitary evolutions play an important role based on Postulate 2.6.

Postulate 2.6. The evolution of a closed[5] quantum system is described by a unitary transformation.

Pauli Matrices As an example of unitary evolution, let's consider the *Pauli matrices*. The Pauli matrices $\{I, X, Y, Z\}$ and their action of single-qubit states are defined in Table 1.

[5]Informally, a closed quantum system is a system that does not interact with its external environment. That is, there is no measurement involved.

Identity	$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto \|1\rangle$
Bit-Flip	$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\|0\rangle \mapsto \|1\rangle$ $\|1\rangle \mapsto \|0\rangle$
Bit-phase-flip	$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\|0\rangle \mapsto i\|1\rangle$ $\|1\rangle \mapsto -i\|0\rangle$
Phase-flip	$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\|0\rangle \mapsto \|0\rangle$ $\|1\rangle \mapsto -\|1\rangle$

Table 1: Single qubit Pauli matrices and their action

Compared to arbitrary unitary evolution, Pauli matrices are favoured by physicists for lots of reasons. Firstly, they are all unitary matrices and hence they can represent a unitary evolution on quantum states. Secondly, they have a clear meaning of actions. For example, the X matrix flips $|0\rangle$ to $|1\rangle$ and vice versa. This clear semantics also make them popular in quantum programming. And more importantly, they form a basis for the 2-dimensional Hilbert space. This allows simplifying reasoning an arbitrary unitary evolution U into reasoning a linear combination of $\{X, Y, Z, I\}$.

Theorem 2.7. (Pauli Matrices Decomposition) Any 2×2 Unitary Matrix U can be decomposed using the *Pauli basis* as :

$$U = a_0 I + a_1 X + a_2 Y + a_3 Z \tag{14}$$

where $a_0 ... a_3$ are four complex numbers.

remark. One proof of Theorem 2.7 be found in [37] on Page 175 as the *Z-Y decomposition* of a single qubit operation.

Multi-qubit systems We have mentioned the tensor product only in the meaning of linear algebra. In quantum information, multiple qubits can be combined to form a larger state by the tensor product '$\otimes$'. For example, $|0\rangle \otimes |1\rangle$ describes a state where the first qubit is 0 and the second is 1. Conventionally, when it does not cause ambiguity, tensor product symbols '$\otimes$' can be omitted and qubit states can be merged, i.e. $|0\rangle \otimes |1\rangle = |01\rangle$.

Lemma 2.8. An n-qubit system is a vector in 2^n dimension Hilbert space $\mathcal{H}_{2^n}$.

Proof. Lemma 2.8 is a direct result of the definition of tensor product. Recall it's definition (Definition 2.1).

$$\otimes : \mathbb{C}^{m \times n} \to \mathbb{C}^{u \times v} \to \mathbb{C}^{mu \times nv} \tag{15}$$

where notation $\mathbb{C}^{m \times n}$ refers to the set of all $m \times n$ matrices with entries as complex numbers. Recall that a single qubit is in $\mathbb{C}^{2 \times 1}$. Then the tensor product of n qubit shall be a vector in $\mathbb{C}^{2^n}$, and this set of vector belongs to the 2^n dimensional Hilbert space. □

Let's consider an important two-qubit state as an example. The *bell state* is an equal superposition of 00 and 11:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{16}$$

Note that this state is not the same as the mere combination of two $|+\rangle$s. To see this,

$$\begin{aligned} |+\rangle \otimes |+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \neq |\beta_{00}\rangle \end{aligned} \tag{17}$$

This follows the linearity of the tensor product and the definition of $|+\rangle$.

This mathematical property has the physical meaning that the subsystems are no longer independent. In the language of quantum mechanics, $|\beta_{00}\rangle$ is a state in *entanglement*, that is, a state that cannot be decomposed into a tensor product of two smaller states. Entanglement makes the information of two (or more) qubits becomes correlated. For example, knowing the first qubit is 0 enables one to immediately determine the second qubit is also 0, no matter how distant these two qubits might be.

Finally, we write a general n-qubit state as:

$$|\psi\rangle = \sum_{i=0}^{2^n-1} a_i |i\rangle \tag{18}$$

Equation 18 implies that fact that n qubits can contain up to 2^n basic states in superposition simultaneously.[6] Recall the normalisation condition in Definition 2.2, it refines to

$$\sum_{i=0}^{2^n-1} |a_i|^2 = 1 \tag{19}$$

Which ensures that the total probability of superposition sums to 1.

2.1.2 Measuring Quantum State

We have discussed what are quantum states. This section discusses another fundamental aspect of quantum computing: *measurements*. According to a well-established postulate in quantum mechanics, the only way to extract information about a quantum system is through measurement [37]. In other words, to classical observers, the state of the system remains unknown until a measurement is performed. In contrast, two quantum states are *indistinguishable* if all measurements yield identical outcome statistics when applied to them.

Measurement is often counter-intuitive and is commonly cited as an aspect of *quantum weirdness* [50]. To build a clearer understanding, this section introduces two forms of measurement: a simpler, more intuitive version aimed at developing intuition, and a more fundamental, but formally complex, definition based on the quantum formalism.

[6] This exponential representational capacity is often the source of quantum speed up. However, quantum complexity theory is not the focus of this thesis hence we do not unfold this fact here. Interested readers can check section 4.5.5 in Nielsen and Chuang's Textbook [37].

Measurement In Computational Basis In quantum mechanics, a *measurement* is the process that tests a quantum state and yields a real number result. This section describes a simplified representation of measurement known as the *measurements in computational basis*. From a computer science perspective, we can describe the measurement in computational basis as a function Meas_C

$$\text{Meas}_C : \mathcal{H} \to \mathbb{R} \times \mathcal{H} \tag{20}$$

where $\mathcal{H}$ is the Hilbert space and $\mathbb{R}$ is the set of real numbers. It is a function that takes a state vector in a Hilbert space, and returns a real number together with the state after measurement. What is not expressed in this type notation is that this function is not deterministic. Rather, it is *probabilistic*. Consider an arbitrary single-qubit quantum state $|\psi\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{21}$$

When measuring $|\psi\rangle$ in the computational basis, it will yield 0 with probability $|\alpha|^2$ and leave the state as $|0\rangle$; or $|1\rangle$ with probability $|\beta|^2$:

$$\text{Meas}_C|\psi\rangle = \begin{cases} (0, |0\rangle) & \text{with probability } |\alpha|^2 \\ (1, |1\rangle) & \text{with probability } |\beta|^2 \end{cases} \tag{22}$$

More generally, Let $|\varphi\rangle = \sum_i^{2^n-1} a_i|i\rangle$ be a arbitrary n-qubit state, then

$$\text{Meas}_C|\varphi\rangle = (m, |m\rangle) \text{ with probability } |a_m|^2 \tag{23}$$

i.e. measuring $|\varphi\rangle$ in computational basis will yield result m with probability $|a_m|^2$, and leave the result after measurement as $|m\rangle$.

The fact that measurement collapses superposition is also known as *wavefunction collapse* [37], which is a permanent loss of information — it is physically impossible to know other component in the state superposition after measurement. We discuss the implications of this quantum phenomenon at the end of this section.

Projective Measurement Postulate The previous section is a simplified definition of measurement. However, to build the theory of quantum error correction code, we need a slightly more fundamental and complex definition of measurement — The *Projective Measurement*. A measurement is described by a *quantum observable*:

Definition 2.9. (Quantum Observable) An observable M is a square matrix that satisfies.

$$M^\dagger = M \tag{24}$$

Where $M^\dagger$ is defined as the transpose conjugate of M. In mathematical language, this property is also known as *Hermitian*.

Theorem 2.10. (Real Decomposition of Hermitian Matrix) For any Hermitian matrix M, M has a unique spectral decomposition with real eigenvalues. Let k be the number of eigenvalues, m be the eigenvalue and v be the eigenvector corresponding to m,

$$M = \sum_{i=0}^{k} m_i |v_i\rangle\langle v_i| \tag{25}$$

In quantum computing, the product $|v_i\rangle\langle v_i|$ is usually referred to as the *projector* onto the eigenspace corresponding to m_i. All eigenspaces are orthogonal to each other.

Theorem 2.10 is a result from linear algebra. Readers can check Nielsen and Chuang's Section 2.1.6 for proofs and terminology [37]. For example, the Pauli-Z matrix

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{26}$$

is Hermitian, and has the following unique spectral decomposition:

$$Z = 1 \cdot |0\rangle\langle 0| + (-1) \cdot |1\rangle\langle 1| \tag{27}$$

A measurement applies an observable — via its spectral decomposition — to yield an eigenvalue outcome and collapse the state; This is formalised as a standard postulate of quantum mechanics as Postulate 2.11.

Postulate 2.11. A projective measurement is described by an observable M which has a spectral decomposition (Theorem 2.10).

$$M \coloneqq \sum_{i=0}^{k} m_i |v_i\rangle\langle v_i| \tag{28}$$

Upon measuring any state $|\psi\rangle$, the possible measurements are the eigenvalues of M, with the probability of getting the eigenvalue m to be:

$$p(m) \coloneqq \langle\psi||v_i\rangle\langle v_i||\psi\rangle \tag{29}$$

And the state after measurement yields m is

$$\frac{|v_i\rangle\langle v_i||\psi\rangle}{\sqrt{p(m)}} \tag{30}$$

For example, if we use Z as an observable (Equation 27) to measure an arbitrary single-qubit state $|\psi\rangle$

$$|\psi\rangle = a|0\rangle + b|1\rangle \tag{31}$$

The probability of yielding 1 is

$$\begin{aligned}\langle\psi||0\rangle\langle 0||\psi\rangle &= (a\langle 0| + b\langle 1|)(|0\rangle\langle 0|)(a|0\rangle + b|1\rangle) \\ &= (a\cdot\langle 0|)|0\rangle\langle 0|(a\cdot|0\rangle) + (a\cdot\langle 0|)|0\rangle\langle 0|(b\cdot|1\rangle) \\ &+(b\cdot\langle 1|)|0\rangle\langle 0|(a\cdot|0\rangle) + (b\cdot\langle 1|)|0\rangle\langle 0|(b\cdot|1\rangle) \\ &= a^2 + 0 + 0 + 0 = a^2\end{aligned} \tag{32}$$

And the state after measuring 1 is

$$\frac{|0\rangle\langle 0|(a|0\rangle + b|1\rangle)}{\sqrt{a^2}} = |0\rangle \tag{33}$$

This can be checked by the fact that $|0\rangle$ and $|1\rangle$ are orthogonal. i.e. $\langle i||j\rangle = 0 \leftrightarrow i \neq j$. Similarly, the probability of measuring $|\psi\rangle$ using observable Z yielding -1 is b^2, and the state after measurement is $|1\rangle$

It is not hard to see that the measurement in the computation basis is a special case of the projective measurement postulate (Postulate 2.11). Indeed, for any single qubit state, the observable is Z; And for any n-qubit state, $Z^{\otimes n}$ is the observable[7]. Therefore, the computational basis is also known as the Z-basis measurements[8].

In particular, we are interested in the measurements that yield a certain result. Thus, we define a notation for such measurements:

Definition 2.12. The notation

$$\text{Meas}(M, |\psi\rangle) = m \tag{34}$$

means applying projective measuring using observable M on $|\psi\rangle$ yields result m with probability 1, and the state $|\psi\rangle$ remain unchanged after measurement.

Having this notation, we introduce an important corollary of Postulate 2.11.

Corollary 2.13. (Measurement on Eigenstate) If a state vector $|\psi\rangle$ is an eigenvector of observable M corresponding to eigenvalue m, measuring M on $|\psi\rangle$ yields m with certainty. i.e.

$$\forall M \forall |\psi\rangle, M|\psi\rangle = m|\psi\rangle \rightarrow \text{Meas}(M, |\psi\rangle) = m \tag{35}$$

remark. Corollary 2.13 is widely accepted as a known fact in literature. One proof can be found in Hall's *Quantum Theory for Mathematicians* [21], specifically in Section 3.6 "Axiom of Quantum Mechanics: Operators and Measurements", where it appears as Proposition 3.11 (Eigenvectors).

Proof. Firstly, consider the spectral decomposition of M:

$$M = \sum_{i=0}^{k} m_i |v_i\rangle\langle v_i| \tag{36}$$

where $\{|v_i\rangle\}$ are orthogonal eigenvectors, and $|v_i\rangle\langle v_i|$ is a projector onto the eigenspace of m_i.

[7]Notation $U^{\otimes n} := U \otimes U \otimes ... \otimes U$ is the tensor product of n Us.

[8]For more information about the equivalence of these two measurement postulates, please see page 90 of [37]

By Postulate 2.11, the probability of getting m_i in the measurement if $p(m) = \langle\psi||v_i\rangle\langle v_i||\psi\rangle$. By case analysis on if $m = m_i$:

- if $m = m_i$, then $|\psi\rangle$ is a vector in the eigenspace of m_i. This gives

$$p(m_i) = \langle\psi||v_i\rangle\langle v_i||\psi\rangle = \langle\psi|\psi\rangle = \||\psi\rangle\|^2 = 1 \tag{37}$$

  Note that the norm of $|\psi\rangle$ is 1, which is by both the properties of Hermitian matrix, and the normalisation condition of quantum states.
- If $m \neq m_i$ then $|\psi\rangle$ is outside the eigenspace of m_i. Hence, $\langle v_i|\psi\rangle = 0$ since all eigenspaces are orthogonal to each other by Theorem 2.10.

$$p(m_i) = \langle\psi||v_i\rangle\langle v_i||\psi\rangle = 0 \tag{38}$$

In conclusion, we have:

$$p(m_i) = \begin{cases} 0 \text{ if } m_i \neq m \\ 1 \text{ if } m_i = m \end{cases} \tag{39}$$

Since we will only measure m, the state after measurement is

$$\frac{|\psi\rangle\langle\psi||\psi\rangle}{\sqrt{p(m)}} = |\psi\rangle \tag{40}$$

These two results together are equivalent to

$$\text{Meas}(M, |\psi\rangle) = m \tag{41}$$

by the definition of the notation. □

Corollary 2.13 is mentioned in many textbooks of quantum computing. Please see Nielsen and Chuang's [37] on page 88 and Yanofsky's [53] on page 126, Postulate 4.3.1.

We would like to mention another important corollary of measurement:

Corollary 2.14. For any observable M and a state $|\psi\rangle$, a global phase does not affect the measurement result. That is, the state $e^{i\theta}|\psi\rangle$ has the same measurement result as $|\psi\rangle$ where θ is a real number.

Proof of Corollary 2.14. Using the definition of measurement (Postulate 2.11), measuring any observable M on state $|\psi\rangle$ yields eigenvalue m is of probability $p(m)$:

$$p(m) = \langle\psi||v_i\rangle\langle v_i||\psi\rangle \tag{42}$$

Now consider $e^{i\theta}|\psi\rangle$, getting m from the same measurement is

$$\begin{aligned} p(m)' &= e^{-i\theta}|\psi\rangle\langle\psi||v_i\rangle\langle v_i||\psi\rangle e^{i\theta}|\psi\rangle \\ &= e^{-i\theta}e^{i\theta}|\psi\rangle\langle\psi||v_i\rangle\langle v_i||\psi\rangle|\psi\rangle \\ &= \langle\psi||v_i\rangle\langle v_i||\psi\rangle = p(m) \end{aligned} \tag{43}$$

Note that these two distributions of probability are the same. Therefore, a global phase is undetectable through any measurement. □

As a result of Corollary 2.14, we can always throw a complicated phase away, because measurement is the sole way to get any information from quantum computing processes (see Section 2.4). Note that this does not apply to the *relative phase* of superposed states.

Limitation of Measurement and Its Implications on Testing Measurement makes quantum computing different from classical computing. In classical computing, viewing program state is a deterministic process and would not modify state. In contrast, measuring program state yields a probabilistic result and potentially changes the underlying quantum states This limitation of quantum measurement poses significant challenges for debugging quantum programs. First, observing the program state during execution is inherently risky, as measurement disturbs the quantum state. Second, a single measurement cannot reveal the full information of a superposed state. This limitation complicates software testing, because many of these techniques essentially rely on inspecting program states. Note that this does not mean testing is completely impossible. Current testing techniques are mostly statistic, based on repeated measurement [3]. However, statistic-based testing is generally more complicated to design and costly to implement.

2.1.3 Quantum Programs and Quantum Circuits

A quantum program is a sequence of operations and measurements. Visually, a quantum program can be represented as a *quantum circuit*, which is inspired by classical logic circuits. Figure 1 shows a quantum circuit as an example.

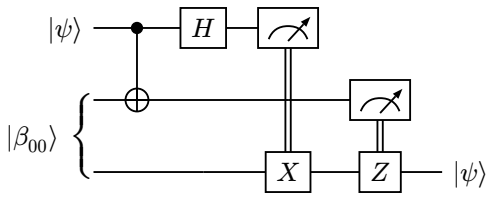


Figure 1: The quantum circuit represents a quantum teleportation program, in which the sender sends information in qubit $|\psi\rangle$ to the receiver's qubits $|\beta_{00}\rangle$.

The horizontal lines in a circuit represent quantum states. Quantum gates are placed on the circuit in temporal order from left to right. In Figure 1, the rightmost boxes on the first two qubits represent measurements. As we mentioned earlier, a measurement collapses the superposition of a quantum state and reads it as a classical, deterministic state.

In Figure 1. The X gate and Z gate represent the Pauli Matrix X and Z in Table 1. And as mentioned earlier, the H gate is usually used to create equal superpositions (Equation 12).

Finally, the controlled-not (CNOT) gate  is a two-qubit gate. It works like the classical "if" command, which flips the second qubit if the first one is $|1\rangle$, and has no effect otherwise.

$$CNOT\,|00\rangle = |00\rangle, CNOT\,|01\rangle = |01\rangle$$
$$CNOT\,|10\rangle = |11\rangle, CNOT\,|11\rangle = |10\rangle \tag{44}$$

Figure 2: The effect of *CNOT* gates on basic states.

No-Cloning Theorem Informally, the *no-cloning theorem* states that it's impossible to create an identical copy of an arbitrary unknown quantum state.

Theorem 2.15. (No-Cloning). There is no such unitary transformation U:

$$|\psi\rangle \otimes |0\rangle \overset{U}{\rightarrow} |\psi\rangle \otimes |\psi\rangle \tag{45}$$

One proof of the no-cloning theorem can be found in [37] on page 532, proved by the definition of unitary transformation. This theorem has a fundamental impact on quantum programming. To illustrate, while it is trivial to copy a variable in classic programs, copying a qubit is impossible. It also contributes to the difficulty of debugging in quantum programs. Quantum programming languages are also required to be designed carefully to avoid this restriction. For example, QWIRE uses linear types to enforce no-cloning [40].

Additional Material In this section, we tried to cover the most relevant aspects of quantum computing and quantum programs for our work. However, this introduction is still highly incomplete and informal. One example is that we deliberately avoid mentioning *mixed states* and *density matrix representation*. We also omit the *Bloch Sphere* visualisation, which views a single-qubit state as a point on a 3-dimensional sphere, and reasoning quantum operations as rotations on the sphere. We strongly encourage readers to check Nielsen & Chuang's Textbook [37] for a more comprehensive and formal introduction to quantum computing and quantum programming.

2.2 Quantum Error Correction Code

We now turn our attention to the primary verification target of this work: *quantum error correction codes*. Following Roffe's tutorial [46], we give the following definition of quantum error correction codes:

Definition 2.16. A quantum error correction code (short as *quantum codes*) is a scheme to encode quantum information into a larger Hilbert space, such that certain quantum errors can be detected or corrected without collapsing the encoded state.

Quantum codes play a central role in fault-tolerant quantum computing. Quantum states are susceptible to *quantum errors*, which corrupt the information in states [42]. For a state-of-the-art quantum computer, one out of every 100 to 1000 operations will result in an error on average [1]. Quantum codes ensure the integrity of quantum information throughout computation. As such, formally verifying the correctness of quantum codes is a critical step toward certified fault-tolerant quantum computing.

This section introduces the general theory of quantum error correction codes to support the discussion and formalism. We first discuss quantum errors in Section 2.2.1. Then, we discuss properties of quantum codes by discussing a concrete code to introduce basic terminology (Section 2.2.2).

2.2.1 Quantum Errors

Quantum information are sensitive to the environment and could be easily corrupted by noise. In quantum computing, environment noise that corrupts states is referred to as *quantum errors*. A quantum error can be seen as an arbitrary unitary evolution.[9] That is to say, let $|\psi\rangle$ be a state, a quantum error E is a unitary matrix that applied to $|\psi\rangle$:

$$|\psi\rangle \xrightarrow{E} E|\psi\rangle \tag{46}$$

Let's consider the Pauli matrices $\{X, Y, Z\}$ as errors. Let $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, an X error typically flips the qubit:

$$X|\psi\rangle = \alpha X|0\rangle + \beta X|1\rangle = \alpha|1\rangle + \beta|0\rangle \tag{47}$$

And a Z error flips the phase:

$$Z|\psi\rangle = \alpha Z|0\rangle + \beta Z|1\rangle = \alpha|0\rangle - \beta|1\rangle \tag{48}$$

And Y has the effect of combining a bit-flip error and a phase-flip error since $Y = XZ$.

Quantum errors are more challenging to address. One reason is the existence of phase-flip errors, which have no classical analogue. The principles behind quantum information also complicate the discussion.

- *No cloning.* A simple coding in classical settings is the repeating code, which simply duplicates the information. However, this is not possible since quantum mechanics prohibits cloning of information.
- *Wavefunction Collapse.* It might be natural to measure states to get the error. But measurement might collapse the quantum state under observation, which causes information loss.

Therefore, quantum codes must be carefully designed with appropriate encoding schemes. It must not violate the no-cloning theorem. Plus, if measurement is involved, one must ensure that the measurement does not collapse the state. We will discuss these aspects in the example code presented in the next section.

2.2.2 The Three-Qubit Bit-flip Code

To understand quantum codes, we first look at classical codes and adapt them to a simple quantum code – The three-qubit bit-flip code –to illustrate concepts in quantum codes.

In the early stage of electronic computers, RAM was often unreliable, like current qubits. The most common errors are bit-flips, where one bit unintentionally switches from 0 to 1 (or vice versa). Hence, the idea of *error correction code memory* is proposed to protect information from bit-flip errors by adding parity bits [22]. One of the simplest schemes is the repetition code, which encodes a logical bit with three physical bits:

$$\begin{aligned} 0 &\to 000 \\ 1 &\to 111 \end{aligned} \tag{49}$$

[9] While physically, quantum noise can be non-unitary, detecting and correcting unitary errors is sufficient to protect against non-unitary quantum noise [31].

This classical code can detect and correct a single bit flip by a majority vote when the logical bit is accessed. i.e. '000' turns to '100', '010' or '001'.

How to apply this classical coding scheme to a quantum setting? For any single logic qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we can also encode it using two redundant qubits:

$$|\psi\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L = \alpha|000\rangle + \beta|111\rangle \tag{50}$$

Where subscript L notates the encoded states, which are known as the *logical state*. All possible logical states form the *code space*:

Definition 2.17. A code space $\mathcal{C} \in \mathcal{H}$ of a quantum code C is a subspace of the Hilbert space that consists of all valid encoded states of C.

remark. The structure of the code space essentially defines a quantum code. We can use the code space to refer to a quantum code. We adopt this convention throughout this thesis.

Following the definition of the code space, we define the following terminology:

- A *code word* is a specific quantum state within the code space
- An *error state* is a state that lies outside the code space
- An *error space* is a subspace consisting of only error states.

One common misunderstanding of this three-qubit code is that it violates the no-cloning theorem. But instead, this is not the case. To see this, if we "clone" the state of the first qubit to the two following qubits, the resulting state should be

$$\begin{aligned} |\psi\rangle'_L &= (\alpha|0\rangle + \beta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle) \\ &= \alpha^3|000\rangle + \alpha^2\beta|001\rangle + \alpha^2\beta|010\rangle + \alpha\beta^2|011\rangle \\ &+\alpha^2\beta|100\rangle + \alpha\beta^2|101\rangle + \alpha\beta^2|110\rangle + \beta^3|111\rangle \end{aligned} \tag{51}$$

Clearly, $|\psi\rangle_L$ is not the same as $|\psi\rangle'_L$. Instead of cloning, what this code does is create an equal superposition across three qubits.

One implementation of $f_{3\text{qb}}$ is known in the form of following circuit program:

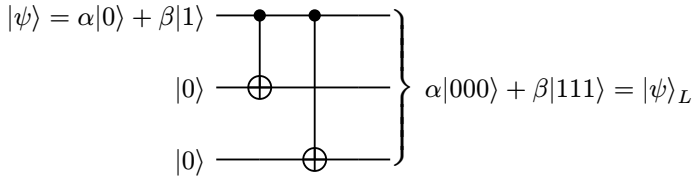


Figure 3: The implementation of 3-qubit bit-flip code $f_{3\text{qb}}$

Now let's consider the bit-flip errors that this code might suffer. Suppose a bit-flip error happened on the first qubit, the state undergoes the following unitary evolution:

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle \overset{X_1}{\longmapsto} \alpha|100\rangle + \beta|011\rangle \tag{52}$$

This evolution is labelled as X_1, since it can be seen as applying the following operation:

Claim 2.18. The operator $X_1 \coloneqq X \otimes I \otimes I$ is has the effect of flipping the first qubit of $|\psi\rangle$.

Proof. This is straightforward to verify by the fact that $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$. □

remark. Similarly, single bit-flip error on the second and the third qubit is $X_2 \coloneqq I \otimes X \otimes I$ and $X_3 \coloneqq I \otimes I \otimes X$.

Then, we can detect if there is an error by measurement. As discussed in the previous section, any measurement must not collapse the logical qubit $|\psi\rangle_L$ if there is no error. We name these special measurements as *syndrome detections*.

Definition 2.19. A measurement represented by an observable M is a syndrome detection measurement of a code $\mathcal{C}$ if measuring M preserves the state of any vector in $\mathcal{C}$, i.e. $\forall|\psi\rangle \in \mathcal{C}, M|\psi\rangle = |\psi\rangle$.

There are multiple possible syndrome detections for $|\psi\rangle_L$. One of them is Z_1Z_2

$$Z_1Z_2 \coloneqq Z \otimes Z \otimes I \tag{53}$$

Claim 2.20. Suppose there is a single bit-flip error applied to $|\psi\rangle$. Measuring observable Z_1Z_2 can determine whether the error is applied on the first two qubits.

Proof. To see this, we can look at the spectral decomposition of Z_1Z_2:

$$Z_1Z_2 = +1 \cdot (|00\rangle\langle 00| + |00\rangle\langle 00|) \otimes I + -1 \cdot (|01\rangle\langle 01| + |10\rangle\langle 10|) \otimes I \tag{54}$$

By Postulate 2.11, measuring $|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle$ yields 1 with certainty:

$$p(m = +1) = \langle\psi|_L(|00\rangle\langle 00| + |00\rangle\langle 00|)|\psi\rangle_L = 1 \tag{55}$$

If there is bit-flip error applied on the first qubit ($|\psi\rangle_L = \alpha|100\rangle + \beta|011\rangle$) yields -1 with certainty:

$$p(m = -1) = \langle\psi|_L(|01\rangle\langle 01| + |10\rangle\langle 10|)|\psi\rangle_L = 1 \tag{56}$$

This is also true if there is a bit flip error on the second qubit[10]. Therefore, we know that there is a bit-flip error on one of the first two qubits of $|\psi\rangle_L$ if measuring Z_1Z_2 yields -1. □

Similarly, observable Z_2Z_3 can be used to know if there is a bit-flip error on the last two qubits. Combining these two results of measurement Z_1Z_2 and Z_2Z_3 allow one to locate exactly which one of the errors $\{X_1, X_2, X_3\}$ has happened (Table 2).

	Measure Z_1Z_2 yields $+1$	Measure Z_1Z_2 yields $= -1$
Measure Z_2Z_3 yields $+1$	No Error	X_1
Measure Z_2Z_3 yields -1	X_3	X_2

Table 2: The measurement of observables Z_1Z_2 and Z_2Z_3 in the 3-qubit bit-flip code.

After locating the bit flip, one can apply a recovery operation to recover the original state.

Claim 2.21. The recovery operation is simply applying the bit-flip operation again.

[10]Readers are encouraged to refer to Gottesman's [19] for a detailed proof.

Proof. Take X_2 as an example, applying it twice on $|\psi\rangle_L$ results in the original state $|\psi\rangle_L$.

$$X_2X_2|\psi\rangle_L = X_2X_2(\alpha|000\rangle + \beta|111\rangle) = X_2(\alpha|010\rangle + \beta|101\rangle) = \alpha|000\rangle + \beta|111\rangle = |\psi\rangle_L \quad (57)$$

Similarly, readers can check that it also holds for the errors X_1 and X_3. □

More generally, for any error operator expressed in the form of Pauli matrices, Recovery is often as simple as reapplying the error operator.

Theorem 2.22. For an n-qubit error operator $E \coloneqq E_1 \otimes ... \otimes E_n$ where $E_i \in \{X, Y, Z, I\}$,

$$E \cdot E = I^{\otimes n} \quad (58)$$

i.e. Applying the error E twice brings back the original state.

Proof of Theorem 2.22. Recall that the Pauli matrices satisfy $\forall P \in \{X, Y, Z, I\}, P \cdot P = I$. Since E is built by composing n Pauli matrices, $E \cdot E = I^{\otimes n}$ by induction.[11] □

Therefore, when designing and reasoning about quantum error correction codes, identifying the type of error and which qubit(s) were affected is usually more important than the recovery operation itself.

At last, we want to mention that a projective measurement can be implemented as circuit programs. Figure 4 shows one implementation of measuring Z_1Z_2.

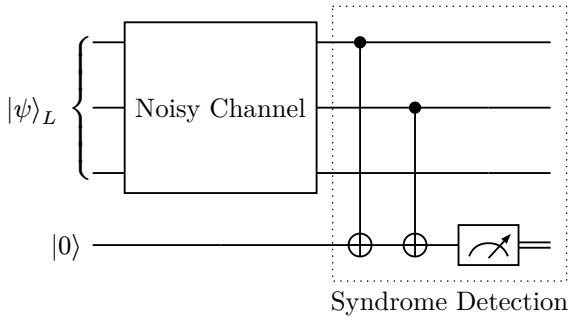


Figure 4: Syndrome detection measurement Z_1Z_2 using an ancillary qubit on the bottom

However, implementing projective measurement through quantum circuits typically requires auxiliary qubits, as illustrated in Figure 4.[12] The implementation is also not unique. Therefore, we primarily use the observable when discussing measurement. Circuits are only reserved for describing the encoding programs of quantum codes.

2.3 Stabiliser Formalism and Quantum Stabiliser Codes

The Stabiliser Formalism, originally introduced by Gottesman [19], is a widely used pen-and-paper framework for reasoning about quantum error correction codes. It simplifies the analysis of quantum codes by employing algebraic methods. In this section, we present the essential components of the formalism relevant to our work, and apply this theory to a relatively larger quantum code in Section 2.3.5.

[11] A formalized proof can be found in the PauliGroup.v as Coq theorem `pauli_involutive` in the source repository.
[12] This statement assumes only measurement in computational basis is support in circuit language.

2.3.1 Pauli Groups

We have defined Pauli matrices in Table 1. The four 2×2 matrices $\{X, Y, Z, I\}$ are unitary and Hermitian. Therefore, they can represent both as observables and unitary evolutions. For example, in Section 2.1.2, we have described Z matrix being used as an observable and in Section 2.2.2, we use X as the unitary evolution of bit-flip errors. More interestingly, when they are multiplied with one another, we get a Pauli matrix in return with possible phase factors $\{\pm 1, \pm i\}$.[13] The full multiplication table of a single-qubit Pauli operator is presented in Table 3.

$\cdot$	X	Y	Z	I
X	I	iZ	$-iY$	X
Y	$-iZ$	I	iX	Y
Z	iY	$-iX$	I	Z
I	X	Y	Z	I

Table 3: The multiplication table of Pauli operators

This means that the multiplication of Pauli matrices is closed. Besides, we know that matrix multiplication is associative, and I is essentially the identity element. This enables the formalism of seeing Pauli matrices as a group.

Definition 2.23. The single-qubit Pauli group $\mathcal{P}_1$ is defined by

$$\mathcal{P}_1 \coloneqq \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\} \tag{59}$$

And the multiplication is defined as the $\cdot$ in Table 3.

remark. If there is no ambiguity, we omit the multiplication symbol and the trivial $+1$ phase in terms. For example, we write $XY = iZ$ instead of $(+X) \cdot (+Y) = +iZ$.

Theorem 2.24. P_1 forms a group.

Proof.

1. Closure. Let $a \coloneqq \alpha A, b \coloneqq \beta B$ where $\alpha, \beta \in \{\pm 1, \pm i\}$ and $A, B \in \{I, X, Y, Z\}$. Then

$$a \cdot b = \alpha A \cdot \beta B = (\alpha\beta)(AB) \in P_1 \tag{60}$$

   by distributivity of scalar over matrix multiplication.
2. Associativity follows directly from the associativity of matrix multiplication:

$$\forall a \in P_1, \forall b \in P_1, \forall c \in P_1, abc = a(bc) \tag{61}$$

3. Consider the operator I, I is the identity element of P_1.

$$\forall a \in P_1, Ia = aI = a \tag{62}$$

4. Inverse element.
   - Consider the matrix component, notice that $\forall a \in \{X, Y, Z, I\}, aa = I$.

[13] The notation $\pm c$ means $\{+c, -c\}$.

- Consider the phase component, it is easy to check $\forall s \in \{\pm 1, \pm i\}, \exists s^{-1}, ss^{-1} = s^{-1}s = +1$. For example, $-i \cdot i = 1$[14].

Therefore, the inverse of any $p \coloneqq \alpha P \in P_1$ is:

$$p^{-1} = \alpha^{-1} P \tag{63}$$

where α is the phase and P is the Pauli matrix.

□

In Hilbert space, smaller states and operators can form a larger one by tensor products. This is also true for Pauli groups. We define the n-qubit Pauli group using the tensor product.

Definition 2.25. The n-qubit Pauli group $\mathcal{P}_n$ is defined by

$$\mathcal{P}_n \coloneqq \{P_1 \otimes ... \otimes P_n \mid P_1 ... P_n \in \mathcal{P}_1\} \tag{64}$$

And the multiplication is defined component-wise. Let P_a P_b be two elements of $\mathcal{P}_n$

$$P_a \cdot P_b = (P_{a1} \cdot P_{b1}) \otimes ... \otimes (P_{an} \cdot P_{bn}) \tag{65}$$

where P_{ai} is the i-th component of P_a.

remark. It is a common convention to omit the tensor product symbol when there is no ambiguity. i.e. $\mathrm{XYZ} = X \otimes Y \otimes Z$. This compact notation is also referred to as a *Pauli string* [37].

By Definition 2.25, it appears that all single-qubit components in $P_n \in \mathcal{P}_n$ can have a different phase. However, we usually consider all the components to have a $+1$ phase, and the P_n has a single global phase for easier reasoning. This simplification is valid because of the distributivity of the tensor product over scalar multiplication:

$$\forall \alpha \in C, \forall \beta \in C, \forall A \in \mathcal{H}_n, \forall B \in \mathcal{H}_n, (\alpha A) \otimes (\beta B) = (\alpha\beta)(A \otimes B) \tag{66}$$

For example:

$$(-Z) \otimes (iX) \otimes (iY) = (-1 * i * i)\ \mathrm{ZXY} = \mathrm{ZXY} \tag{67}$$

Therefore, when we discuss any element in P_n, we assume that it only has a global phase throughout this thesis.

Having this simplification in mind, we now prove $\mathcal{P}_n$ is a finite group.

Theorem 2.26. $\mathcal{P}_n$ is a group.

A machine-checked proof of Theorem 2.26 is presented in the PauliGroup.v in the Coq repository. We briefly mention the proof idea:

[14]Here we use $\cdot$ to represent multiplication of complex numbers. This is because the phase set $\{\pm 1, \pm i\}$ also forms a group. Readers are encouraged to check it.

Since we have already known group P_1 (Theorem 2.24), according to how P_n is constructed, we can prove P_n forms a group by induction on length n.

- If $n = 1$, then $P_n = \mathcal{P}_1$ forms a group.
- If $n = n' + 1$ and we assume $P_{n'}$ forms a group and try to show P_n forms a group.

we know that

$$\forall p \in P_n, \exists p' \in P_{n'}, \exists h \in P_1, p = h \otimes p' \tag{68}$$

by the construction of $\mathcal{P}_n$. Let p_1 and p_2 be two elements of P_n, then

$$\begin{aligned} p_1 \cdot p_2 &= (h_1 \otimes p_1') \times (h_2 \otimes p_2') \\ &= (h_1 \times h_2) \otimes (p_1' \times p_2') \end{aligned} \tag{69}$$

Since $h_1 \times h_2$ is the multiplication on group $\mathcal{P}_1$ and we have proved it is a group multiplication (see Theorem 2.24). By the induction hypothesis, $(p_1' \times p_2')$ is also a multiplication on group $P_{n'}$. We can prove that $p_1 \cdot p_2 = (h_1 \times h_2) \otimes (p_1' \times p_2')$ is a group multiplication by the linearity of tensor product $\otimes$.

We now discuss a few important properties of Pauli groups.

Lemma 2.27. $\forall P \in \mathcal{P}_n, P$ is unitary.

Proof of Lemma 2.27. We can do an induction on n.

- If $n = 1$, it matches the fact that all elements in $\mathcal{P}_1$ are unitary.
- If $n = n' + 1 \wedge \forall P' \in \mathcal{P}_{n'}, P'$ is unitary. By the construction of P, we have

  $$P = P' \otimes P_1 \tag{70}$$

  where $P_1 \in \mathcal{P}_1$. Using the fact that

  $$\forall A \forall B, \text{Unitary}(A) \to \text{Unitary}(B) \to \text{Unitary}(A \otimes B) \tag{71}$$

  We know that P is also unitary.

□

As mentioned in Section 2.1, any unitary matrix can represent a unitary evolution of quantum states. Then we can attach this physical meaning to any element of P_n as a unitary evolution. In addition, when we applied two elements to a state, this simplification always holds:

Theorem 2.28. $\forall a, b \in P_n, \forall |\psi\rangle \in \mathcal{H}, a \times (b \times |\psi\rangle) = (a \cdot b) \times |\psi\rangle$

remark. In this proposition, $\times$ is the matrix multiplication in the Hilbert space $\mathcal{H}$, where the dimension of space is $\dim(\mathcal{H}) = 2^n$ (See Lemma 2.8). In contrast, $\cdot$ is the multiplication on group P_n of length n. Therefore Theorem 2.28 suggests a reduction in the cost of computation from exponential to linear.

Proof of Theorem 2.28. It can be proved by the fact that matrix multiplication in $\mathcal{H}_n$ is associative:

$$\forall a, b, c \in \mathcal{H}, a \times b \times c = a \times (b \times c) \tag{72}$$

□

2.3.2 Pauli Operators and Quantum Stabilisers

Lemma 2.27 implies that any element in $\mathcal{P}_n$ is unitary. In addition, we restrict our discussion only to Hermitian elements of $\mathcal{P}_n$, as they can also be used as an observable.

Definition 2.29. An n-qubit Pauli operator is a Hermitian element of $\mathcal{P}_n$.

remark. Remember Postulate 2.11, a Hermitian matrix can be used as a measurement observable. As such, we use the term *Pauli observable* when we are talking about using a Pauli operator as an observable to reduce ambiguity.

Lemma 2.30. $\forall P \in \mathcal{P}_n$, The phase of P is $\pm 1 \rightarrow P$ is Hermitian

Proof of Lemma 2.30. Let P_n be an element of $\mathcal{P}_n$. First of all, it is easy to check that all single-qubit Pauli matrices are Hermitian, i.e.

$$\forall P_1 \in \{X, Y, Z, I\}, P_1 P_1 = I \tag{73}$$

By the definition of multiplication on $\mathcal{P}_n$, all $P_n P_n = (P_n.\text{phase})^2 I^{\otimes n}$. By case analysis of $P.\text{phase} = \pm 1$:

- if $P_n.\text{phase} = \pm 1$, $P_n P_n = I^{\otimes n}$ i.e. P_n is Hermitian.
- if $P_n.\text{phase} = \pm i$, $P_n P_n = -I^{\otimes n}$

In conclusion, P_n is Hermitian if the global phase of P_n is ± 1. □

Lemma 2.30 means that any $P_n \in \mathcal{P}_n$ can be used as an observable. For example, the observable $Z_1 Z_2$in Section 2.2.2 is an element of P_3 as:

$$Z_1 Z_2 = Z \otimes Z \otimes I \in \mathcal{P}_3 \tag{74}$$

Technically, Lemma 2.30 allows any element of P_n to act as an observable. We particularly select elements that are with $+1$ phase as observable for easing difficulty in reasoning.

Let $|\psi\rangle$ be the codewords and M be an observable. Recall that all the syndrome measurements satisfy:

$$\text{Meas}(M, |\psi\rangle) = +1 \leftrightarrow M|\psi\rangle = +1|\psi\rangle = |\psi\rangle \tag{75}$$

By unfolding the definition of notation. Physically, this property states that applying operator M on $|\psi\rangle$ maintain its original information. In other words, the operator M fixes the code space where $|\psi\rangle$ might live. And measuring M on $|\psi\rangle$ is guaranteed not to cause any wavefunction collapse. We can define the *stabiliser* of a code space to be:

Definition 2.31. An Pauli operator $S \in \mathcal{P}_n$ is a stabiliser of an n-qubit state $|\psi\rangle$ if $|\psi\rangle$ is the $+1$ eigenvector of S:

$$S|\psi\rangle = |\psi\rangle \tag{76}$$

We also call $|\psi\rangle$ the stabiliser state of S.

For example, remember the code space of f_{3b} is

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle \tag{77}$$

Where α and β are variables and carry the information equivalent to a single qubit. It is easy to verify that Z_1Z_2 is a stabiliser of the code space $|\psi\rangle_L$

$$Z_1Z_2|\psi\rangle_l = |\psi\rangle_l \leftrightarrow \text{Meas}(Z_1Z_2, |\psi\rangle_L) = 1 \tag{78}$$

A single bit-flip error applied to any state in the code space can be described by $X_1|\psi\rangle_L$.

Finally, it is sufficient to prove the observable $Z_1Z_2 \coloneqq ZZI$ can detect a bit-flip error $X_1 \coloneqq XII$ on by checking the eigenvalue on the error state By Corollary 2.13:

$$Z_1Z_2(X_1)|\psi\rangle_L = -|\psi\rangle_L \leftrightarrow \text{Meas}(Z_1Z_2, X_1|\psi\rangle_L) = -1 \tag{79}$$

Furthermore, we can even throw $|\psi\rangle_L$ away and only consider the relation between observable (Z_1Z_2) and error (X_1).

Theorem 2.32. (Error Detecting Condition) A stabiliser S can detect error E if

$$SE = -ES \tag{80}$$

Where "$-$" operator is the negation of the global phase, i.e.

$$-(\pm 1) = \mp 1, -(\pm i) = \mp i \tag{81}$$

One detailed proof of Theorem 2.32 can be found in [37] on page 463 at § 10.5. We present a succinct proof as follows.

Proof of Theorem 2.32. Suppose $SE = -ES$, let $|\psi\rangle$ be the codewords stabilized by S,

$$SE|\psi\rangle = -ES|\psi\rangle = -E|\psi\rangle = -1 * E|\psi\rangle \leftrightarrow \text{Meas}(S, E|\psi\rangle) = -1 \tag{82}$$

That is to say, measuring S on state $E|\psi\rangle$ yields -1 with certainty. Since we have defined $|\psi\rangle$ to be the $+1$ eigenvector of S (Definition 2.31), the measurement result is enough to tell if error E has been applied. □

2.3.3 Properties of Quantum Stabilisers

We now discuss some properties of stabilisers to further illustrate why this formalism can make reasoning about quantum codes easier. Firstly, we can concatenate two Pauli operators:

Lemma 2.33. (Concatenation of Stabilisers). Let $|\psi\rangle$ and $|\varphi\rangle$ be two states stabilised by A and B, then $|\psi\rangle \otimes |\psi\rangle$ is a stabiliser state of $A \otimes B$.

Proof of Lemma 2.33. First, let's introduce the hypotheses from the definition ($|\psi\rangle$ and $|\varphi\rangle$ are stabiliser states)

$$A|\psi\rangle = |\psi\rangle, B|\varphi\rangle = |\varphi\rangle \tag{83}$$

Now consider the tensor product:

$$\begin{aligned}(A \otimes B)(|\psi\rangle \otimes |\varphi\rangle) &= (A \otimes B)(|\psi\rangle \otimes |\varphi\rangle) \\ &= (A|\psi\rangle) \otimes (B|\varphi\rangle) = |\psi\rangle \otimes |\varphi\rangle\end{aligned} \tag{84}$$

□

Lemma 2.33 is handy when one needs to prove a larger stabiliser state by composition of smaller ones. For example, if we know $|0\rangle$ is a stabiliser state of Z and $|+\rangle$ is the stabiliser state of X, then $|0\rangle|+\rangle$ is the stabiliser state of ZX.

Corollary 2.34. All Pauli operators have and only have eigenvalues ± 1.

Proof of Corollary 2.34. Firstly, note that all involutory matrices have and only have eigenvalues ± 1. e.g.

$$\forall A \in \mathcal{H}, A^2 = I \rightarrow Av = \lambda v \rightarrow \lambda = -1 \vee \lambda = 1 \tag{85}$$

And this applies to Pauli operators, since for any Pauli operator P , $PP = I$.[15] □

Recall Postulate 2.11, the measurement results of any observable can only be its eigenvalues. This implies that measuring any Pauli observables on any states, the result is either 1 or -1. This result enables one to view a Pauli observable as a means to divide the Hilbert space: the states with measurement result $+1$, and the states with result -1.

2.3.4 Stabiliser Code

With all the above theories, we can discuss the concept of stabiliser code.

Definition 2.35. A *stabiliser code* is a quantum error correction code whose code space is precisely described by a set of stabilisers.

Recall the example of the three-qubit bit-flip code and its syndrome (Table 2), the code space is precisely described by the condition that measuring the two stabiliser Z_1Z_2 and Z_2Z_3 yields $+1$.

[15] We also proved this corollary in the Coq Repository as Coq theorem `operator_eigenvalue` in Observable.v.

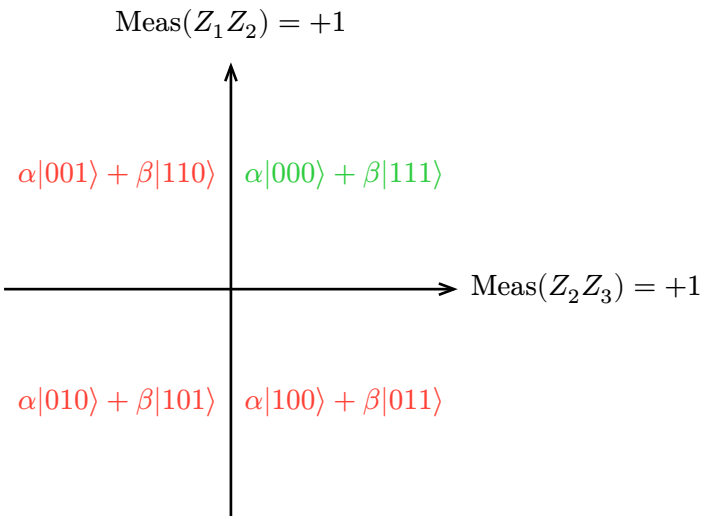


Figure 5: The stabiliser Z_1Z_2 and Z_2Z_3 divide the Hilbert space $\mathcal{H}_3$.

Figure 5 shows the visualisation of the discussed fact — The entire plane represents the 3-dimensional Hilbert space. Each quadrant corresponds to a subspace which decides by the pair of measurement results ± 1. Consider the 3-qubit bit-flip code f_{3b}, the code space $\alpha|000\rangle + \beta|111\rangle$ is identified by both measurements are $+1$. These three error subspaces can be reliably distinguished by the stabiliser measurements.

On the other hand, if measuring any stabiliser yields the result -1, we know that the state is in the error space. Figure 5 shows this analysis. For example, the error subspace $\mathcal{C}' = \alpha|100\rangle + \beta|011\rangle$ is identified by measuring Z_2Z_3 yields $+1$ while measuring Z_1Z_2 yields -1. Assuming there is only one single bit-flip error, we know that the error is on the first qubit, i.e. $X_1 := XII$. This analysis can be generalised as the concept of ***error syndrome***:

Definition 2.36. The *syndrome* of an error E on a quantum code $\mathcal{C}$ is the set of stabilisers of $\mathcal{C}$ that yield -1 upon measurement. In other words,

$$\text{Syndrome}(E, \mathcal{C}) = \{S \mid S \in \text{Stab}(\mathcal{C}) \wedge \forall |\psi\rangle \in \mathcal{C}, \text{Meas}(S, E|\psi\rangle) = -1\} \tag{86}$$

Using this language, we say the syndrome of error X_1 is $\{Z_1Z_2\}$.

Since measurement is the only way to get any information from a quantum system, if the syndrome of an error is empty i.e. no stabiliser measurement can detect this error, we say this code cannot detect the error. Formally,

Definition 2.37. An error E is undetectable by a stabiliser code $\mathcal{C}$ if the Syndrome$(E, \mathcal{C}) = \emptyset$.

For example, one can verify that the three-qubit bit-flip code cannot detect a phase-flip error $Z_1 := ZII$ since

$$Z_1Z_2(Z_1(\alpha|000\rangle + \beta|111\rangle)) = Z_1Z_2(\alpha|000\rangle - \beta|111\rangle) = \alpha|000\rangle - \beta|111\rangle \tag{87}$$

This shows the error state $\alpha|000\rangle - \beta|111\rangle$ is in the $+1$ eigenspace of Z_1Z_2. Therefore, measuring Z_1Z_2 yields $+1$ on this error space. Readers can verify that it's also true for the other stabiliser Z_2Z_3.

Finally, if two errors have the same syndrome, it's impossible to tell which error has happened. We say these two errors are *indistinguishable*.

Definition 2.38. Two errors E_1 and E_2 are indistinguishable by a stabiliser code $\mathcal{C}$ if

$$\text{Syndrome}(E_1, \mathcal{C}) = \text{Syndrome}(E_2, \mathcal{C}) \tag{88}$$

2.3.5 The Nine-Qubit Shor's Code

Now we discuss a concrete example of stabiliser code. We have discussed the 3-qubit bit-flip code (Section 2.2.2). However, practical codes require a larger size, and can tolerate both bit-flip and phase-flip errors. One such code is the nine-qubit Shor'code, presented as the earliest attempt to build a larger code by concatenating small codes [48]. The Shor's Code uses 9 physical qubits to form a single logical qubit. The code space of the shor's code contains $|\psi\rangle = \alpha|0\rangle_L + \beta|1\rangle_L$, where

$$\begin{aligned} |0_L\rangle &= \frac{1}{2\sqrt{2}} \cdot (|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) \\ |1_L\rangle &= \frac{1}{2\sqrt{2}} \cdot (|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle) \end{aligned} \tag{89}$$

Recall that the 3-qubit bit-flip code can only correct a bit-flip error (an X error). Shor's code is stronger as it can detect and correct single-qubit bit-flip, phase-flip, and bit-phase-flip errors $\{X, Y, Z\}$. Let's define this property more formally.

Definition 2.39. (Distinguishability of Pauli basis errors). A quantum code is said to distinguish all Pauli basis errors on a qubit if it produces a unique *error syndrome* for each of the single-qubit Pauli errors $\{X, Y, Z\}$.

Definition 2.39 actually admits that the quantum code can correct more errors than just $\{X, Y, Z\}$. Recall Theorem 2.7, the Pauli matrices $\{X, Y, Z, I\}$ form a basis of the 2×2 vector space. In other words, any unitary error can be seen as a combination of $\{X, Y, Z\}$ errors.[16] Hence, it is sufficient to correct *any* arbitrary single-qubit error if a code can correct $\{X, Y, Z\}$ errors. This result is known as the *digitisation of quantum errors* [46]. Hence, the distinguishability of Pauli basis errors indicates that Shor's code can correct any arbitrary single-qubit error.

To see how the Shor's code satisfies the distinguishability of Pauli basis errors (Definition 2.39), let's first only consider the first qubit as an example. That is, consider three errors $\{X_1, Y_1, Z_1\}$ where[17]

$$X_1 \coloneqq X \otimes I^{\otimes 8}, Y_1 \coloneqq X \otimes I^{\otimes 8}, Z_1 \coloneqq Z \otimes I^{\otimes 8} \tag{90}$$

[16] The identity I is not seen as an error because it has no effect. i.e. $\forall|\psi\rangle, I|\psi\rangle = |\psi\rangle$

[17] We use notation $M^{\otimes n}$ as $M \otimes ... \otimes M$ where there are n Ms.

And we now prove that there are a set of stabilisers of the Shor's that produce unique syndrome for these three errors. Consider these two stabilisers of Shor's code:[18]

$$S_1 \coloneqq XXXXXXIII, S_2 \coloneqq ZZIIIIIII \tag{91}$$

We can check if these three errors satisfies the error detecting condition (Theorem 2.32):

	S_1	S_2
X_1	$X_1S_1 = S_1X_1$	$X_1S_2 = -S_2X_1$
Y_1	$Y_1S_1 = -S_1Y_1$	$Y_1S_2 = -S_2Y_1$
Z_1	$Z_1S_1 = -S_1Z_1$	$Z_1S_2 = S_2Z_1$

Table 4: The commutivity of errors $\{X_1, Y_1, Z_1\}$ with the two stabilisers $\{S_1, S_2\}$ in the Shor's code

Recall Theorem 2.32, if an error E and a stabiliser S satisfy $ES = -SE$, then E can be detected by measuring S (measurement yields -1); otherwise, if $ES = SE$, using S cannot detect E (measurement yields $+1$). By Table 4, for error state $X_1|\psi\rangle$, we know measuring S_1 yield -1 (detectable) and measuring S_2 yields 1 (undetectable). When we apply the same reasoning on Y_1 and Z_1, it is clear that $\{X_1, Y_1, Z_1\}$ each produces an unique error syndrome. Hence, they are distinguishable and we say the Shor's code can distinguish the Pauli basis errors on the first qubit.

2.3.6 Discussion

Benefits of Stabiliser Formalism The stabiliser formalism is popular in quantum error correction. The most apparent benefit of it is that it offers more scalable reasoning for large quantum codes. Without stabiliser formalism, the reasoning processes of quantum programs needs to deal with the exponential growth of the Hilbert space — an n-qubit state needs 2^n-dimensional space (Lemma 2.8). Using Theorem 2.32, the reasoning can be carried out by using multiplication on the Pauli group, rather than analysing the unitary evolution. For groups, the length of operation grows only linearly with respect to the length of codewords. Hence, although the stabiliser formalism is originally for pen-and-paper reasoning of quantum code, We strongly believe that it might also benefit computer-assisted verification.

Limitation of Stabiliser Formalism While the stabilier formalism is popular, not all quantum codes can be applied with this framework. A notable example is the measurement-free quantum error correction (MF QEC) [24]. While stabilier codes are measurement-based, in MF QEC, a unitary recovery program replaces the non-unitary measurement-based recovery. To simulate the effect of measurement, it requires ancillary qubits and a reset operation – an oracle function that resets any qubit to $|0\rangle$.

However, the measurement-free quantum error correction (MF QEC) is primarily regarded as an alternative to the measurement-based codes. It is typically motivated by hardware constraints that there is a lack of efficient measurement capabilities on certain quantum devices [24]. Meanwhile, stabiliser codes still account for the vast majority of known and practically implemented quantum codes. One of the first experimental demonstrations showing that quantum codes can reduce error

[18]Here, we omit the proof of stabilisers for brevity. Readers can verify them or check Roffes' tutorial section 4.6 [46].

rates employed a subclass of stabiliser codes—the surface code [1]. As formal verification remains an unexplored topic, sticking to stabiliser formalism as a standard model is beneficial, and can be later extended to other non-stabiliser quantum codes.

Additional Materials Several advanced components of the stabilier formalism has been omitted for brevity. For example, we omit the concept of *code distance* and *logical operators*. For interested readers, we recommend Gottesman's thesis [19] for a thorough and foundational exposition of the stabiliser formalism.

2.4 Quantum Programming Languages

To verify any quantum program, one must understand how the program is implemented. Quantum programming languages are the essential tool to implement quantum algorithms.[19] This section discusses the two common categories of quantum programming languages: circuit-based languages and QRAM-based languages.

2.4.1 Circuit-based Programming Languages

We have mentioned the quantum circuit model in Section 2.1.3. As the most celebrated model, many foundational quantum programming languages are based on such a computation model.

First introduced on the IBM Quantum Computing Platform, the Open Quantum Assembly Language (OpenQASM) is an intermediate representation for communicating with quantum computer hardware [10]. For example, the teleportation program in Figure 1 can be described in OpenQASM Listing 1. Currently, OpenQASM and quantum circuits are widely regarded as the standard computational model for most quantum computers.

```
qreg q[3];       // initialize 3 qubits
creg c[2];       // initialize 2 classical bitsresults

// Teleportation begins
cx q[0], q[1];
h q[0];

// Measure first two qubits
// And store the result to classical bits
measure q[0] -> c[0];
measure q[1] -> c[1];

// Conditional operations based on measurements
if (c[1] == 1) x q[2];
if (c[0] == 1) z q[2];
```

Listing 1: QASM source of the quantum teleportation circuit in Figure 1.

Circuit-based programming languages can express programs that are executable on a quantum device. Hence, they hold a critical position in the area of quantum programming languages. However, some quantum algorithms go beyond plain quantum circuit models. They usually require classical

[19]Like classical algorithms, quantum algorithms can also be directly implemented at the hardware level, bypassing programming. In this thesis, we focus exclusively on implementation via quantum programming.

control flow, loops, and subroutines carried out by a classical computation process. One such example is Shor's factorisation algorithm [47], which includes a classical subroutine to compute the greatest common divisor.

2.4.2 QRAM-based Programming Languages

To solve this problem, another hybrid computation model – the Quantum Random Access Memory (QRAM) – is proposed by Giovannetti et. al [16]. In the QRAM model, a classical computer works closely with a quantum computer, and they share a different set of registers. Figure 6 shows the model visually. To solve a problem, the classical computer sends quantum instructions, usually in the form of quantum circuits, to the quantum computer. And after the quantum computer finishes the computation, the classical computer carry out a measurement to get the classical information.

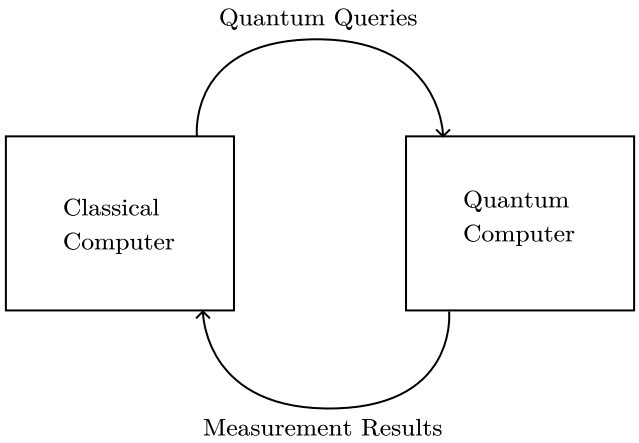


Figure 6: An illustrative image of the QRAM model

Following the QRAM computation model, a wide range of quantum programming languages that feature hybrid computation have been proposed.

One such example is the programming language Silq.[20] Silq is a quantum programming language that supports classical control flow, quantum variables and strong type checking. Hence, it can express hybrid and complicated quantum algorithms more easily than mere circuit-based programs. Listing 2 shows an example Silq program.

[20]https://silq.ethz.ch/

```
def grover[n:!ℕ](f: const uint[n] !→ lifted 𝔹):!ℕ {
  nIterations:=floor(π/(4·asin(2^(-n/2))));
  cand:=0:uint[n];
  for k in [0..n){ cand[k]:=H(cand[k]); }
  for k in [0..nIterations){
    if f(cand){ phase(π); }
    cand:=groverDiffusion(cand);
  }
  return measure(cand) as !ℕ;
}
```

Listing 2: The Silq implementation of the Grover Search Algorithm [20]. The code is originally implemented by Bichsel et. al. [5]

As shown in Listing 2, Silq programs do not explicitly construct quantum circuits. Instead, they include classic control flows like loops and conditionals, and do not impose a strict separation between quantum and classical computation. This design follows the QRAM model, where quantum and classical operations are interleaved under a unified control flow.

However, the QRAM model makes reasoning about a quantum program complicated. When only considering quantum circuits, denotational semantics suffices to describe the behaviours of a quantum program. However, when classical control flows are included, it is usually required to consider operational semantics based on the basic denotational semantics of circuits. This will introduce advanced components into reasoning, including *quantum predicates* and *quantum variables* [56]. And it will generally complicate the verification process. Hence, QRAM model is not always an optimal choice when select tools. Rather, it is encouraged to choose the programming tool on the appropriate level based on the tasks.

2.5 Formal Methods and Verification

As mentioned, the goal of this research involves *formally* verifying quantum error correction codes. This section explains the broader context of *formal methods* for two purposes:

1. To support the discussion of why formal methods are well-suited for quantum programs (Section 2.6).
2. To introduce the tool we selected for verification, namely the proof assistant Coq [4].

However, we do not discuss formal verification frameworks for quantum programs that are directly related to our work in this section. Interested readers may refer to the next chapter (Section 3.1) for those details.

Formal methods are techniques used for modelling complex systems through formal mathematical language. In software verification, formal methods can be used to verify program properties and provide correctness and safety guarantees [51]. Hence, when discussing verification, formal methods are also referred to as *formal verification*[21]. Formal verification has displayed its strength in providing

[21]This thesis focuses exclusively on formal methods as a verification means. Accordingly, the term *formal methods* strictly refers to techniques of formal verification. However, we acknowledge that formal methods also have applications beyond verification.

a correctness guarantee for critical software systems like railway schedule systems [14] and operating systems [30].

In this study, we primarily employ Coq, a proof assistant that supports *interactive theorem proving* (ITP). Hence, we place a special emphasis on ITP in Section 2.5.1. Later, we also mention *program logic*, another formal method that has been employed extensively in verification (Section 2.5.2).

2.5.1 Interactive Theorem Proving

Interactive theorem proving (ITP) is a formal method that enables users to construct proofs with the assistance of a computer (also known as a *proof assistant*) [29]. This research employs interactive theorem proving as the primary means for formal verification. Some popular proof assistant includes Coq [4] and Lean [36].

Theoretically, it is impossible to build a fully automated, sound and complete technique to verify program properties. This is the direct consequence of the Rice theorem — any non-trivial semantic properties of programs are undecidable [49]. Therefore, interactive theorem proving (ITP) is a method that trades full automation for completeness and soundness. Hence, it is well-suited for verifying programs that are small in scale but complex in semantics and program properties. This is a characteristic typical of current quantum programs.

Practically, ITP is usually more flexible and versatile for formalism and verification research. Compared to automated approaches, such as model checking [9], it enables complex verification tasks to be fulfilled under human guidance.

Another common question about ITP is why one might want a machine to check the proof. i.e. why not just write and read a pen-and-paper proof. There are a few motivations behind this. First of all, it ensures trustworthy proofs. Pen-and-paper proofs might contain incorrect application of lemmas, and it's time-consuming to manually check them. Secondly, it supports the development of complex proofs, especially when large-scale case analysis is involved. One example is that proof of the four-colour theorem is completed by enumerating 633 configurations[22] [18].

Proof Assistant Coq As mentioned earlier, Coq is a proof assistant that supports interactive theorem proving. What makes it unique is that it provides both a programming language (*Gallina*) and an interactive environment for theorem proving. This allows implementation and verification to be carried out in the same environment. Coq has been successfully applied to a wide range of verification tasks. Some notable uses include a formal proof of the four colour theorem [18] and a formally verified C compiler [32].

We selected Coq as the primary verification tool, as it is adopted by the majority of existing frameworks (Chapter 3). We will further explain and discuss Coq later in Section 4.1.

Mathematical Components: A Formal Mathematics Library in Coq Mathematical Components (short as MathComp) is a Coq library of formal mathematics [35]. MathComp was initially developed for the formal proof of the Four-Colour Theorem [18], and was later applied to other areas of formal verification, including the odd order theorem [17].

[22]Informally, a *configuration* is a solvable pattern that is generated by case analysis.

MathComp is an extensive library that covers data structures and algebraic structures. For algebra structure, type `{group gT}` is a group with elements of type `gT` and type `ringType` is the interface type for a ring structure. For example, type `seq T` is a list of type `T` and type `{set T}` is a finite set of type `T`

MathComp provides this research with a set of trustworthy reasoning tools for algebraic structures. We will further explain and discuss MathComp later in Section 4.2.2.

2.5.2 Program Logic

Another popular formal method is *program logic*, which originates from the celebrated Hoare logic system [27]. The core of Hoare logic is the Hoare triple:

$$\{P\}C\{Q\} \tag{92}$$

In a Hoare triple, the P and Q are the precondition and postcondition expressed in first-order logic, and C is a command (usually a line of code). For example, we can say the following Hoare triple is valid:

$$\{x > 0\}x := x + 1; \{x > 1\} \tag{93}$$

This triple expresses how the precondition $\{x > 0\}$ evolves into the postcondition after executing the code. According to the assignment axiom:

$$\text{Assignment}\ \frac{}{\vDash \{\psi[E/x]\}x := E\{\psi\}}$$

Figure 7: The assignment axiom in Hoare Logic

in which $\psi[E/x]$ denotes the formlua obtained by taking ψ and replace all free occurrences of x with E. A complete set of inference rules and correctness proof can be found in [29] on Page 269. Hoare logic was later extended to programs containing pointers (separation logic) [45], concurrent programs [38], and more.

These axiomatic approaches to verify program behaviours are often referred to as ***program logic***. We will discuss some program-logic-based verification of quantum programs later in Section 3.2.

2.6 Key Insights from Background

Debugging and Testing is Complicated for Quantum Programs When discussing measurment (Section 2.1.2), we have mentioned how it complicates testing and debugging. For instance, it is impossible to extract the complete information from a state in superposition due to the limitation of measurement. Futhermore, other properties of quantum computing also contribute to this challenge. For example, the no-cloning theorem implies that it is impossible to run the same test multiple times under the very same conditions. Another difficulty of testing quantum programs is that the computation resources are scarce and expensive [42]. While statistic-based testing is viable, noise could further complicate it, as it's hard to trace whether the cause of unexpected result is due to environment noise or bugs in programs.

Quantum Programs are Amenable to Formal Verification Unlike testing, nothing prevents applying formal verification to a quantum program. Formal verification does not rely on observing intermediate program states or program output, nor does it require execution on a real quantum computer. Further, as discussed in Section 2.1, quantum programs are typically associated with precise denotational semantics [33]. This contrasts with classical programs, whose meanings are often informally understood and defined through operational behaviour. This characteristic makes formal verification particularly suitable, as quantum programs are inherently defined in a formal and mathematical manner.

Applying formal methods to quantum programs also provides other unique advantages. Firstly, it allows verifying quantum programs with generalised parameters, i.e. higher-order quantum programs. In contrast, testing-based verification only provides correctness evidence for a specific set of inputs. Formal methods also enable the development of rigorous foundations for quantum programming languages. For example, QWire [40] is equipped with a type system that is formally proved to enforce the non-cloning property. This kind of guarantee cannot be achieved through testing-based verification.

The Importance of Formal Verification for Quantum Error Correction Achieving fault-tolerant quantum computing is a milestone of current quantum computing researcb [42]. Quantum error correction codes have long been regarded as a promising technique toward this objective[46], and recent empirical studies have demonstrated their potential to suppress error rates[1]. Consequently, developing efficient methods for verifying quantum codes directly supports the broader aim of building fault-tolerant and trustworthy quantum computing systems.

Meanwhile, quantum codes are not trivial. Rather, they have rich properties and representations. Verifying them validates that formal verification can be applied to non-trivial quantum programs.

Stabiliser formalism is well-suited for verifying quantum codes. As discussed in Section 2.3.6, the stabiliser formalism enables reasoning of quantum code properties based on a key algebraic structure — the Pauli group — instead of conventional reasoning using the Hilbert space representation. Typically, while a n-qubit quantum code needs 2^n dimensional Hilbert space for representation, it is a Pauli group of length n if we use the stabiliser formalism. This can be seen as a logarithmic reduction in dimension. Given its success in pen-and-paper reasoning, we have strong confidence in hypothesising that, stabilier formalism can also achieve an efficient formal verification of quantum codes.

Chapter 3

Related Work

We have discussed the broader context of this research in Chapter 2. In contrast, this chapter discusses existing works that are closely related to our research question:

"Whether formal verification of quantum error correction codes can be efficiently achieved by stabiliser formalism?"

We first discuss existing verification frameworks of quantum programs. Based on previous discussion on quantum programming languages (Section 2.4), we also categorise verification frameworks by their underlying computation model: circuit-based and quantum random access memory (QRAM) based verification. For circuit-based verification, we select the verification framework SQIR as an example. SQIR and its underlying mathematical library, QuantumLib, have the strongest impact on our work (Section 3.1.1). While not directly involved, we also select a QRAM-based verification framework CoqQ for comparison.

Then we turn to the verification work of quantum error correction codes (Section 3.3). While this topic is rather under-explored, a few existing efforts offer valuable insights.

3.1 Verification of Circuit-based Programs

3.1.1 SQIR: Formal Language for Circuit Programs

A celebrated tool for verifying low-level quantum programs is SQIR. The name SQIR stands for Small Quantum Intermediate Representation, because it provides a formal language for expressing low-level circuit programs, and hence supports verification. SQIR is both implemented and verified in the proof assistant Coq (See Section 2.5.1). This allows users to implement and verify SQIR programs in Coq.

Let's consider an example usage of SQIR — optimization of quantum circuits. One simplest optimization principle is that, between two semantically equivalent programs, the one with fewer instructions is usually more efficient. Figure 8 shows one such equivalence relation.[23]

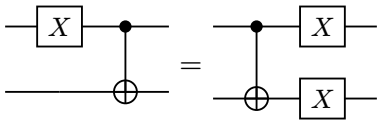


Figure 8: The two circuits are of the same unitary evolution.

[23]This example in Figure 8 comes from the circuit optimizer (VOQC) [26], which employs SQIR as the language to express circuits.

We first express the two programs in Figure 8 using SQIR. SQIR features a syntax aligned with OpenQASM [10], enabling easy representation of circuit programs. SQIR is also equipped with a denotational semantics — a program is interpreted as a unitary evolution.[24] Hence, Figure 8 can be expressed as the equivalence of two SQIR programs Listing 3.

```
X 0;
CNOT 0 1
```
≡
```
CNOT 0 1;
X 0; X 1;
```

Listing 3: The equivalence relation in Figure 8 expressing in SQIR, in which ≡ stands for equivalence of semantics

To understand Listing 3, let's first start with syntax of SQIR:

```
Inductive ucom (U: N → Set) (d: N) : Set :=
    | useq: ucom U d → ucom U d → ucom U d
    | uapp1: U 1 → N → ucom U d
    | uapp2: U 2 → N → N → ucom U d
```

The type name `ucom` stands for "unitary commands". For the SQIR program in Listing 3, the semicolon "`_ ; _`" is the notation for the constructor `useq`, which concatenates two `ucom` programs. The constructors `uappn` stands for applying a n-qubit quantum operator U to the indexed qubits. For example, `(H 0)` in Listing 3 is a notation for `(uapp1 H 0)`. Hence, without notation, the program on the left of Listing 3 is

```
(* The SQIR Program: X 0; CNOT 0 1 *)
useq(
  uapp1(X, 0),
  uapp2(CNOT, 0, 1)
)
```

There are a few noticeable designs in its syntax:

- Qubits are globally indexed by natural numbers. SQIR is designed intentionally without the notion of variables to simplify verification.
- The `ucom` type accepts a parameter $U : \mathbb{N} \to$ `Set`, a list of support quantum operators. Here we use the default operator set `base` provided alongside SQIR, which contains common operators including `X` and `CNOT` used in the program.

We now shift our focus from syntax to semantics. The denotational semantics of SQIR program can be interpreted by function `uc_eval`.

```
Fixpoint uc_eval {d}: (ucom d)  → Matrix (2^d) (2^d)
```

The definition of `uc_eval` implies that any d-dimensional unitary SQIR program is interpreted as a $2^d \times 2^d$ matrix, which is a unitary evolution on d qubit systems.

[24]The interpretation from a program into a unitary evolution is called the *unitary semantics*. SQIR also support *density matrix semantics*, which is for non-unitary quantum programs. However, non-unitary semantics is rather complicated and will not be used in this research. Readers interested in it might want to check [37] and [25]

Since SQIR is implemented in Coq, we can formally express the equivalence (Figure 8) as a Coq theorem.

```
Theorem equivalence_x_cnot:
  uc_eval (X 0; CNOT 0 1) = uc_eval (CNOT 0 1; X 0; X 1).
```

This theorem expresses an equivalence between the quantum circuits shown in Figure 8, meaning they have the same denotational semantics — i.e., they produce the same unitary matrix.

3.1.2 QuantumLib: Formal Mathematical Library for Quantum Computing

Note that in the interpretation function `uc_eval` of SQIR, a Matrix type is used to represent the Hilbert space. This matrix library comes from QuantumLib, a Coq library that contains mathematical structures to support reasoning about quantum computing processes [40]. As it closely aligns with our goal, our work builds heavily upon QuantumLib. We give a brief overview of QuantumLib in this section. And later we will further discuss QuantumLib when it is used in our formalism.

QuantumLib is based on Coquelicot's real analysis for reasoning on real numbers $\mathbb{R}$ [6]. The complex number $\mathbb{C}$ is defined as a pair of real numbers.

```
Definition C := (R * R).
```

Here operator `*` is the Cartesian product.

Having complex numbers, QuantumLib formalises the Hilbert space with the key structure — complex matrices.

```
Definition Matrix (m n : N) := N → N → C.
```

Here, a matrix is represented as a function that takes a pair of indices and returns a complex number. The type parameter m and n records its dimension. Unlike the commonly used nested array representation, this functional encoding allows for more flexible and concise construction, particularly suitable for sparse and generic matrices. For example, the identity matrix I of general d dimension can be implemented as:

```
Definition I (d: N): Matrix d d :=
  (fun x y ⇒ if (x == y) && (x < d) then 1 else 0).
```

Operations on matrices are the core features of QuantumLib. We will discuss them further later in this thesis. Interested readers can check Section 4.2.1 for details.

3.1.3 Example: Verifying Circuit Program SWAP in SQIR

After introducing SQIR and QuantumLib, let's consider a more concrete and trackable example of quantum program verification.

Consider the program f_{swap} that swaps two quantum states. i.e. for any two quantum states $|\psi\rangle$ and $|\varphi\rangle$,

$$f_{\text{swap}} : |\psi\rangle \otimes |\varphi\rangle \mapsto |\varphi\rangle \otimes |\psi\rangle \tag{94}$$

Note that the tensor product is not commutative. i.e. $|\psi\rangle \otimes |\varphi\rangle \neq |\varphi\rangle \otimes |\psi\rangle$

f_{swap} is a foundational component for many complex quantum algorithms. Consider two single-qubit states, f_{swap} can be constructed by the following circuit according to [37],

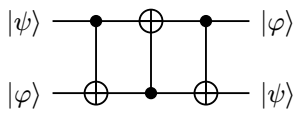


Figure 9: The implementation of swap program f_{swap} on two single-qubit states.

How do we know that this implementation is correct? In other words, how do we prove the circuit program (Figure 9) satisfies the correctness property (Equation 94)?

Firstly, we can do this proof by pen-and-paper. As shown in Figure 9, the program has the denotational semantics as:

$$f_{\text{swap}}(|\psi\rangle \otimes |\varphi\rangle) := \text{CNOT}_{01}\text{CNOT}_{10}\text{CNOT}_{01}(|\psi\rangle \otimes |\varphi\rangle) \tag{95}$$

Where the control qubit and the applied qubit are notated as subscripts on CNOT.

Let $|\psi\rangle = \alpha\ |0\rangle + \beta\ |1\rangle$ and $|\varphi\rangle = \gamma\ |0\rangle + \delta\ |1\rangle$, then

$$\begin{aligned} |\psi\rangle \otimes |\varphi\rangle &= (\alpha\ |0\rangle + \beta\ |1\rangle) \otimes (\gamma\ |0\rangle + \delta\ |1\rangle) \\ &= \alpha\gamma\ |00\rangle + \alpha\delta\ |01\rangle + \beta\gamma|10\rangle + \beta\delta\ |11\rangle \end{aligned} \tag{96}$$

By definition, CNOT_{ab} (controlled-not) gate flips the target qubit b if the controlling qubit a is $|1\rangle$; And it leaves the target qubit unchanged if the controlling qubit is $|0\rangle$. Using this definition of CNOT_{ab} and Equation 96, we can simplify and rewrite the semantics of the program as

$$\begin{aligned} f_{\text{swap}}(|\psi\rangle \otimes |\varphi\rangle) &:= \text{CNOT}_{01}\text{CNOT}_{10}\text{CNOT}_{01}(\alpha\gamma\ |00\rangle + \alpha\delta\ |01\rangle + \beta\gamma|10\rangle + \beta\delta\ |11\rangle) \\ &= \text{CNOT}_{01}\text{CNOT}_{10}(\alpha\gamma\ |00\rangle + \alpha\delta\ |01\rangle + \beta\gamma|11\rangle + \beta\delta\ |10\rangle) \\ &= \text{CNOT}_{01}(\alpha\gamma\ |00\rangle + \alpha\delta\ |11\rangle + \beta\gamma|01\rangle + \beta\delta\ |10\rangle) \\ &= \alpha\gamma\ |00\rangle + \alpha\delta\ |10\rangle + \beta\gamma|01\rangle + \beta\delta\ |11\rangle \\ &= (\gamma\ |0\rangle + \delta\ |1\rangle) \otimes (\alpha\ |0\rangle + \beta\ |1\rangle) = |\varphi\rangle \otimes |\psi\rangle \end{aligned} \tag{97}$$

which completes the proof of the correctness property (Equation 94). How do we know the proof is correct? We can, of course, spend some time manually checking that each step of the proof is correct. But we can also have a more efficient and rigorous way, that is, by computer-assisted proof using SQIR.

We first define the program `SWAP`. SQIR is embedded in proof assistant Coq [4], so we can define the program as a Coq function.

```
Definition SWAP a b := CNOT a b; CNOT b a; CNOT a b.
```

Then we declare the correctness property as a theorem.

```
Theorem swap_correct: ∀ (psi phi: Vector 2),
  (eval SWAP) × (psi ⊗ phi) = (phi ⊗ psi).
```

Theorem `swap_correct` declares that for all single-qubit quantum states $|\psi\rangle$ and $|\varphi\rangle$, evaluating program `SWAP` on the input $|\psi\rangle \otimes |\varphi\rangle$ equals $|\varphi\rangle \otimes |\psi\rangle$. Recall Section 2.1, a single-qubit quantum state is represented as a 2-dimensional vector. Therefore, the type of $|\psi\rangle$ is a `Vector 2`.

Finally, we write a proof script that proves the `SWAP` program meets the specification.

```
Proof.
  intros psi phi. (* introduce psi and phi from context *)
  simpl eval. (* unfold and simplify the evaluation of SWAP *)
  (* rewrite using the properties of CNOT to simplify the equation *)
  autorewrite with eval_db; simpl; try lia.
  solve_matrix. (* check both sides of the matrix equation are equal *)
Qed.
```

This proof is written in the Ltac proof language, which is the default proof language of Coq [4].[25]

3.2 Verification of QRAM-based Programs

As discussed in Section 2.4, some quantum algorithms are constructed with recursion and classical subroutines, which is more than mere quantum circuits. This class of programs is based on the Quantum Random Access Memory (QRAM) model. We now discuss some of them for comparison to circuit-based verification in SQIR.

3.2.1 Quantum Hoare Logic

Quantum programs based on the quantum random access memory (QRAM) model usually contain classical imperative control flows. Hence, using denotational semantics to express it will be rather complicated. Instead, most QRAM-based programming languages lie Silq [5] adopt *operational semantics*, where program execution is defined by step-by-step state transitions.

Recall Section 2.5.2, one such system for verifying operational semantics is the *Hoare Logic*. For QRAM-based programs, Hoare logic is extended into *Quantum Hoare Logic* (QHL) [54, 56]. Specifically, the quantum Hoare logic is a set of inference rules that supports reasoning about the correctness of quantum programs. It has a similar structure to the classical Hoare logic (Section 2.5), with the form $\{A\}C\{B\}$. What makes QHL different is that, instead of first-order logic, the precondition and postcondition are expressed as linear operators. Take the assignment rule in [56] as an example (Figure 10).

$$\text{Ax.InF'}\ \frac{S \cap \text{set}(x) = \emptyset}{\vDash \{|v\rangle_S\}\ x := |t\rangle\ \{|v\rangle_S \otimes |t\rangle_x\}}$$

Figure 10: The assignment rule of the quantum Hoare logic [56]

The inference rule "Ax.InF'" stands for "axiom of initialisation formula, alternative definition" [55]. rule "Ax.InF'" stands for "axiom of initialisation formula, alternative definition" [55]. It states that

[25] We omit some details of this proof for brevity. The actual proof can be found on https://github.com/ExcitedSpider/SQIR/blob/main/examples/examples/Thesis2025.v

the assignment command $x := |t\rangle$ adds state $|t\rangle$ into the original quantum state $|v\rangle$ by tensor product. This reflects the mathematical representation that quantum states (subsystems) form a larger system by tensor product (Section 2.1). The premise $S \cap \text{set}(x) = \emptyset$ ensures that the original quantum state $|v\rangle$ has no intersection with the subsystem that x refers to. The S in subscript $|v\rangle_S$ denotes that S is the subsystem that $|v\rangle$ lives in. To get more intuition, we encourage readers to compare this inference rule with the classical assignment rule (Section 2.5).

3.2.2 CoqQ: Mechanized Verification of QRAM Programs

Based on quantum Hoare logic, the formal verification framework CoqQ [55] is able to verify QRAM-based quantum programs. Using CoqQ, programs can be expressed in a more expressive syntax that resembles classical programming languages like C, and quantum computing is incorporated with classical control flow.

As mentioned earlier, it takes its logical foundation from extending the traditional Hoare logic to *Quantum Hoare Logic* (QHL) [56], which incorporates quantum subroutines into traditional procedural programming languages. One example of a high-level quantum program is listed in Listing 4.

```
Definition Grover (r: N) :=
  x := |t0⟩;
  x := Hn[x];
  for i < r do (
    x := Hn^A[x];
    x := PhOracle(f)[x];
    x := Hn[x];
  )
```

Listing 4: An example quantum program (The Grover's Search Algorithm) in CoqQ

This program in (Listing 4) is an implementation of Grover's database searching algorithms [20] using CoqQ. Without discussing correctness, we can already notice some differences from SQIR by examining the program's construction. First of all, it does not explicitly construct the quantum circuit. And it uses a variable `x` to represent quantum states, instead of globally indexing by number in SQIR. It contains loops and subroutines. These features make CoqQ similar to traditional procedural programming languages like C, but with a special type system that is compatible with quantum states.

Like SQIR (Section 3.1.1), CoqQ is also both implemented and verified in proof assistant Coq. This allows SQIR program verification to be directly carried out in Coq. The verification of the Grover algorithm in Listing 4 is presented in section 8.7 of Zhou et.al [55].

3.3 Verification of Quantum Error Correction Codes

This section selects works that are related to the formal verification of quantum codes.

3.3.1 Certified Quantum Codes Examples In SQIR

SQIR comes with a few examples of certified quantum programs.[26] Among the literature, these examples are closely related to our research question. We refer to these certified quantum code

[26]https://github.com/inQWIRE/SQIR/tree/main/examples/error-correction,

examples provided in the SQIR repository as *SQIR-QECC examples*. In these examples, some important correctness properties of a few small-scale quantum codes have been verified.

- The encoding, decoding and error-correcting ability of the three-qubit bit-flip code (Section 2.2.2).
- The encoding, decoding of the nine-qubit Shor's code (Section 2.3.5).

These examples show the feasibility of verifying quantum error correction codes. And we are greatly motivated by them.

These codes are implemented based on measurement-free quantum error correction (MF-QEC)[24]. Recall Section 2.3.6, MF QEC is primarily regarded as an alternative technique to the standard, measurement-based QEC. But formal verification does not need realisation on a quantum device, and we have no evidence why MF QEC is employed by them. From the result, using MF QEC makes the implementation avoid discussing non-unitary semantics. Reasoning about non-unitary semantics is generally more challenging than unitary semantics in all formal frameworks of quantum computing, because the involvement of density matrix representation [8]. In addition, MF QEC is designed only for error-correcting codes, and has no mechanism to provide proof of detecting-only codes.

These certified quantum code examples are primarily intended to demonstrate the capabilities of SQIR, rather than to establish reusable theories for quantum error correction codes. Hence, we notice some potential improvement.

The first issue is the scalability. For example, to reason about properties of a 9-qubit code, it requires a $2^9 = 512$ dimensional Hilbert space, which makes it difficult to manage. Scalability is also the reason why the verification of the 9-qubit code does not involve an error correction procedure (recovery). Because reasoning recovery needs an additional 8 qubits following the measurement-free quantum error correction, which makes the verification impractical. We will further discuss the scalability limitations in SQIR-QECC examples when making comparisons (Section 6.1.5).

Another issue with the SQIR-QECC examples is that they are not designed to be reusable. This is because the goal is to serve as an example of SQIR, and not to provide a general abstraction of quantum codes. As a result, the formalism is constructed or applied as needed for individual cases, without a unifying theory. In other words, these examples are case-by-case, and there is no generalised formalism. To illustrate, notice that there are no reused predicates or properties between the 3-qubit code and the 9-qubit code.

This observation strongly motivates us to pursue a more scalable and general verification approach. That said, we highly value the SQIR-QECC examples — not only do they investigate the feasibility of verifying quantum codes, but also serve as a crucial baseline for assessing the benefits of our Coq-QECC-based verification. We will compare our Coq-QECC to them in Section 6.1.5.

3.3.2 Other Verification Work of Quantum Error Correction Codes

Some other work is intended to verify quantum codes via alternative means. While these may not be directly aligned with our approach, we include them to provide a broader perspective on the field and to position our contribution within the existing landscape.

In 2021, Wu et. al. presented a framework QECV, to build and verify quantum code [52]. QECV employs an assertion language to express predicates about stabilisers. However, this work is not implemented in any existing programming languages or proof assistant, and only serves as a theoretical framework to describe and reason about QEC programs.

Recently, Fang presented a framework for symbolic analysis of QEC programs QSE [11]. QSE is based on the symbolic execution of quantum programs. Unlike ITP, verification through symbolic execution is neither static nor complete. Instead, it finds counterexamples by executing the program with symbolic variables. This marks a major difference from our work.

There are also other computational models instead of quantum circuits. Chancellor et. al. employed a graphical language based on categorical theory, namely ZX-Calculus [7]. The graph language can also be employed to model and verify quantum programs. But we focused on the circuit representation as it remains the standard for most quantum hardware.

3.4 Discussion and Summary

Scalability Problem of Verification Scalability is a major problem for verifying quantum programs. The dimension of unitary semantics grows exponentially as the size of qubits grows (Lemma 2.8). For example, a 3-qubit state $|000\rangle$ is represented as a vector of size $2^3 = 8$,

$$\begin{aligned} |000\rangle &= |0\rangle \otimes |0\rangle \otimes |0\rangle \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \in \mathbb{C}^8 \end{aligned} \tag{98}$$

Accordingly, any operation on this 3-qubit state should be represented by a unitary matrix of size 8×8. This implies that the reasoning should be carried out in an 8-dimensional vector space. This exponential growth presents a major scalability problem for both pen-and-paper proofs and computer-assisted proofs.

Selection of Tools This study selects circuit-based verification framework SQIR (Section 3.1.1) for investigation. Recall our goal involves formal verification of quantum error correction codes (quantum codes). Quantum codes are fundamentally based on circuits. Hence, SQIR is well-suited for our investigation.

We also adopt Quantumlib (Section 3.1.2) for mathematical structures, including (the ring of) complex numbers, matrices in Hilbert space, etc. As a formal mathematical library dedicated to reasoning about quantum computing, QuantumLib is also a handy tool for this research.

In addition to QuantumLib, we adopt Mathematical Components (MathComp) (See Section 2.5.1). This is because QuantumLib cannot cover all our needs for formal mathematical structures. Specifically, we use MathComp mainly for the algebraic structures of groups and the data structure of tuples (fixed-length lists). We are also adviced to adopt MathComp when presenting our work-in-progress on CoqPL25 workshop [13].

Summary In this chapter, we have reviewed the closely related work of our research question —

"Whether formal verification of quantum error correction codes can be efficiently achieved by stabiliser formalism?"

The first insight is about verification frameworks. Both circuit-based and QRAM-based verification techniques are developed in the field (Section 3.1 and Section 3.2). We choose to use circuit-based verification framework SQIR based on the characteristics of quantum codes — They are implemented at the circuit level and their properties are denotational (See Section 2.2).

Secondly, we observe that existing formal verification approaches for quantum codes face scalability limitations (Section 3.3.1), even on small scale quantum codes of 9 qubits. This limitation further motivates our introduction of the stabiliser formalism, which aims to simplify and scale the verification process.

These findings, together with the discussion on why quantum codes are important to verify Section 2.6, make we believe our research question remains valuable. In following chapters, we will move on to discussing the investigation, including formalism of quantum stabiliser code theory and verification of a few selected quantum codes in proof assistant Coq.

Chapter 4

Formalism of Pauli Groups in Coq

We now shift our focus from related work to implementation. Pauli groups are the foundational algebraic structure in the stabiliser formalism. This chapter discusses the formalism of the stabiliser formalism. We implement and verify the Pauli group both in the theorem prover Coq [4] This is possible because Coq contains a functional programming language, an interactive environment for building theorems and proofs.

We first give a short introduction to Coq Section 4.1. Experienced readers can safely skip this part. Then we further explain the two most important libraries that we take advantage of — The QuantumLib and MathComp — in section Section 4.2. Then we turn to discuss the implementation and verification of Pauli groups in Coq. Based on the certified Pauli groups, we build group actions over them.

This chapter is based on the theory of Pauli group introduced in Section 2.3.1.

4.1 A Short Introduction to Coq

Before we begin the discussion, it is helpful for us to give a short introduction to Coq. For readers who are familiar with Coq, they can safely skip this section. We only assume readers have basic knowledge of functional programming.

4.1.1 The Programming Language Gallina

Coq comes with a programming language, named *Gallina*. Gallina follows a purely functional paradigm, which implies that no side effects are allowed in Gallina. In addition, Gallina features algebraic data types and pattern matching. For example, the Boolean type is defined as[27] :

```
Inductive bool : Type := true | false.
```

Where `true` and `false` are referred to as constructors of the inductive type `bool`. Functions over data types are built based on pattern matching.

```
Definition negb (b : bool) :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.
```

We can define a function based on case analysis Types are “first-class citizens” in Gallina, which means we can use types as parameters to construct another datatype. For example, the Cartesian

[27]These basic data types, including `bool`, `prod` and `list`, come from the library Coq.Init.Datatypes: https://rocq-prover.org/doc/v8.9/stdlib/Coq.Init.Datatypes.html

product type `prod` is defined using two `Type` parameters. Data types can also be defined recursively. One such example is the `list` type. The `list A` is a sequential list of elements of `A`. A list is constructed either as an empty list (`nil`), or a concatenation of an element of `A` with an existing list.

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

To ensure totality, Gallina requires that all functions must terminate. This implies that there are no general loops allowed. Instead, Gallina requires that any recursive function provide evidence that it will terminate.

```
Fixpoint length (A : Type) (l: list A): N :=
  match l with
   | nil ⇒ 0
   | _ :: l' ⇒ 1 + (length l')
  end.
```

The function `length` checks the length of a list[28]. The keyword `Fixpoint` is a label of recursive functions. This is a valid recursive function because each step destructs the input list to a smaller one.

At last, Gallina features dependent types. A small but non-trivial example is the vectors, which is a list of size n whose elements belong to a set `A`.

```
Inductive vector A : N → Type :=
  | nil : vector A 0
  | cons : ∀ (h:A) (n:N), vector A n → vector A (S n).
```

For example, we can define a byte as a list of booleans with length 8,

```
Definition byte := vector bool 8.
```

Unlike a simple `list bool`, a value of `byte` must have exactly 8 elements. Defining any `byte` without giving 8 elements with result in an error[29].

```
Fail Definition nibble: byte := [true; true; false; false].
```

This design choice gives the immediate benefit that one can define functions with a desired length just by using types. For example, for a `map` function, we can encode the fact that the return value is of the same length as the input just by typing:

```
Definition map {A B: Type} {n:N}: (A → B) → vector A n → vector B n.
```

where `A` and `B` are implicit parameters that represent any two types, and `n` is the length of the list.

At last, Gallina can also be used for specification. Recall that the De Morgan's law of negation on "A or B" is

[28]The ℕ represents natural number data types.
[29]The `Fail` command in Coq checks any statements following it indeed fail.

$$\forall a \forall b, \neg(a \vee b) = \neg a \wedge \neg b \tag{99}$$

This translates to Gallina as:

```
Lemma negb_orb : ∀ b1 b2:bool, negb (b1 || b2) = negb b1 && negb b2.
```

4.1.2 The Tactic Language and Interactive Theorem Proving

As mentioned, Coq contains an interactive environment for building theorems and proofs. It means that Coq comes with a proof language, namely *tactic*. Continuing the example of De Morgan's law, we can prove that it is correct.

```
Lemma negb_orb : ∀ b1 b2:bool, negb (b1 || b2) = negb b1 && negb b2.
Proof. (* this tells Coq the start of tactic language scope *)
  intros b1 b2. (* move the universally quantified variable b1 and b2 into context *)
  unfold negb. (* replace `negb` with its definition *)
  destruct b1, b2; (* perform case analysis based on b1 and b2  *)
  simpl; (* simplify decidable expressions by evaluating them *)
  reflexivity. (* closes goals of the form x=x *)
Qed. (* Close the tactic language scope and triggers the proof checker *)
```

Every `Qed.` commands check the proof built by the tactic language and ensures logical consistency and type correctness. If any part of the proof is inconsistent, Coq will throw an error and will not accept it.

The theorem-proving environment is interactive, which means that the evaluation of proof scripts written in tactic can be paused at every full stop `.` ; And during the pause, users can view the current goals and context. This allows users to build proof by interactively getting feedback from the theorem prover.

4.1.3 Additional Materials

This section does not aim to comprehensively introduce Coq. For readers interested in exploring Coq further, the following resources are recommended:

- jsCoq provides an interactive, in-browser environment for experimenting with Coq [15][30].
- The *Software Foundations* textbook offers a structured and in-depth introduction to Coq and its foundational principles [41].
- The Coq Reference Manual serves as the authoritative source for tactic descriptions and the Gallina specification [4].

4.2 Dependent Libraries

This section introduces the libraries and axioms that the Coq formalism relies on.

4.2.1 QuantumLib

As mentioned in Section 3.1.2, QuantumLib is a Coq library for reasoning about quantum programs (Section 3.1.2).

[30] As of May 2025, a live version is available at https://coq.vercel.app/

QuantumLib is built based on the Coq real analysis library Coquelicot [6], which provides real numbers $\mathbb{R}$ and a set of tools to reason about goals involving $\mathbb{R}$.

Then, the complex number ($\mathbb{C}$) is defined as the Cartesian product of two real numbers. The operations on complex numbers are carried out by projection, for example:

```
(* Complex Number *)
Definition C := (R * R).

(* Multiplications of two complex numbers *)
Definition Cmult (x y : C) : C := (
  fst x * fst y - snd x * snd y,
  fst x * snd y + snd x * fst y
).
```

With with basics of $\mathbb{C}$, QuantumLib builds a comprehensive formalism of linear algebra and finite Hilbert space. Matrices are defined as:

```
Definition Matrix (m n : N) := N → N → C.
```

That is, a matrix is a function that takes two natural numbers, returns a complex number as the value at that cell. This definition has an apparent limitation – although m and n are put into parameters, there is no means for Coq to prevent users from defining a matrix larger than $m \times n$. For example, this is a definition that is well-typed with respect to Coq's type system:

```
Definition M22 : Matrix 2 2 :=
  fun x y ⇒
  match (x, y) with
  | (0, 0) ⇒ 1
  | (0, 1) ⇒ 2
  | (1, 0) ⇒ 4
  | (1, 1) ⇒ 5
  | (1, 2) ⇒ 6
  | _ ⇒ 0
  end.
```

Since it is a 2×2 matrix, index (1,2) is physically invalid. However, it cannot be found by type-checking. To solve this, `n` and `m` are included as the bound using a technique named *phantom types* [43]. That is, instead of using dependent type techniques to check, QuantumLib provides a predicate `WF_Matrix` to ensure that a matrix is only nonzero within the bounds.

```
Definition WF_Matrix {m n: N} (A : Matrix m n) : Prop :=
  ∀ x y, x >= m ∨ y >= n → A x y = 0.
```

For example, to prove two matrices are equal, one must show that both matrices are well-formed:

```
(* Checking each cell is equal within the bound *)
Definition mat_equiv {m n : N} (A B : Matrix m n) : Prop :=
  ∀ i j, i < m → j < n → A i j = B i j.

Lemma mat_equiv_eq : ∀ {m n : N} (A B : Matrix m n),
  WF_Matrix A → WF_Matrix B → mat_equiv A B → A = B.
```

Based on this formalism of matrices, QuantumLib builds many useful quantum operations, such as tensor products, Dirac bracket notation and padding functions. QuantumLib also provides a wide range of lemmas and automation tools for proving facts about matrices, complex numbers and real numbers. Tactic `lma` and `solve_matrix` prove equality between matrices. Both of them are built based on `lca`, which applies `lra` to solve linear real arithmetics by projection [6].

4.2.2 Mathematical Components

MathComp is a Coq library for formal mathematics. It employs a hierarchical and compositional way of building mathematical objects. For example, Listing 5 is the definition of a finite multiplicative group (`isMulGroup`), which corresponds to Definition 4.1.

```
Record isMulGroup G of Finite G := {
  mulg : G → G → G;
  oneg : G;
  invg : G → G;
  mulgA : associative mulg;
  mul1g : left_id oneg mulg;
  mulVg : left_inverse oneg invg mulg;
}.
```

Listing 5: Definition of a finite multiplicative group in MathComp

To understand Listing 5, recall the mathematical definition of finite groups:

Definition 4.1. A finite group is a finite set G together with a multiplication $\cdot$ defined on G, in which the following properties are satisfied:

1. Closure. $\forall a, b \in G, ab \in G$
2. Associativity. $\forall a, b, c \in G, abc = a(bc)$
3. Identity. There is an identity element e such that $\forall a \in G, ea = ae = a$
4. Inverse. There is an inverse element for each element in G, i.e. $\forall a \in G, a \cdot a^{-1} = a^{-1} \cdot a = e$

In Listing 5, `isMulGroup` is a constructor of a multiplicative group: a group must operate on a finite set `G`, with three components defined: a multiplication function `mulg`, an identity element `oneg` and an inverse function `invg`. Additionally, it requires proofs that the multiplication operation is associative (`associative`), the identity element is a left identity (`left_id`), and the `invg` defines a left inverse (`left_inverse`) for `mulg` and `oneg`. Hence, this formalism (Listing 5) basically formalises the standard definition of group in Definition 4.1.[31]

[31] One difference of the theory (Definition 4.1) and the formalism (Listing 5) is that the later only requires users to provide proofs of left identity and left inverse. This is because MathComp proves the right identity and right inverse internally.

Another feature of MathComp is that its extensive use of canonical structures. Canonical structures in Coq can automatically perform type unification, and thus improve the automation level [4]. Although does not affect the design of our formalism, this feature mainly helps us to ease difficulty in theorem proving. We will discuss an example of using canonical structure to simplify proving in Section 4.3.3.

4.3 Formalising Pauli Groups

This section discusses the implementation of certified Pauli groups. The corresponding Coq code is in the code repository of Coq-QECC as PauliGroup.v. We strongly recommend that readers check the Coq code when reading this section.

4.3.1 Inductive Data Types and Interpretation to Matrices

According to Definition 2.23, any elements of Pauli groups are essentially matrices in Hilbert space. It is also technically possible to replicate this pen-and-paper definition in Coq, as QuantumLib has provided the formalism of matrices (Section 3.1.2). However, we choose not to directly involve matrices. Instead, we use Coq's inductive data type in formalism and build decidable interpretation functions to translate them into QuantumLib's Matrices.

For example, we define the basic Pauli operator as an inductive data type:

```
Inductive PauliBase : Type :=
| I : PauliBase
| X : PauliBase
| Y : PauliBase
| Z : PauliBase.
```

And an interpretation function is attached to the definition:

```
Definition p1_int (p : PauliBase) : Square 2 :=
match p with
| I ⇒ Quantum.I 2
| X ⇒ Quantum.σx
| Y ⇒ Quantum.σy
| Z ⇒ Quantum.σz
end.
```

Here, the module prefix `Quantum` represents that they are definitions from QuantumLib. The name `p1_int` represents that this is the interpretation function of the single-qubit Pauli group.

This design choice is based on the rationale that operations on matrices are more costly than those on inductive data types. Remember Lemma 2.8, an n-qubit system needs 2^n dimensional Hilbert space for representation. If Pauli groups are formalised in the form of matrices, they will also follow Lemma 2.8 to grow exponentially in size.

Finally, we write $[\![x]\!]$ as interpreting any inductively defined data type x to matrices in Hilbert space.

4.3.2 Overview of Implementation

To ease the difficulty in implementation, we use a bottom-up approach to formalise the Pauli groups. Starting from the basic single-qubit Pauli group with fixed +1 phase, we extended it incrementally to the n-qubit Pauli group with $\{\pm 1, \pm i\}$ phases (see Section 2.3.1). The hierarchy of the formalism is presented in Figure 6.

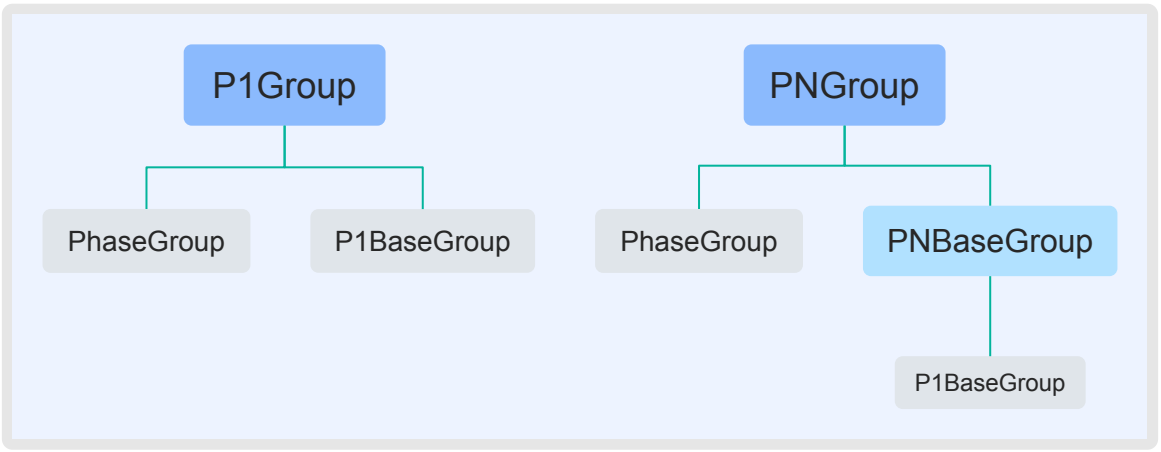


Figure 6: Hierachy of Pauli group formalism

In Figure 6, the structure `P1Group` formalises the single-qubit Pauli group $\mathcal{P}_1$ (See Section 2.3.1), and `PNGroup` formalises $\mathcal{P}_n$. They are composed by lower-level group structures `PhaseGroup`, `P1BaseGroup` and `PNBaseGroup`. In the following sections, we provide a bottom-up explanation of these structures.

4.3.3 $\mathcal{P}_1/Z_4$: Quotient Single-Qubit Pauli Group

The `P1BaseGroup` is actually the $\mathcal{P}_1$ with 4 possible phases ignored. In the language of group theory, it is usually referred to as the quotient group $\mathcal{P}_1/Z_4$. More generally, we can define the quotient group $\mathcal{P}_n/Z_4$:

Definition 4.2. The quotient group $\mathcal{P}_n/Z_4$ is the the n-qubit Pauli group $\mathcal{P}_n$ but with the global phase ignored.

The module `P1BaseGroup` describes the formalism of $\mathcal{P}_1/Z_4$ based on the inductive type `PauliBase` (See Section 4.3.1). The multiplication on the `PauliBase` is defined according to Table 1:

```
Definition mul_p1b(a b: PauliBase): PauliBase :=
  match a, b with
  | X, X ⇒ I | X, Y ⇒ Z | X, Z ⇒ Y
  | Y, X ⇒ Z | Y, Y ⇒ I | Y, Z ⇒ X
  | Z, X ⇒ Y | Z, Y ⇒ X | Z, Z ⇒ I
  | I, p ⇒ p | p, I ⇒ p
end.
```

The name `mul_p1b` stands for multiplication over base elements of single-qubit Pauli groups. The term "base elements" simply means assuming all phase factors to +1.

It is straightforward to prove that the multiplication `PauliBase` forms a finite group on the multiplication `mul_p1b`: The associativity can be proved by case analysis; The identity element is `I`; The

inverse of any `PauliBase` is simply itself. We select some key definitions and lemmas, and full proof is in the Coq repository as PauliGroup.v

```
Definition id_p1 := I.
Definition inv_p1: PauliBase → PauliBase := fun p ⇒ p.

Lemma mul_p1b_assoc   : associative mul_p1b.
Lemma mul_p1b_id      : left_id id_p1 mul_p1b.
Lemma mul_p1b_left_inv: left_inverse id_p1 inv_p1 mul_p1b.

HB.instance Definition P1BaseGroup :=
  isMulGroup.Build PauliBase mul_p1b_assoc mul_p1b_id mul_p1b_left_inv.
```

We first define the identity element `I` and the inverse operation `inv_p1`. Then we prove three lemmas: the associativity of multiplication, the element `I` serves as the left-identity element, and the `inv_p1` is a left-inverse function for `mul_p1b`. Finally, we use these three lemmas to construct the group by the command `HB.Instance Defintion` and constructor `isMulGroup.Build`.

We briefly explain the language of MathComp. The `HB.Instance` command is for constructing algebraic structures. It provides various additional benefits compared to Coq's original `Definition` command. The most important one is that it employs a language feature of Coq named *canonical structures* to automatically infer group-related operations and lemmas. For example, after constructing the group, this proposition

```
Goal ∀ (p1 p2: PauliBase), p1 * p2 = mul_p1b p1 p2.
Proof. reflexivity. Qed.
```

The tactic `reflexivity` applies to a goal with form `t=u` and checks that `t` and `u` are convertible[4]. The fact that the tactic `reflexivity` solves the goal suggests that `mul_p1b` and `*` are recognised as the same function after type inference. This is only possible because the command `HB.Instance` registered `mul_p1b` as a canonical multiplication operation and allows operation `*` to act as `mul_p1b`.

Similarly, MathComp registers "`x^-1`" as the canonical operation for group inverse on `x`, and "`1`" as the canonical identity element of a group. When there is no ambiguity, we will use MathComp's canonical definition in the following section.

Another advantage of using MathComp is its extensive library of proven lemmas for algebraic structures. To show this, notice this small example:

```
Variable (a b c d: PauliBase).

Example gsimpl_exp:
  a * b * c * d = a * ((b * c * 1) * d * a * a^-1).
Proof. gsimpl. Qed.
```

The goal is solved by a single tactic `gsimpl`, which is provided by MathComp as an automation tool to simplify expressions of multiplication on groups.

4.3.4 Z_4: Global Phases as a Group

We have briefly mentioned that phase scalers also form a group (see Section 2.3.1). Now we discuss its implementation. Like `PauliBase`, instead of directly using complex numbers, we also define phases are inductive types:

```
Inductive phase : Type :=
| One : phase   (*  1 *) | NOne : phase  (* -1 *)
| Img : phase   (*  i *) | NImg : phase. (* -i *)
```

Similar to `P1BaseGroup`, we then define the multiplication function

```
mul_phase: phase → phase → phase
```

And then we instantiate phases as a group `PhaseGroup` using `HB.instance` with the necessary lemmas.

On a mathematical perspective, this group is equivalent to the cyclic group Z_4. To see this, observe that

$$1 = (-1) \cdot (-1) = i^2 \cdot i^2 \tag{100}$$

which shows that i generates a cyclic group of order 4 under multiplication.

But the reason why we chose to formalise phases as a group is not purely for mathematical rigorous. Instead, it is primarily based on a proof engineering perspective: the phase group can be reused in both complete Pauli groups $\mathcal{P}_1$ and $\mathcal{P}_n$ (see Figure 6). And this strategy indeed benefits us. We will discuss the benefits later when discussing the formalism of $\mathcal{P}_1$ and $\mathcal{P}_n$.

4.3.5 $\mathcal{P}_1$: Single-Qubit Pauli Group

The group `P1Group` implements the single-qubit Pauli group $\mathcal{P}_1$. By its definition, $\mathcal{P}_1$ is the Cartesian product of phases and Pauli matrices. Therefore, we define the group structure using the product type `prod`[32].

```
Definition PauliElem1 := prod phase PauliBase.
```

Now we can express an element of $\mathcal{P}_1$ with a global phase. For example $-X$ is written as:

```
Notation "-X" := (NOne, X).
```

As `phase` and `PauliBase` are both groups, we can easily formalise $\mathcal{P}_1$. First, we need to encode the product phase when two Pauli matrices are multiplied (See Table 3):

[32]In the library `Coq.Init.Datatypes`, the constructor `prod A B` comes with a notation `A * B`. We will not use this notation because it causes ambiguity with multiplication over group elements.

```
Definition rel_phase(a b: PauliBase): phase :=
  match a, b with
  | I, _ ⇒ One | _, I ⇒ One
  | X, X ⇒ One | Y, Y ⇒ One | Z, Z ⇒ One
  | X, Y ⇒ Img | Z, X ⇒ Img | Y, Z ⇒ Img
  | Z, Y ⇒ NImg| Y, X ⇒ NImg| X, Z ⇒ NImg
  end.
```

The function name `rel_phase` represents the *relative phase* of Pauli matrices. A relative phase of two Pauli matrices A and B is the phase product of $A \times B$. Using this, we can define the multiplication on $\mathcal{P}_1$:

```
Definition mul_p1 (a b: PauliElem1): PauliElem1 :=
  match a, b with
  | (sa, pa), (sb, pb) ⇒
    (rel_phase pa pb * sa * sb, pa * pb)
  end.
```

This definition follows the principles of matrix multiplication. To see this, check this example,

$$(-X)(iY) = -1 \cdot i \cdot (XY) = -1 \cdot i \cdot (iZ) = Z \tag{101}$$

As Equation 101, three phases are multiplied together when computing $A \circ B$: two global phases of A and B, and the relative phase of AB.

Since the single-qubit Pauli group $\mathcal{P}_1/Z_4$ and the phase group Z_4 have already been formalised, it is straightforward to define the inverse function `inv_p1g`:

```
Definition inv_p1 (a: PauliElem1): PauliElem1 :=
  match a with
  | (s, p) ⇒ (s^-1, p^-1)
  end.
```

This definition exploits the canonical structure of MathComp. An element `(P: PauliElem1)` is constructed by a phase `s` and a PauliBase `p`. Therefore, the inverse of `PauliElem1` can be defined as the inverse of both components.

Similarly, the identity element is the combination of the identity elements of both `PhaseGroup` and `P1BaseGroup`.

```
Definition id_p1: PauliElem1 := (1, 1).
```

Note that the two `1`s in `id_p1` refer to different identity elements, which are inferred by the definition of `PauliElem1`.

Finally, we provide sufficient lemmas as obligations to construct `PauliElem1` with the multiplication `mul_p1` as a group. As such, `PauliElem1` is the implementation of $\mathcal{P}_1$.

4.3.6 $\mathcal{P}_n/Z_4$: Quotient N-Qubit Pauli Group

Now we consider extending the single-qubit Pauli group to the general n-qubit. Any element of the n-qubit Pauli group is composed of a global phase and a *Pauli string* (see Section 2.3.1) of length n. As usual, we first consider the elements with $\{+1\}$ phase. We selected the data type `tuple` from MathComp to construct the Pauli string :

```
Definition PauliString n := {tuple n of PauliBase}.
```

The data type `{ tuple n of T }` is a list of T with length `n`. The type of `tuple` puts a restriction on the length of a list, implying that it employs the mechanism of *dependent type*. This allows us to use a natural number `n` to control the desired length of the Pauli string.

Using this definition of `PauliString`, we can formalise the $\mathcal{P}_n/Z_4$ group (See Definition 4.2). Here we define the multiplication on it:

```
Definition mul_pnb {n: N} (a b: PauliString n): PauliString n :=
  map (fun x ⇒ (x.1 * x.2)) (zip a b).
```

This function `mul_pnb` indicates that the multiplication of two Pauli strings is defined as multiplication on each component. We require that the two parameters `a` and `b` are both of length `n` using dependent typing. The functions `map` and `zip` are defined on `list`. We can use it on `PauliString` because `tuple` is implemented as a subtype of `list`.

Although we define Pauli strings as a finite structure, we do not prove properties on Pauli strings by checking every position. Instead, we primarily use the induction rule to build proofs. Induction is a powerful tool in Coq to allow reasoning about recursive structures. Recall that proving a property $P(n)$ holds for all natural number n, we proceed with induction:

- If $n = 0$, we show that $P(0)$
- If $n = n' + 1$ for some n', we assume $P(n')$ and prove $P(n)$.

If both cases are provable, we say $P(n)$ holds for any $n \in \mathbb{N}$ by the principle of induction. The formal proof can be found in the source file PauliGroup.v.

4.3.7 $\mathcal{P}_n$: N-Qubit Pauli Group

The final step is to formalise the n-qubit Pauli group $\mathcal{P}_n$ using the existing materials. We use the term *Pauli Element* to refer to a set of elements in $\mathcal{P}_n$, with this formal definition.

```
Definition PauliElement (n: N) := prod phase (PauliString n).
```

This definition captures the definition of $\mathcal{P}_n$ (Definition 2.25) that any element of $\mathcal{P}_n$ can be seen as a global phase with a Pauli string. We follow Coq's notation convention to write $P.1$ as the global phase of P, and $P.2$ as the Pauli String of P.

Let's consider the multiplication on $\mathcal{P}_n$. To give the intuition, please consider this pen-and-paper example of multiplication:

$$
\begin{aligned}
iXX \cdot -ZY &= (i \cdot -1)(X \cdot Z) \otimes (X \cdot Y) \\
&= (i \cdot -1)(-iY) \otimes (iZ) \\
&= (i \cdot -1 \cdot (-i \cdot i))(YZ) \\
&= -iYZ
\end{aligned} \tag{102}
$$

Observe Equation 102, when multiplying two Pauli elements A B, it is required to compute:

1. The product phase, i.e. the relative phase of A and B.
2. The product Pauli string.

And the product is the combination of both the product phase and string. Following this analysis, let `rel_phase_n` be a process that computes the relative phase of two Pauli Elements, We define the multiplication function `mul_pn` as:

```
Definition mul_pn {n: ℕ} (a b: PauliElement n): PauliElement n :=
  ( rel_phase_n a b, a.2 * b.2 ).
```

In this definition, computing the product string is straightforward by reusing the multiplication defined on group $\mathcal{P}_n/Z_4$.

Now the problem reduces to computing the relative phase of two Pauli Elements A and B. The relative phase of A and B can derived from:

1. $A.1$, the global phase of A.
2. $B.1$, the global phase of B.
3. The relative phase between Pauli String $A.2$ and $B.2$.

On the third line of Equation 102, the phase scalars are composed of the global phase of two Pauli Elements (i and -1), and the relative phase of the Pauli strings ($-i \cdot i$). And notice that the relative phase of two Pauli strings is produced by *folding* the relative phases of each single Pauli matrix. Following this rationale, we give this definition:

```
Definition fold_rel_phase {n: ℕ} (a b: PauliString n): phase :=
  foldl mul_phase One [rel_phase p.1 p.2  | p ← zip a b].

Definition rel_phase_n {n: ℕ} (a b: PauliElement n): phase :=
  match a, b with
  | (sa, pa), (sb, pb) ⇒ (fold_rel_phase pa pb * sa * sb)
  end.
```

The function `fold_rel_phase` performs the phase-folding by scanning the two input Pauli Strings[33]. And the function `rel_phase_n` computes the relative phase of Pauli Elements of size n.

Like how we define $\mathcal{P}_1$, we reuse the existing group structures to define the identity elements and inverse function on $\mathcal{P}_n$:

[33]Notation `[E(x) | x ← t]` represents `map E t`. This notation is formally defined in MathComp.

```
Definition inv_png {n}(a: PauliElement n): PauliElement n :=
  match a with
  | (s, p) ⇒ (s^-1, p^-1)
  end.

Definition id_png (n:N): PauliElement n := (1, 1).
```

Finally, we defined and proved sufficient obligations to show that `PauliElement` forms a group. In conclusion, we formalise $\mathcal{P}_n$ as `PauliElement` with a multiplication function `mul_pn` defined over it.

4.3.8 Verifying Pauli Groups

We have formalised Pauli groups and verified that they indeed form finite groups. However, this work is based on inductively defined data types instead of directly on matrices (see Section 4.3.1). Therefore, there might exist the chance that data types like `PauliElement` and the multiplication over them are not consistent with Hilbert space. For example, one could simply make a pattern-matching mistake in `rel_phase` (Section 4.3.5).

To rule out this possibility, we define the correctness of the Pauli group as follows

Definition 4.3. We say the Pauli group $\mathcal{P}_n$ is *correct* if

$$\forall A \in \mathcal{P}_n, \forall B \in \mathcal{P}_n, [\![A]\!] \times [\![B]\!] = A \cdot B \tag{103}$$

Where $\cdot$ is the group multiplication, $[\![x]\!]$ interprets inductive data types x as a matrix in Hilbert space, and $\times$ is the matrix multiplication.

We formalised this correctness property and proved it is true at the end of PauliGroup.v.

```
Theorem int_pn_Mmult n:
  ∀ (x y: PauliElement n), int_pn x × int_pn y = int_pn (x * y).
```

In which the function `int_pn` is the interpretation of `PauliElement n`. The notation `×` stands for matrix multiplication, and `*` stands for group multiplication.

Theorem `int_pn_Mmult` (and Definition 4.3) essentially verifies the correctness of group multiplication is consistent with interpretation on Hilbert space. In addition, it actually formalises Theorem 2.28, which has the functionality of reducing matrix multiplication into group multiplication.

4.4 Formalism of Group Actions

This section discusses the formalism of Pauli group actions in the Coq development Action.v. Recall that an n-qubit Pauli Element can be interpreted as a 2^n square matrix, and thus it can apply to an n-qubit state (Section 2.3.1). We adopted the formalism of *group action* to describe this intuition.

Definition 4.4. A group G is said to act on a set X where there is a map $\phi : G \to X \to X$ which satisfies:

1. (Identity Action) $\forall x, \phi(e, x) = x$ where e is the identity element of G.
2. (Compatibility) $\forall g, \forall h, \forall x, \phi(g, \phi(h, x)) = \phi(g \cdot h, x)$

remark. In Definition 4.4, the second obligation is the generalisation of Theorem 2.28, in which a group action ϕ is reduced to a group multiplication $\cdot$.

The group action provides a fundamental description of how groups operate on sets in a way that respects the group operation. In addition to mathematical rigour, this abstraction can be used to support compositional reasoning and automation in proof engineering.

At the outset, we tried to adopt the group action formalism from MathComp (Library mathcomp.fingroup.action). However, we found it is impractical because

- MathComp require that the set X on which actions are applied is finite. However, this is not true if X refers to the quantum states, as any unit vector in Hilbert space can be seen as a quantum state.
- QuantumLib requires the assumptions that input quantum states are well-formed (See Section 3.1.2). Hence, the definition of group actions needs to be modified slightly to incorporate this design. In particular, we need to add the assumption that all $x \in X$ are well-formed.

Hence, we have modified MathComp's implementation of group actions to our needs. We call our definition of group action *quantum group actions*, which abstracts the effects of a group structure applied to quantum states.

4.4.1 Quantum Group Actions

Based on the definition of group action (Definition 4.4), we firstly define the action function:

```
Definition ActionType (aT: finGroupType) := Vector (2^dim) → aT → Vector (2^dim).
```

Then, we formalize the obligations in the Definition 4.4, binding it into a structure `Action`.

```
(* identity action *)
Definition act_id (to: ActionType) := ∀ (x: Vector (2^dim)),
  WF_Matrix x → to x 1 = x.

(* compatibility *)
Definition act_comp to x :=
  ∀ (a b: aT), to x (b * a) = to (to x a) b.

Definition is_action to :=
  act_id to ∧ ∀ x, act_comp to x.

Record action := Action {
  act :> ActionType;
  _ : is_action act
}.
```

The structure Action is constructed by an action map `act` and a proof that `act` satisfies two properties: the `act_id` ensures the identity action, and the `act_comp` ensures the compatibility. Note that we assume well-formedness (`WF_Matrix`) in this definition. This assumption comes from the requirement of QuantumLib (see Section 3.1.2).

4.4.2 Actions of Pauli Groups

In the definition of `ActionType`, we avoid assuming the actual instance of groups. Instead, we use a canonical type `finGroupType` as a placeholder. We now define a function `applyP` as applying an n-qubit Pauli element to an n-qubit state:

```
Definition applyP : Vector (2^n) -> PauliTuple n -> Vector (2^n) :=
  fun psi op => (int_pn op) × psi.
```

The map `applyP` is built based on the interpretation function `int_pn` and the matrix multiplication `×`. The map `applyP` is equivalent to:

$$\forall P \in \mathcal{P}_n, \forall |\psi\rangle, \phi(P, |\psi\rangle) := P|\psi\rangle \tag{104}$$

With the position of argument modified. We then show `applyP` is a group action:

```
Lemma applyP_is_action: is_action applyP.

Canonical act_n := (Action applyP applyP_is_action).
```

Remember Definition 4.4, a group action needs to satisfy two obligations:

- The identity action property comes from the fact that the identity element of $\mathcal{P}_n$ is I;
- The compatibility can be proven using the theorem `int_pn_Mmult` in Section 4.3.8.

The full proof can be found in the Action.v in the Coq development. Finally, we construct `applyP` as a canonical structure of Action.

To improve readability, we will use the notation `Apply ... on ...` for the function `applyP` in the following sections. For example, consider these two equivalent lemmas:

```
Lemma applyP_mscale { n: N }:
   ∀ (operator: PauliElement n) (psi: Vector (2^n)) (a: C),
     Apply operator on (a .* psi) = a .* Apply operator on psi.

Lemma applyP_mscale' { n: N }:
  ∀ (operator: PauliElement n) (st: Vector (2^n)) (a: C),
    applyP (a .* st) operator = a.* (applyP st operator) .
```

The lemma `applyP_mscale` indicates that applying a Pauli operator commutes with scalar multiplication.

4.4.3 Pauli Operator

A *Pauli operator* strictly refers to a Hermitian element of the Pauli group (Definition 2.29), that is, a Pauli element with global phase $\{\pm 1\}$. They can represent both unitary evolution and quantum observables. Therefore, we are particularly interested in them when discussing group actions.

From a proof engineering perspective, we would like most operators to be with $+1$ phases. Because they can be represented using a data type PauliString (Section 4.3.6) and hence allow us to avoid discussing global phases. Excluding operators with -1 phase does not impair the expressiveness of Pauli operators because:

- Suppose the operator represents a unitary evolution. The global phase brought by the operator is not measurable by Corollary 2.14.
- Suppose the operator represents an observable. Since any Pauli operator only has possible measurement results $\{\pm 1\}$ (Corollary 2.34), adding a -1 global phase merely flips the measurement result. i.e. Let P be a Pauli operator and $|\psi\rangle$ be a state. If measuring P on $|\psi\rangle$ yields $+1$, then measuring $-P$ on $|\psi\rangle$ yields -1, and vice versa. A sign flip does not affect the information we can get from a measurement.

With this observation, we can formalise Pauli operators in our Coq development.

```
Notation PauliOperator := PauliString.

Definition PauliOpToElem {n: ℕ} (x : PauliOperator n) : PauliElement n := (One,x).
Coercion   PauliOpToElem : PauliOperator >-> PauliElement.
```

The PauliOperator is merely a notation that refers to PauliString (See Section 4.3.6). In addition, we enabled a type coercion to PauliOperator as a PauliElement with a $\{+1\}$ phase. This coercion is enabled when the function requires a PauliElement, such as applyP (Section 4.4.2).

4.5 Operations on Pauli Groups

This section presents several operations defined on Pauli groups. We omit the function implementations and instead provide their type signatures along with correctness properties that specify their denotational semantics. Readers interested in the full definitions can refer to the Operations.v file in the Coq repository.

Concatenation The Lemma 2.33 involves an operation that concatenates two Pauli operators. We formalise this operation as a function `concate_pn`:

```
Definition concate_pn {n m}:
  PauliElement n -> PauliElement m -> PauliElement (n + m).
```

This definition employs dependent type to encode the relation between Pauli Element lengths: It combines two Pauli elements – one acting on n qubits and the other on m qubits – into a single Pauli element that acts on n + m qubits.

We verify `concate_pn` by showing it is equivalent to the tensor product:

```
Theorem compose_pstring_correct:
  ∀ {n m: ℕ}  (ps1: PauliElement n) (ps2: PauliElement m),
  int_pn (concate_pn ps1 ps2) = int_pn ps1 ⊗ int_pn ps2.
```

Negation Theorem 2.32 involves an operation that negates the global phase of a Pauli operator. We define this operation as `negate_Pn`, and later verify its correctness.

```
Definition negate_Pn {n} : PauliElement n -> PauliElement n.

Theorem negate_phase_Pn_correct n:
  ∀ (a: PauliElement n),  int_pn (negate_Pn a) = -1 .* int_pn a.
```

The correctness theorem states that negating the phase has the effect that scaling the matrix representation by -1, which is the mathematical meaning of negation.

4.6 Discussion and Summary

The Role of Ssreflect One aspect that has not been mentioned is the involvement of the ssreflect proof language. The ssreflect language is an extension to the tactic language that comes by default in Coq. The name ssreflect stands for *small-scale reflection*. Reflection is a technique in formal proofs that changes the form of a goal to a computable equivalence when possible. By doing so, we can achieve a higher-level of automation. For example, while the general equivalence between numerical data types are undecidable, the equivalence between natural numbers, as a special case, is decidable. Hence, ssreflect features reasoning by reflection in a flexible way (therefore the name *small-scale*).

MathComp is built with ssreflect and employs reflection-based reasoning extensively. To use the lemmas and automation from MathComp, we also adopted ssreflect language for building proofs. This makes proofs in the Coq Repository look slightly different from the ones written using only the tactic language. Listing 7 shows an example that compares two styles.

```
Lemma pn_idP {n: N}: id_pn n.+1 = [tuple of I :: (id_pn n)].
```

```
Proof.
  unfold id_pn.
  unfold id_p1b.
  apply eq_from_tnth.
  intros i.
  repeat rewrite (tnth_nth I).
  simpl.
  reflexivity.
Qed.
```

```
Proof.
  rewrite /id_pn /id_p1b.
  apply: eq_from_tnth ⇒ i.
  by rewrite !(tnth_nth I).
Qed.
```

Listing 7: Proof script of a same lemma `pn_idP`. ON the left is the proof writing in tactic language, and on the right is the equivalent proof in ssreflect language. The lemma is from PauliGroup.v in the Coq repository.

Observe that the ssreflect proof is significantly shorter. One reason is its compact syntax. Another reason is that lemma `tnth_nth` turns the goal to a decidable form, and thus can be solved by a single `by` command.

A common misconception about ssreflect is that it introduces new axioms or alters the underlying logic of Coq. In fact, ssreflect is entirely conservative—it does not extend or modify the logic system. Instead, what it does is to introduce automation mechanics based on reflection. If readers find the ssreflect proofs in the Coq repository hard to read, we recommend Gonthier's introduction [18] and its manual[34] .

Summary This chapter discusses the formalism and verification of Pauli groups. The formalism and verification are both implemented in the interactive theorem prover Coq (Section 4.1). We took advantage of QuantumLib and MathComp of their certified mathematical objects (Section 4.2). The

[34]As for May 2025, one online manual of ssreflect is available at https://rocq-prover.org/doc/V8.20.1/refman/proof-engine/ssreflect-proof-language.html

formalism is developed in an incremental approach – from the single-qubit quotient group $\mathcal{P}_1/Z_4$ to the general n-qubit Pauli group $\mathcal{P}_n$ (Section 4.3). Based on the group structure, we further implement group actions (Section 4.4) and additional operations on Pauli elements (Section 4.5).

Recall the primary objective of this research is to investigate the question that, if we can use stabiliser formalism to achieve an efficient verification process of quantum error correction codes. The Pauli groups are the fundamental structure of the stabiliser formalism (Section 2.3.1). In the following chapter, we will discuss how to use Pauli groups to implement quantum observables and stabilisers.

Chapter 5

Formalism of Quantum Stabiliser Code

This chapter discusses the implementation of quantum observables and stabilisers. In quantum mechanics, an observable is a Hermitian operator (Definition 2.9). Hermitian operators have real eigenvalues, which correspond to the possible measurement values (Postulate 2.11). In quantum computing, measuring an observable is the only way to get the information from a quantum computing process. On the other hand, stabilisers are special types of observables that define subspaces of the Hilbert space (see Figure 5). In the stabiliser formalism, a stabiliser is typically a Pauli operator that leaves certain quantum states unchanged. Quantum observables and stabilisers play an important role in quantum error correction.

In the following sections, we will formalise observables and stabilisers in Coq, building on the Pauli group structures developed previously.

5.1 Observable and Projective Measurement

This section discusses the formalism of quantum observables and projective measurement. The full implementation is in Observable.v in the Coq repository.

In quantum computing, an observable is a Hermitian matrix (Definition 2.9). We use the Coq notation `M†` for the conjugate transpose of `M` and define the predicate `hermitian`:

```
Definition hermitian {n:N} (H: Square (2^n)): Prop := H† = H.
```

Although the Projective Measurement has the general form of the probabilistic distribution (see Postulate 2.11), we are only interested in special cases where the states are the eigenvectors of the observable. To illustrate, consider the single-qubit observable Z (which corresponds to measurement in the computational basis):

- Measuring Z on $\alpha|0\rangle + \beta|1\rangle$ yields $+1$ with probability α^2, and -1 with probability β^2; The state after measurement is $|0\rangle$ if observing $+1$, or $|1\rangle$ if observing -1.
- Measuring Z on $|0\rangle$ will deterministically result in $+1$, since $Z|0\rangle = |0\rangle$. The state is left unchanged after measurement.
- Measuring Z on $|1\rangle$ will deterministically result in -1, since $Z|0\rangle = -|0\rangle$. The state is left unchanged after measurement.

The first case is the general case of projective measurement, which results in a probabilistic distribution. In contrast, the latter two are special cases where the state is an eigenvector of the observable, resulting in deterministic outcomes. This observation is a direct result of Projective Measurement postulate and one proof is discussed in Corollary 2.13. Measurement on eigenvectors precisely aligns with the needs of stabilisers, as stabiliser measurements are required to preserve the quantum state.

Hence, we formalise this special case of measurements as `eigen_measure` in the Coq development:

```
Definition eigen_measure {n: N} (m: C) (M: Square (2^n)) (psi: Vector (2^n)) :=
  WF_Matrix M ∧ hermitian M ∧ M × psi = m .* psi.
```

The predicate `eigen_measure` captures the behaviour of projective measurement in the special case where the quantum state `psi` is an eigenvector of the observable `M`. While it is not a complete formalism of the Projective Measurement Postulate. This special case is commonly accepted as a known fact in the literature. For instance, the canonical textbook by Nielsen and Chuang [37] notes:

"(Let g be an observable and $|\psi\rangle$ a state). In this instance, $g|\psi\rangle = |\psi\rangle$, and thus a measurement of g yields $+1$ with probability one."[35]

The proposition `eigen_measure` does not assume any specific class of observables, such as the Pauli operators. To apply it to the Pauli operators introduced in the previous chapter, we must first establish that all Pauli operators are Hermitian.

```
Theorem pauli_hermitian {n: N} :
  ∀ (operator: PauliOperator n), hermitian (int_p operator).
```

Here the function `int_p {n}: PauliOperator n → Matrix (2^n)` is the interpretation function of Pauli operator, which has been verified previously (see Section 4.3.8). The proof is by induction on `n`. Readers can check the Coq repository for the detailed proof.

Finally, we can formalise the measurements using Pauli operators.

```
Definition eigen_measure_p {n:N} (m: C) (P: PauliOperator n) (psi: Vector (2^n)) :=
  (int_p P) × psi = m .* psi.

Theorem eigen_measure_p_correct {n}:
  ∀ (m:C) (P: PauliOperator n) (psi: Vector (2^n)),
    eigen_measure_p m P psi ↔ eigen_measure m (int_p P) psi.
```

The function `eigen_measure_p` describes the measurement using Pauli operators as observables. We prove its correctness by establishing the equivalence to `eigen_measure`.

We now use the more readable Coq notation "`Meas' P on psi ⟶ m`" to represent "`eigen_measure_p m P psi`". For example, we can verify the examples at the beginning of this section.

```
Goal 'Meas [Z] on | 0 ⟩ ⟶ 1.
Goal 'Meas [Z] on | 1 ⟩ ⟶ -1.
```

We use *Pauli observable* to refer to using Pauli operators as a quantum observable.

```
Notation PauliObservable := PauliString.
```

In addition to correctness, another important observation of Pauli observable is that their eigenvalues are either $+1$ or -1 (see Corollary 2.34). This means when using the Pauli observable to

[35] Quoted from Section 10.5.3 in [37]

measure a state, the only possible result is $+1$ or -1. We verified this fact as a Coq theorem `pauli_observable_measurement_results`.

```
Corollary pauli_observable_measurement_results {n}:
  ∀ (ob: PauliObservable n) (psi: Vector (2^n)) m,
  psi ≠ Zero → WF_Matrix psi →
  'Meas ob on psi ⟶ m →
  m = 1 ∨ m = -1.
```

Having the observable, now we can discuss quantum stabilisers.

5.2 Quantum Stabiliser

A quantum stabiliser S of a state $|\psi\rangle$ has the effect that leaves the state $|\psi\rangle$ unchanged. We formalise the notion of stabiliser in Stabiliser.v in the Coq repository. Following the definition, we define a stabilizer as a predicate `stabiliser_of`.

```
Definition stab {n:ℕ} (pstring: PauliElement n) (psi: Vector (2^n)):=
  act_n psi pstring = psi.
```

Where `act_n x g` is the canonical structure that represents applying a group action `g` to `x` (see Definition 4.4). The name `stab` is a replica of the mathematical notation of stabilisers:

$$\mathrm{Stab}_G(x) := \{g \in G \mid g(x) = x\} \tag{105}$$

Which states that the group action g has no effect on x, and this is exactly what Definition 2.31 means.

We prove this definition correct by Coq corollary `stabiliser_correct`:

```
Theorem stb_eigen_measure_1 {n}:
  ∀ (p: PauliElement n) (psi: Vector (2^n)),
  stab p psi ⟷ eigen_measure 1 (int_pnb p) psi.
```

It describes that if p of a Pauli observable is a stabiliser of state $|\psi\rangle$, then measuring p on $|\psi\rangle$ yields 1 with certainty. Therefore, proving this theorem verifies the correctness of the stabiliser.

One beneficial aspect of stabiliser is that it allows compositional reasoning using group properties. To illustrate, we select a few lemmas:

```
Lemma stb_compose:
  ∀ {n: ℕ} (pstr1 pstr2: PauliElement n) (ψ1 ψ2:  Vector (2^n)),
  let cpstring := concate_pn pstr1 pstr2 in
  stab pstr1 ψ1 → stab pstr2 ψ2 →
  stab cpstring (ψ1 ⊗ ψ2).

Lemma inv_stb:
  ∀ {n: ℕ} (pstr: PauliElement n) (ψ:  Vector (2^n)),
  WF_Matrix ψ → stab pstr ψ → stab (pstr^-1) ψ.

Lemma stb_closed:
  ∀ {n: ℕ} (pstr1 pstr2: PauliElement n) (ψ:  Vector (2^n)),
  stab pstr1 ψ →
  stab pstr2 ψ →
  stab (pstr1 * pstr2) ψ.
```

- The Coq lemma `stb_compose` verifies Lemma 2.33: Let $|\psi\rangle$ and $|\varphi\rangle$ be two states stabilised by A and B, then $|\psi\rangle \otimes |\psi\rangle$ is a stabiliser state of $A \otimes B$[36].
- The `inv_stb` states that the inverse of a stabiliser is also a stabiliser.
- The `stb_closed` states that the set of stabilizers for a quantum state is closed under multiplication.

Note that in these theories, we use `PauliElement` instead of `PauliObservable`. Recall that the difference between `PauliElement` and `PauliObservable` is that the former can have global phases other than the trivial $+1$.

5.3 Error Detecting Codes

We say a quantum code is a detecting code if the code can detect certain errors. A non-trivial quantum code needs to detect at least one error. Therefore, all quantum error correction codes are detecting codes.

Firstly, following the stabiliser formalism, we represent any error as a Pauli operator:

```
Notation ErrorOperator := PauliOperator.
```

As an aside, note that observables `PauliObservable` and error operators `ErrorOperator` are both represented by Pauli operators. This is intentional and replicates the pen-and-paper stabiliser formalism: since a Pauli operator is both Hermitian and unitary, it can act as both an observable and a unitary evolution.[37]

What should a quantum error detecting code contain? Recall the pen-and-paper reasoning in Section 2.3.3, which involves a code space, a set of syndrome measurements, and a set of detectable errors. For example, consider the 3-qubit bit-flip code (Section 2.2.2):

- The code space is $\mathcal{C} = \alpha|000\rangle + \beta|111\rangle$
- The set is syndrome measurements are $\{Z_1Z_2, Z_2Z_3\}$
- The set of detectable errors are $\{X_1, X_2, X_3\}$

[36]See Section 4.5 for the definition of function `concate_pn` that concatenates two Pauli elements

[37]An error is a unitary evolution, see Section 2.2.1

Capturing this observation, we introduce the following parameters in the formalism of the detecting code.

```
Variable (dim: ℕ).

Variable (SyndromeMeas: {set (PauliObservable dim)}).
Variable (DetectableErrors: {set (ErrorOperator dim)}).
Variable (psi: Vector (2^dim)).

Hypothesis Hnz: psi ≠ Zero.
Hypothesis Hwf: WF_Matrix psi.
```

- `dim` is the dimension of the code, which is implicit in the pen-and-paper formalism.
- `SyndromeMeas` is the set of syndrome measurements of the code, represented by Pauli observables.
- `DetectableErrors` is the set of detectable errors, represented by Pauli operators.
- `psi` is the code space, we write it as $|\psi\rangle$ in this section.
- In addition, we require that the code space is non-zero and well-formed.

In the stabiliser formalism, all the syndrome measurements should be stabilisers of the code space. The predicate `obs_be_stabiliser` is a translation of this requirement.

```
Definition obs_be_stabiliser :=
  ∀ (M: PauliObservable dim), M ∈ SyndromeMeas → stab M psi.
```

Recall that stabilisers are defined to be with $+1$ measurement.[38] We can distinguish error code spaces with -1 measurement result. This idea is formalised by the predicate `detectable E`.

```
Definition detectable (E: ErrorOperator dim) :=
  let psi' := Apply E on psi in
    ∃ M,  M ∈ SyndromeMeas ∧ 'Meas M on psi' ⟶ -1.
```

It says that an Error E is detectable if there exists a syndrome measurement M such that measuring M on the corrupted state $E|\psi\rangle$ yields -1.[39]

Before building the structure of the detecting code, let's verify that the predicate `detectable E` is correct. But what does it mean for an error to be *detectable*? Remember that in quantum computing, measurement is the only way to extract information from a quantum state. We have defined the code space as the subspace where all measuring stabilisers return $+1$. Hence, when the measurement yields a different result than $+1$, it indicates that some error has occurred. To formalise this intuition, we prove two correctness properties: `detectable_necessary` and `detectable_sufficient`. Together, they show that `detectable` is the necessary and sufficient condition for an error to be detectable by a code.

[38] See formalised theorem `stb_eigen_measure_1` in Section 5.2

[39] The notation `Apply E on psi` is defined in Section 4.4.2 as the applying group action E on state $|\psi\rangle$.

```
Theorem detectable_necessary :
  ∀ (E: ErrorOperator dim),
  (let psi' := Apply E on psi in
    ∃ (M: PauliObservable) (m: C),
      M ∈ SyndromeMeas ∧ 'Meas M on psi' ⟶ m ∧ m ≠ 1)
    → detectable E.

Theorem detectable_sufficient :
  ∀ E, detectable E →
  let psi' := Apply E on psi in
    ∃ M m,  M ∈ SyndromeMeas ∧ 'Meas M on psi' ⟶ m ∧ m ≠ 1.
```

They are proved by the fact that any Pauli observable has the measurement result being either $+1$ or -1.[40].

We can then build the structure of a quantum error detecting code:

```
Record ErrorDetectingCode := BuildDetectingCode {
  dim: ℕ
(* Codespace *)
; code: Vector (2^dim)
(* Observables *)
; obs: {set PauliObservable dim}
(* Detectable Errors *)
; err: {set PauliOperator dim}
(* Obligation1: observables must be stabilisers of codespace *)
; ob1: obs_be_stabiliser obs code
(* Obligation2: all errors must be detectable by measurement *)
; ob2: errors_detectable obs err code
}.
```

The structure `ErrorDetectingCode` describes a valid quantum error detecting code:

- Field `code` is the code space.
- Field `obs` is the set of syndrome measurement observables.
- Field `err` is the set of detectable errors.

And two obligations are

1. Obligation `ob1` states that all observables must be stabilisers of codespace.
2. Obligation `ob2` states that all errors must be detectable by some stabilisers in `obs`.

5.4 Recovery and Correcting Code

A detecting code can distinguish the code space from the error space. But it does not need to be able to carry out a recovery when detecting an error. To achieve this, the code structure must be able to *locate* the error, i.e. knowing what error has happened and which qubits the error is applied to. The code that is able to locate errors is referred to as the *error correcting code*[41].

[40]See theory in Corollary 2.34 and formalised theorem `pauli_observable_measurement_results` in Section 5.1

[41]Please do not be confused by the terms *error correction code* and *error correcting code*. An error correction code is a general term for all coding schemes, including detecting codes (see Definition 2.16). The term originates from

5.4.1 Recovery from Error

Why does locating a unique error suffice for recovery? Let's consider what it means to recover from an error. Essentially, an operator R is said to recover an error E if

$$\forall |\psi\rangle, R(E|\psi\rangle) = |\psi\rangle \tag{106}$$

That is, applying the recovery R can recover the error state $E|\psi\rangle$.

A sufficient condition for Equation 106 is

$$P \cdot E = I \tag{107}$$

Since

$$\forall |\psi\rangle, R(E|\psi\rangle) = R \cdot E|\psi\rangle = I|\psi\rangle = |\psi\rangle \tag{108}$$

We formalise this reasoning and prove it's correct in Coq theorem `recover_by_correct`

```
Definition recover_by {n} (E: ErrorOperator n) (R: PauliOperator n) :=
  R * E = 1.

Theorem recover_by_correct {n} :
  ∀ (E: ErrorOperator n) (R: PauliOperator n) (phi: Vector (2^n)),
  WF_Matrix phi →
  recover_by E R →
  let phi' := Apply E on phi in
    (Apply R on phi') = phi.
```

The theorem `recover_by_correct` states that first applying the error on any state $|\varphi\rangle$, and then applying the recovery on $|\varphi\rangle$, the original state $|\varphi\rangle$ is restored. Therefore, the problem reduces to for any error operator `E`, finding a recover operator `R` that satisfies `recover_by E R`.

Recall Theorem 2.22 that for every Pauli operator P, $P \cdot P = I$. This implies that to for any error represented by a Pauli operator E, applying E again can recover the error. This renders recovery trivial: applying the error operator again can recover the error. For example, applying the bit-flip twice on the same qubit eventually brings the qubit to the original state:

$$\alpha|0\rangle + \beta|1\rangle \overset{X}{\to} \alpha|1\rangle + \beta|0\rangle \overset{X}{\to} \alpha|0\rangle + \beta|1\rangle \tag{109}$$

We verify this fact (applying an error operator again can recover the error) by Coq theorem `get_recover_correct`:

```
Definition get_recover {n:N} (E: ErrorOperator n): (PauliOperator n) := E.

Theorem get_recover_correct {n:N}:
  ∀ (E: ErrorOperator n),
    recover_by E (get_recover E).
```

historical usage. In contrast, we use *correcting code* to specifically refer to a quantum code that can recover from certain error states.

The function `get_recover` computes the recovery operator of input `E`. In fact, the implementation is simply returning `E`. Then, we prove that `E` can be recovered by `get_recover E`.

5.4.2 Error Correcting Code

Since the recovery from a located error is trivial, the key to building an error correcting code is to make it capable of locating errors. How do we locate an error? Again, measurements are the only way to get information from quantum computing processes (Section 2.1.2). Hence, we must require the measurements to have some special properties for error correcting codes.

Consider the minimal non-trivial case — when there are only two errors E_1 and E_2. The code needs to be able to distinguish E_1 and E_2 through a set of measurements. Take Figure 5 as an example, the error states $\alpha|001\rangle + \beta|110\rangle$ and $\alpha|010\rangle + \beta|101\rangle$ cam be differentiated by if the measurement result of Z_1Z_2 is $+1$ or -1.

We formalise this idea as a property `distinguishable_by` on an error detecting code `edc`:

```
Definition distinguishable_by
  (edc: ErrorDetectingCode) (E1 E2: PauliOperator) (M: PauliObservable) :=
∀ r q,
  let psi_e1 := Apply E1 on edc.(code) in
  let psi_e2 := Apply E2 on edc.(code) in
    M ∈ edc.(obs) →
    ('Meas M on psi_e1 ⟶ r) →
    ('Meas M on psi_e2 ⟶ q) →
    r ≠ q.
```

The property `distinguishable_by` states that measuring stabiliser M on states corrupted by two errors E_1 and E_2 yields different results. Therefore, measuring M can help us to locate the error.

Let's then extend this minimal two-error case to a general n-error case: for every pair of detectable errors $\langle E_1, E_2\rangle$ in a code C, E_1 and E_2 can be distinguished by a stabiliser M of C. This defines the property `error_identified_uniquely` on an error detecting code:

```
Definition error_identified_uniquely (edc: ErrorDetectingCode): Prop :=
  ∀ (E1 E2: PauliOperator (dim edc)),
    E1 ∈ edc.(err) → E2 ∈ edc.(err) →
    E1 ≠ E2 →
    ( ∃ M, distinguishable_by edc E1 E2 M ).
```

Equivalently, the property `error_identified_uniquely` states that every error in the detectable error set has a unique syndrome. This additional property allows us to formalise the error correcting code:

Definition 5.1. An error correcting code is a detecting code in which every error has a unique syndrome.

This definition is formalised as the structure `ErrorCorrectingCode`.

```
Record ErrorCorrectingCode := {
  edc :> ErrorDetectingCode;
  correction_obligation: error_identified_uniquely edc
}.
```

5.5 Undetectable and Indistinguishable Errors

How do we reason about the limitations of error detecting codes? Section 2.3.4 provide two definitions can we can use: undecidable and indistinguishable errors.

First, let's consider undetectable errors (Definition 2.37). An error E is undetectable by code $\mathcal{C}$ if the syndrome E is empty. In other words, measuring all stabilisers on the error state created by E yields $+1$. Therefore, we formalise the definition of undecidable errors as

```
Definition undetectable (edc: ErrorDetectingCode) E :=
  let psi' := Apply E on edc.code in
    ∀ M,  M ∈ edc.obs → 'Meas M on psi' ⟶ 1.
```

Then, we consider the indistinguishable errors (Definition 2.38). By definition, two errors E_1 and E_2 are indistinguishable if their syndromes are identical. If we unfold the definition of syndrome, it essentially says that the set of stabilisers that yield -1 when measuring error states created by E_1 and E_2. We formalise this idea as *indistinguishable errors*:

```
Definition indistinguishable (edc: ErrorDetectingCode) E1 E2 :=
  ∀ M, M ∈ edc.(obs) →
  let psi_e1 := Apply E1 on edc.(code) in
  let psi_e2 := Apply E2 on edc.(code) in
    ('Meas M on psi_e1 ⟶ -1) ⟷ ('Meas M on psi_e2 ⟶ -1)
```

For the 3-qubit bit-flip code, although it can detect bit-flip errors, it cannot distinguish between the error $X_1 \coloneqq XII$ and error $X_2X_3 \coloneqq IXX$. This indicates this code is limited to only tolerating a single bit-flip error.

The undecidable and indistinguishable errors as a means to show the limitation of a code. Together with detectable and distinguishable errors defined in the previous section, the tool is enough to formally reason a stabiliser code. We will show a few examples of reasoning about undetectable and indistinguishable errors in Section 6.1.4.

5.6 Error Detecting Condition

Recall the error detecting condition Theorem 2.32: A stabiliser code $\mathcal{C}$ can detect an error E if there exists a stabiliser S such that $SE = -ES$. Error detecting condition is a critical result in the stabiliser formalism. It shows that the stabilisers have enough information about the code space. From a proof-engineering perspective, this result enables a simplification of the reasoning — it is a sufficient condition to show E is correctable by $\mathcal{C}$. Let `-` notates the negation operation `negate_Pn` we defined in Section 4.5. The error detecting condition is formalised as

```
Corollary detectable_sufficient {n}:
  ∀ (S: PauliOperator n) (psi: Vector (2^n)) (Er: PauliOperator n) ,
  stab Ob psi -> S * Er = - (Er * S) ->
  let psi_E = (Apply Er on psi) in
    (Meas S on psi_E ⟶ -1).
```

The Coq corollary `detectable_sufficient` states that measuring the stabiliser `S` on error state `psi_E` yields the result `-1` if

1. `S` is a stabiliser of code space `psi`.
2. `S * Er` is the negation of `Er * S`.

Measuring -1 matches the definition of detectable errors (see Section 5.3). Therefore, this corollary can be applied when one needs to prove that an error is detectable. We will discuss an example of proving error detectability in Section 6.1.3.

On the other hand, we also prove the sufficient condition for an error not being detectable.

```
Corollary undetectable_sufficient
  (edc: ErrorDetectingCode) (Er: PauliOperator dim):
  (∀ (S: PauliObservable dim), S ∈ edc.obs -> S * Er = Er * S)
  -> undetectable edc Er.
```

The Coq corollary `undetectable_sufficient` states that an error is undetectable if an error `Er` commutes with all stabiliser of a quantum code `edc`.

5.7 Conclusion

This chapter discusses the formalism of stabilisers and quantum codes. Firstly, we formalise quantum observables and projective measurement postulate in Section 5.1. Based on the formalism of quantum observables, we formalise quantum stabiliser Section 5.2, a special observable that maintains the information of a quantum code. Then, we build the error detecting code in Section 5.3 and the error correcting code Section 5.4. Then, we formalise the undetectable and indistinguishable errors as a means to reason about the limitation of a quantum code Section 5.5. Finally, the error detecting condition is verified as a critical result of the stabiliser formalism.

These theories replicate the pen-and-paper stabilizer formalism commonly used in quantum error correction codes. Just as the traditional approach allows abstract and efficient reasoning about QECCs, our formalised theories provide a toolbox for formally verifying properties of QECC programs.

The following chapter discusses examples that use this formalism to verify some quantum codes.

Chapter 6

Evaluating Coq-QECC through Case Studies

To assess the effectiveness of Coq-QECC, we demonstrate its application in verifying several representative quantum codes, including the 3-qubit bit-flip code and the 9-qubit Shor's code. Both codes have been discussed in Section 2.2.

As mentioned in Section 3.3.1, the SQIR repository contains a few certified quantum error correction codes as examples. We refer to these examples as *SQIR-QECC examples*. SQIR-QECC examples contain the 3-qubit bit-flip and the 9-qubit Shor's code. This allows us to compare our certified code examples (referred to as *Coq-QECC examples*) with the prior SQIR-QECC examples, to evaluate how much benefit is gained by introducing the stabiliser formalism.

6.1 Fully Certified Three-Qubit Bit-flip Code

Recall Section 2.2.2, the three-qubit bit-flip code can detect and correct a single-qubit bit-flip error. i.e. an X error. This section discusses this example to evaluate Coq-QECC. We start by introducing the baseline — the SQIR-QECC example of certified bit-flip code (Section 6.1.1). Then, we focus on discussing our formalism based on Coq-QECC from Section 6.1.2 to Section 6.1.4. Finally, we evaluate Coq-QECC by presenting a comparison between these two versions (Section 6.1.5).

6.1.1 The SQIR-QECC Example of Three-Qubit Bit-Flip Code

The SQIR-QECC examples are not part of our contribution. Before starting the discussion of the SQIR-only verification of the three-qubit bit-flip code, we want to highlight that it was originally developed and formalised by external researchers. This section is included solely to provide background for subsequent discussion and comparison.

In the SQIR-QECC examples, the verification of the encoding program is basically the same as ours, but with a dimension of 5-qubits.

```
Theorem encode_correct : ∀ (α β : C),
  (uc_eval encode) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ |0,0,0,0⟩ )
  = α .*|0,0,0,0,0⟩ .+ β .* |1,1,1,0,0⟩.
```

Later, we will explain why this formalism employs 5 qubits, even though the code itself is only 3-dimensional. Now let's continue discussing the formalism.

The single-qubit bit-flip errors are defined as an inductive data type:

```
Inductive error : Set :=
  | NoError
  | BitFlip0
  | BitFlip1
  | BitFlip2.
```

The `error` type encodes no error, or a bit-flip on the first, second, or third qubit. The correctness of error recovery is `error_recover_correct`.

```
Theorem error_recover_correct (e : error) : ∀ (α β : C),
  (uc_eval (apply_error e; recover)) × (α .* |0,0,0,0,0⟩ .+ β .* |1,1,1,0,0⟩)
  = (α .* |0,0,0⟩ .+ β .* |1,1,1⟩) ⊗ (error_syndrome e).
```

In `error_recover_correct`, there are some helper functions:

- Function `apply_error: error → base_ucom dim` constructs a SQIR program that implements the effect of the error.

```
Definition apply_error (e : error) : base_ucom dim :=
  match e with
  | NoError ⇒ SKIP
  | BitFlip0 ⇒ X 0
  | BitFlip1 ⇒ X 1
  | BitFlip2 ⇒ X 2
  end.
```

- The SQIR program `recover` is a recovery procedure.
- The `error_syndrome: error → Vector` takes the error and returns the state of ancillary qubits.

```
Definition error_syndrome (e : error) : Vector 4 :=
  match e with
  | NoError  ⇒ |0,0⟩
  | BitFlip0 ⇒ |0,1⟩
  | BitFlip1 ⇒ |1,0⟩
  | BitFlip2 ⇒ |1,1⟩
  end.
```

And as shown in the listing, this implementation does not involve measurement. Instead, the two ancillary qubits are used to store the error syndrome. This follows measurement-free quantum error correction (MF QEC) implementation [24] (see Section 2.3.6), which we have discussed in Section 3.3.1.

6.1.2 Correctness of Encoding

Let the state before encoding be $|\psi\rangle_b = \alpha|0\rangle + \beta|1\rangle$. We say the program f_E is a correct encoding if

$$f_E(|\psi\rangle_b) = \alpha|000\rangle + \beta|111\rangle = |\psi\rangle \tag{110}$$

Now let's formalise it and verify its correctness. The encoding program shown in Figure 3 can be written as an SQIR program `encode`, which is a 3-qubit dimensional program.

```
Definition dim:N := 3.
Definition encode : base_ucom dim :=
  CNOT 0 1; CNOT 0 2.
```

Then, we define its code space $\mathcal{C}_{3b}$ that contains all possible $|\psi\rangle = \alpha|000\rangle + \beta|111\rangle$.

```
Variable (α β : C).
Definition psi: Vector (2^dim) := (α .* |0,0,0⟩.+ β .* |1,1,1⟩).
Hypothesis norm_obligation: α^* * α + β^* * β = 1.
```

Here, the hypothesis `norm_obligation` requires the code words to satisfy the normalisation condition (See Definition 2.2):

$$\begin{aligned} & \|\alpha|000\rangle + \beta|111\rangle\| = 1 \\ \vdash \quad & \sqrt{|\alpha|^2 + |\beta|^2} = 1 \\ \vdash \quad & |\alpha|^2 + |\beta|^2 = 1 \\ \vdash \quad & \alpha^* \cdot \alpha + \beta^* \cdot \beta = 1 \end{aligned} \tag{111}$$

Where c^* for $c \in \mathbb{C}$ represents complex conjugate of c.

After defining the encoding program and the code space, we can verify the correctness. The correctness of program `encode` is defined as a correct encoding to the code space.

```
Theorem encode_correct :
  (uc_eval encode) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ |0,0⟩ )
  = α .* |0,0,0⟩ .+ β .* |1,1,1⟩.
```

In which the `uc_eval` function takes a SQIR program and returns its denotational semantics as a unitary matrix. This theorem says the program `encode` is a unitary evolution that encodes the input state $|\psi\rangle_b = \alpha|0\rangle + \beta|1\rangle$ to $|\psi\rangle = \alpha|000\rangle + \beta|111\rangle$ in the code space $\mathcal{C}$.

6.1.3 Verification of Detectability and Correctability

We now turn to verify the construction of the bit-flip errors. In particular, we are going to verify:

1. The observables $\mathcal{S} = \{Z_1Z_2, Z_2Z_3\}$ are the stabiliser to the code $\mathcal{C}_{3b}$.

$$\forall|\psi\rangle \in \mathcal{C}_{3b}, \forall S \in \mathcal{S}, \text{Meas}(S, |\psi\rangle) = +1 \tag{112}$$

2. All single-qubit bit-flip errors $\mathcal{E} = \{X_1, X_2, X_3\}$ are detectable and correctable by this code.
   - **Detectability** means any error E can be detected by some stabiliser S through measurement. More formally,

$$\forall E \in \mathcal{E}, \forall|\psi\rangle \in \mathcal{C}_{3b}, \exists S \in \mathcal{S}, \text{Meas}(S, E|\psi\rangle) = -1 \tag{113}$$

   - **Correctability** means any error E has a unique error syndrome, that is, be located precisely.[42]

$$\forall(E_1, E_2) \in \mathcal{E} \times \mathcal{E}, E_1 \neq E_2 \rightarrow \text{Syndrome}(E_1, \mathcal{C}_{3b}) \neq \text{Syndrome}(E_2, \mathcal{C}_{3b}) \tag{114}$$

We first verify that the set of observables $\mathcal{S} := \{Z_1Z_2, Z_2Z_3\}$ are the stabilisers of the code space.

[42]Located error can be recovered by applying the error operator again, see Section 5.4.2.

```
(* Syndrome measurement *)
Definition Z12 := [Z, Z, I].
Definition Z23 := [I, Z, Z].
Definition SyndromeMeas: {set PauliObservable 3} :=
  [set Z12, Z23].

Theorem stabiliser_set_correct :
  ∀ (M: PauliObservable 3),
    M ∈ SyndromeMeas → Stab M psi.
```

Then we verify the detectability: for each single bit-flip error E in $\mathcal{E} := \{E_1, E_2, E_3\}$, there exists a stabiliser $S \in \mathcal{S}$ that can detect it. i.e. producing -1 in measurement.[43]

```
(* Set of single-qubit bit-flip error *)
Definition X1 := [X, I, I].
Definition X2 := [I, X, I].
Definition X3 := [I, I, X].
Definition BitFlipError: { set ErrorOperator 3 } :=
  [set X1, X2, X3].

Theorem errors_detectable_correct :
  ∀ E ∈ BitFlipError →
    ∃ M ∈ SyndromeMeas, Meas M on (Apply E on psi) ⟶ -1.
```

Proving `errors_detectable_correct` involves three case studies:

- When $E = X_1$, we can use $S = Z_1Z_2$ to detect it.
- When $E = X_2$, we can use $S = Z_2Z_3$ to detect it.[44]
- When $E = X_3$, we can use $S = Z_2Z_3$ to detect it.

Then we build the structure of the code using the `ErrorDetectingCode` in Section 5.3.

```
Definition BitFlipCode :=
  BuildDetectingCode stabiliser_set_correct errors_detectable_correct.
```

The structure `BitFlipCode` carries the information of code space, stabilisers, and detectable errors. It also carries two evidence about stabiliser set and detectable errors. To see this, use `Print BitFlipCode`.

```
BitFlipCode = {
  dim  := 3;
  code := psi;
  obs  := SyndromeMeas;
  err  := BitFlipError;
  ob1  := stabiliser_set_correct;
  ob2  := errors_detectable_correct
}
```

[43] The code is formatted for readability

[44] Actually, Z_1Z_2 can also detect error X_2. But for proving this, one existential evidence already suffices.

Then we turn to verify the code is a correcting code: every error $E \in \mathcal{E}$ has a unique syndrome when being measured by $\mathcal{S}$. We can use the predicate `error_identified_uniquely` defined in Section 5.4.2 for this purpose:

```
Theorem bit_flip_code_unique_syndrome:
  error_identified_uniquely BitFlipCode.
```

An unfolded version of this theorem is:

```
Theorem bit_flip_code_unique_syndrome':
  ∀ E1 E2 : ErrorOperator 3,
  E1 ∈ BitFlipCode.err →
  E2 ∈ BitFlipCode.err →
  E1 ≠ E2 →
  ∃ M : PauliObservable 3, M ∈ BitFlipError.obs ∧ distinguishable_by E1 E2 M.
```

Let's translate this theorem to natural language: for every pair of detectable errors $\langle E_1, E_2 \rangle$ in the bit-flip code, if $E_1 \neq E_2$, then there exists a stabiliser $S \in \mathcal{S}$ such that has different measurement result.

Proving theorem `bit_flip_code_unique_syndrome` requires enumerating every pair of errors and performing case analysis. For example, to show that $\langle X_1 X_2 \rangle$ can be distinguished by the code, we showed that the stabiliser $Z_2 Z_3$ has a different measurement result on error state $X_1|\psi\rangle$ and $X_2|\psi\rangle$. Since there are 3 detectable errors, there are 6 cases required to prove the theorem. Luckily, the proof can be mechanised. We made an automation tactic `prove_detectable E M` to mechanically check E can be detected by M.

Finally, we build the structure of the correcting code.

```
Definition BitFlipCorrectingCode :=
  BuildCorrectingCode BitFlipCode bit_flip_code_unique_syndrome.
```

Hence, the `BitFlipCorrectingCode` is a certified error correcting code that can detect and recover from any single-qubit bit-flip error.

6.1.4 Verification of Undetectable and Indistinguishable Errors

We now verify the limitation of the three-qubit bit-flip code:

- This code cannot detect phase-flip errors.
- This code cannot distinguish between certain single- and multi-qubit bit-flip errors. Particular, the $X_1 \coloneqq XII$ error and the $X_2 X_3 \coloneqq IXX$ error.

By verifying these two examples, we want to analyse the strengths and weaknesses of our stabiliser formalism.

Phase-flip errors are Undetectable

Recall Table 1, a single-qubit phase-flip error is represented by a Pauli operator Z, since

$$Z(\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle - \beta|1\rangle \tag{115}$$

For any state $|\psi\rangle = \alpha|000\rangle + \beta|111\rangle$ in this code, a phase-flip error will change it to $|\psi\rangle' = \alpha|000\rangle - \beta|111\rangle$. Hence, we define the error as `Z1` and verify that the effect of the error is expected:

```
Definition Z1: ErrorOperator 3 := [p Z, I, I].

Fact phase_flip_error_effect:
  Apply Z1 on psi = (α * |0,0,0⟩ + -1 * β *|1,1,1⟩).
```

Then, we verify this code cannot detect the error:

```
Fact undetectable_phase_flip_0:
  undetectable BitFlipCode Z1.
```

The predicate `undetectable` is equivalent to saying that, using any stabilisers $S \in \mathcal{S}$ from the code $\mathcal{C}$, measuring S on error state $Z_1|\psi\rangle$ yields $+1$. Translate to Coq:

```
Fact undetectable_phase_flip_0':
  ∀ M : ErrorOperator 3,
  M  ∈ BitFlipCode.obs → Meas M on (Apply Z1 on BitFlipCode.code) ⟶ 1
```

To prove this fact, we use the `stabiliser_undetectable_error` lemma in Section 5.6. It reduces the proof to simply check $\forall M, M \cdot Z_1 = Z_1 \cdot M$, which is automatically carried out by reflection.

Indistinguishable Bit-Flip Errors

A known fact about the three-qubit bit-flip code is that it can only correct a single bit-flip error. Let us verify this fact by using our Coq formalism.

Consider two errors in Pauli operator representation: the $X_1 := XII$ and $X_{23} := IXX$. The former one is a bit-flip on the first qubit, and the latter flips the second and the third qubits. We are going to verify this fact by showing that these two errors are indistinguishable under this code. Since any recovery is only possible when an error can be located precisely.

We first define the error operator X_{23} with a proof of its action:

$$\alpha|000\rangle + \beta|111\rangle \overset{X_{23}}{\to} \alpha|011\rangle + \beta|100\rangle \tag{116}$$

```
Definition X23 : PauliOperator 3 := [p I, X, X].

Lemma apply_X23_effect:
  Apply X23 on psi = (α .* |0,1,1⟩ .+ β .* |1,0,0⟩).
```

Finally, we verified that these two errors are indistinguishable under the three-qubit bit-flip code (See the definition of indistinguishable errors in Definition 2.38 and formalism in Section 5.5)

```
Fact indistinguishable_X1_X23:
  indistinguishable BitFlipCode X1 X23.
```

The fact that X_1 and X_{23} are indistinguishable indicates that this code cannot handle more than one bit-flip errors.

Statistics The verification of a certified three-qubit bit-flip code is about 140 lines of code in total, of which about 70 lines are proofs.

6.1.5 Comparison with SQIR-QECC Examples

Comparison with Encoding Program As mentioned earlier, we use a 2^3 dimensional Hilbert space, while the SQIR-QECC examples use a 2^5 dimensional Hilbert space. This is because we use the standard stabiliser formalism and thus do not require discussing ancillary qubits when reasoning about encoding.

```
(* our version *)
Theorem encode_correct :
  (uc_eval encode) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ |0,0⟩ )
  = α .* |0,0,0⟩ .+ β .* |1,1,1⟩.

(* sqir-only version *)
Theorem encode_correct : ∀ (α β : C),
  (uc_eval encode) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ |0,0,0,0⟩ )
  = α .*|0,0,0,0,0⟩ .+ β .* |1,1,1,0,0⟩.
```

Therefore, compared to the SQIR-QECC examples, we have a reduction in dimension by the stabiliser formalism.

Comparison with Detectability and Correctability The SQIR-QECC examples of the detectability and Correctability look quite different from us. The main reason is that the SQIR-QECC examples employ the measurement-free quantum error correction (MF QEC) technique, and we stick to the standard QEC implementation. This means what we're verifying is essentially a different quantum program.

To show this, let's compare the correctness theorems side-by-side:

```
(* stabiliser based version *)
Theorem bit_flip_code_unique_syndrome:
  ∀ E1 E2 : ErrorOperator 3,
  E1 ∈ BitFlipCode.err →
  E2 ∈ BitFlipCode.err →
  E1 ≠ E2 →
  ∃ M : PauliObservable 3, M ∈ BitFlipError.obs ∧ distinguishable_by E1 E2 M.

(* SQIR only version based on MF QEC *)
Definition error_recover_correct (e : error) : ∀ (α β : C),
  (uc_eval (apply_error e; recover)) × (α .* |0,0,0,0,0⟩ .+ β .* |1,1,1,0,0⟩)
  = (α .* |0,0,0⟩ .+ β .* |1,1,1⟩) ⊗ (error_syndrome e).
```

The SQIR-QECC example of three qubit-code verifies that the unitary evolution `apply_error e; recover` can maintain the code space in the presence of error. Our Coq-QECC-based verification states that all errors have a unique syndrome, and implicitly, these errors can be recovered by this code (See Section 5.4). We cannot present a similar theorem like the SQIR-QECC examples because

of the involvement of measurements — Measurements are essentially non-unitary, and hence cannot be represented as a unitary evolution.

Summary We summarise this fact together with other differences in Table 5.

Aspect	**Coq-QECC Examples**	**SQIR-QECC Examples**
Implementation	Standard QEC[45]	MF QEC[46]
Proof Strategy	Reason about group actions	Reason about unitary evolutions
Qubits Used	3 Qubits (No ancilla)	5 Qubits (need 2 ancillary qubits)
Correctable Errors	Verified	Verified
Detectable Errors	Verified	Not Verified[47]
Undetectable Errors	Verified	Not Verified
Indistinguishable Errors	Verified	Not Verified

Table 5: The comparison of certified three-qubit code in Coq-QECC and SQIR-QECC examples

For building proofs, our verification is based on reasoning about group actions of stabilisers and errors. This allows us to use the compose actions through group multiplication (see Theorem 2.28) and use error detecting condition (see Section 5.6) to reduce or avoid reasoning in Hilbert space. In contrast, the SQIR-QECC examples depend entirely on reasoning about unitary evolutions in Hilbert space.

Furthermore, we classify quantum codes into two structures: the error detecting code and error correcting code. While in this SQIR-QECC instance of formalism, the verification is only about correctability and lacks support for validating detecting related properties. Detecting-only codes have been extensively studied in the literature. Comparing to correcting codes, detecting-only codes in general can carry more information using the same amount of physical qubits. Therefore, they are favoured in scenarios such as quantum communication. For example, the quantum key distribution (QKD) protocol uses a detecting code to check for data tampering or transmission errors [37]. Hence, the ability to formally verify detecting-only codes is another advantage of adopting the stabiliser formalism.

Lastly, our stabiliser-based verification also supports reasoning the limitation of this code — the undetectable and indistinguishable errors — which is not presented in the SQIR-QECC examples. Understanding which errors a code cannot detect or correct gives a complete characterisation of its behaviour. For example, in practice, different codes are chosen based on the *noise model* of the

[45]Measurement-based Quantum Error Correction

[46]Measurement Free Quantum Error Correction

[47]The MF QEC framework is essentially designed for recovery from errors, and lacks a mechanism for analysing detecting-only codes.

hardware [34]. Hence, verifying the limitations helps assess whether a code is suitable for a given physical device.

For both versions, the main theorem takes around 30 lines of proof. However, we want to highlight that a direct comparison of lines of proof is not meaningful in this context, as the proof goals differ. In addition, we use ssreflect proof language, while the SQIR-QECC examples stick to tactic language. Hence, comparison of lines of proof cannot imply the effectiveness of different approaches.

6.2 Verifying Key Properties of the Nine-Qubit Shor's Code

To evaluate the scalability of our formal analysis approach, it would be valuable to apply it to a larger code base. One such candidate is Shor's code (see Section 2.3.5), which encodes one logical qubit using nine physical qubits. Unlike the bit-flip code, Shor's code can detect and correct any arbitrary single-qubit error—namely, any linear combination of bit-flip and phase-flip errors.

Unlike the previous case of the three-qubit code, where the verification goals overlap, this section shifts focus to verify properties that were previously difficult to establish due to scalability limitations. We start by reviewing what has already been verified in SQIR-QECC (Section 6.2.1). Then, we present a new property that hadn't been verified in prior work before due to scalability — but which we were able to verify using our Coq-QECC.

6.2.1 Certified Encoding and Decoding Programs in SQIR-QECC

The SQIR-QECC examples are not part of our contribution. In the SQIR repository, a formalism of Shor's code is provided by researchers who are external to our study.[48] However, the SQIR-QECC formalism avoids discussing any error-correcting properties. Instead, this example focuses on verifying the encode and decode circuit programs. To see this, let's review their main theorem.[49]

```
Definition encode : base_ucom 9.
Theorem encode_correct : ∀ (α β : C),
  (uc_eval 9 encode) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ 8 ⨂ |0⟩)
   = encoded α β.

Definition decode : base_ucom 9.
Theorem decode_correct : ∀ (α β : C),
  (uc_eval 9 decode) × encoded α β
  = (α .* |0⟩ .+ β .* |1⟩) ⊗ 8 ⨂ |0⟩.

Definition shor (e : error) : base_ucom 9 :=
  encode;
  apply_error e;
  decode.
Theorem shor_correct (e : error) : ∀ (α β : C),
  (uc_eval (shor e)) × ((α .* |0⟩ .+ β .* |1⟩) ⊗ 8 ⨂ |0⟩)
  = (α .* |0⟩ .+ β .* |1⟩) ⊗ ancillae_for e.
```

[48] See NineQubitCode.v in https://github.com/inQWIRE/SQIR/blob/9081860ff5f064f6e866e0e817532d4ac349df22/examples/error-correction/NineQubitCode.v

[49] The main theorem is at line 925 in NineQubitCode.v.

As the code listing shows, the verification is carried out by a verification on the encoding program `encode` and the decoding program `decode` (where both bodies are omitted for brevity). In the main correctness theorem `shor_correct`, the program under verification (`shor`) does not contain an error recovery procedure, and the error remains in the code as `ancillae_for e`. Hence, this verification does not include error-correcting-related properties. Instead, it provides valuable proof of the encoder and the decoder.

The following explanation is attached in the source code of SQIR-QECC examples, explaining why there is no error recovery procedure involved:

"Attempting to do so (including error recovery) requires 8 additional qubits. This makes the following analysis rougher.[50]"

Indeed, the encoding circuit alone operates over a Hilbert space of dimension $2^9 = 512$. The size of the Hilbert space already causes performance problems. Moreover, if one attempts to verify the code using measurement-free quantum error correction—similar to the approach taken for the three-qubit bit-flip code—an additional 8 ancillary qubits are required to store the syndrome. This will expand the Hilbert space to $2^{9+8} = 131,072$ dimensions. Such a size is impractical for verification techniques that use only Hilbert space representation.

Since both our version (Coq-QECC examples) and the SQIR-QECC examples use SQIR to formalise quantum programs, their verification on encoding and decoding is readily reusable for us. Hence, in our examples, we focus primarily on what has not been verified: the detectable and correctable errors of the nine-qubit Shor's code.

6.2.2 Verifying Distinguishability of Pauli Basis Errors in Coq-QECC

The experience in the SQIR-QECC attempt indicates that if one wants to verify any error-detecting related properties, it is necessary to avoid reasoning using only the Hilbert space representation. We use stabiliser formalism to achieve this. Specifically, we verify the distinguishability of Pauli basis errors on the first qubit by using stabiliser formalism, as shown in the pen-and-paper reasoning presented in Section 2.3.5. The code is in the Example.v in the Coq repository.

We first define the code space of Shor's code (Equation 89). We reuse the code space defined in the SQIR-QECC examples — a subspace in the 2^9 dimensional Hilbert space:

```
Notation L0 := (3 ⊗ (∣0,0,0⟩ .+       ∣1,1,1⟩)).
Notation L1 := (3 ⊗ (∣0,0,0⟩ .+ -1 .* ∣1,1,1⟩)).
Notation norm := (/2 * /√ 2).
(* The code space *)
Definition psi: Vector (2^9) := norm .* (α .* L0 .+ β .* L1).
```

Then, we define a helper property measuring_different$(M, |\psi\rangle_1, |\psi\rangle_2)$, which states that measuring M on $|\psi\rangle_1$ and $|\psi\rangle_2$ produces different results:

[50]Quoted line 496 from NineQubitCode.v

```
Definition measuring_different (M: PauliObservable 9) (psi_1 psi_2: Vector (2^9)) :=
  ('Meas M on psi_1⟶  1) ∧ ('Meas M on psi_2 ⟶ -1) ∨
  ('Meas M on psi_1⟶ -1) ∧ ('Meas M on psi_2 ⟶  1).
```

Then we formalise the Pauli errors $\{X_1, Y_1, Z_1\}$ on the first qubit, and a theorem `pauli_basis_distinguishable` to verify that their syndromes are unique.

```
Definition X1: PauliOperator 9 := [X, I, I, I, I, I, I, I].
Definition Y1: PauliOperator 9 := [Y, I, I, I, I, I, I, I].
Definition Z1: PauliOperator 9 := [Z, I, I, I, I, I, I, I].
Definition PauliErrorBasis := [set X1, Z1, Y1].

Theorem pauli_basis_distinguishable:
  ∀ (E1 E2: ErrorOperator 9),
  E1 ∈ ShorErrorBasis →
  E2 ∈ ShorErrorBasis →
  E1 ≠ E2 →
  let psi_e1 := Apply E1 on psi in
  let psi_e2 := Apply E2 on psi in
  ∃ (M: PauliObservable 9), M ∝1 psi ∧ measuring_different M psi_e1 psi_e2 .
```

The theorem `pauli_basis_distinguishable` is quite complex and worth explaining. Consider a pair $\langle E_1, E_2\rangle \in \{X_1, Z_1, Y_1\}$ and $E_1 \neq E_2$. We denote by $|\psi\rangle_{E1} \coloneqq E_1|\psi\rangle$ as the error state resulting from applying error E_1 on the codeword $|\psi\rangle$; And similarly $|\psi\rangle_{E2} \coloneqq E_2|\psi\rangle$. Then, there is a stabiliser M of Shor's code such that measuring M on error states $|\psi\rangle_{E1}$ and $|\psi\rangle_{E2}$ yields different results.

This code operates on a $2^9 = 512$-dimensional Hilbert space. To keep the verification tractable, we avoid reasoning directly within the Hilbert space. Instead, we prove the theorem by relying on the error detecting condition — and its negation — as defined in Section 5.6. The error detecting condition enables reasoning solely in terms of group operations, without referring to the code space.

Theorem `pauli_basis_distinguishable` sufficiently verifies that the three Pauli-basis errors $\{X, Y, Z\}$ have different syndromes on the first qubit. Since for each pair in $\{X_1, Y_1, Z_1\}$, there is a stabiliser M of the Shor's code that can distinguish them. Hence, we verify the distinguishability on Pauli basis errors (see Definition 2.39) on the first qubit. This result directly shows the code can detect and correct $\{X, Y, Z\}$ errors on the first qubit. In addition, recall Section 2.3.5, this also shows that Shor's code can detect and correct *any* arbitrary error on the first qubit due to the linearity of quantum errors (see Section 2.3.5).

Statistics The verification of distinguishability of Pauli basis errors takes about 200 lines in total, of which 82 lines are proofs. And the main theorem `pauli_basis_distinguishable` is about 20 lines of proof.

6.2.3 How Coq-QECC Overcomes Prior Limitations

As mentioned earlier, the SQIR-QECC formalism for the nine-qubit Shor's code does not include any verification of its error-correcting properties due to the scalability issue. Instead, they focus on the verification of the encoding and decoding programs. And since our version (Coq-QECC-

based Verification) also uses SQIR to represent encoding and decoding programs, their verification is readily reusable for us.

In addition, based on SQIR-QECC examples, we verified an important error-correcting property — the distinguishability of Pauli basis errors on the first qubit. Without Coq-QECC, this property is otherwise hard to verify, as shown by the SQIR-QECC examples Section 6.2.1. Fortunately, by leveraging the error detecting condition, we can reduce the verification goal to a group-algebraic form, which is significantly easier to reason about (see Section 6.2.2). This demonstrates that the stabiliser formalism and Coq-QECC have the potential to effectively verify properties of a large quantum code like Shor's.

6.3 Summary

These two case studies discussed in this chapter display the benefits of Coq-QECC in three main aspects.

Firstly, Coq-QECC supports verifying a broader range of properties relevant to quantum error correction. In the case study on the three-qubit code Section 6.1, we verified the indistinguishable and undetectable errors, which have not been verified by previous work. The ability to verify additional properties of interest enhances the trustworthiness of quantum code implementations.

Secondly, Coq-QECC offers a more principled and modular approach for reasoning quantum codes. It uses Pauli operators to represent discrete quantum errors and observables. And we support combining smaller Pauli operators into larger ones for reusability (See concatenation in Section 4.5). In addition, Coq-QECC introduces abstract structures for error-detecting and error-correcting codes, facilitating clearer specification of code properties. In contrast, prior work (Section 6.1) is based on case-by-case analysis, and cannot be used across different codes.

More importantly, Coq-QECC addresses the scalability challenge in verifying large quantum codes. In Section 6.2, we verify a key error-correcting property of Shor's code, which is deliberately avoided in the prior SQIR-QECC examples Section 6.2.1. Scalability is achieved by substituting the computationally expensive reasoning in Hilbert space with algebraic reasoning over group actions.

We also find that the current development of Coq-QECC can still be improved. We will discuss these limitations together with other aspects in the next chapter (Chapter 7).

Chapter 7

Discussion and Conclusion

7.1 Employed Axioms in Coq-QECC

Axioms are propositions that are assumed to be true. They are essentially not provable from the logic system of Coq and serve as fundamental principles to start reasoning. Usually, we expect axioms to be as less as possible in any formalism. Coq-QECC mainly employs two axioms: functional extensionality and the classical logic hypothesis. While we could justify this choice by noting that SQIR also adopts these axioms [25], we aim to provide a more explanatory justification through a concrete example.

Functional Extensionality A basic design choice in verification is how to decide if two functions are equal. The *axiom of functional extensionality* essentially states that two functions are equal if and only if their values are equal at every argument [41]. We introduced this axiom from the Coq library Coq.Logic.FunctionalExtensionality.

```
Axiom functional_extensionality_dep : ∀ {A} {B : A → Type},
  ∀ (f g : ∀ x : A, B x),
  (∀ x, f x = g x) → f = g.
```

The opposite of functional extensionality is the *definitional equality*, which states that two functions are equal iff they have the same definition. i.e. Two functions are considered unequal if they have different definitions, although they might have the same output for every input.

We need functional extensionality mainly because matrices are formalised as functions (See Section 3.1.2). For example, we can have two different definitions of the identity matrices:

```
Definition I: Square 2 :=
  (fun x y ⇒ if (x == y) && (x < 2) then 1 else 0).

Definition I': Square 2 :=
  match (x, y) with
  | (0, 0) ⇒ 1
  | (0, 1) ⇒ 0
  | (1, 0) ⇒ 0
  | (1, 1) ⇒ 1
  | _      ⇒ 0
  end.
```

Although they are both valid definitions of the 2-dimensional identity matrix I, they are not considered equal unless functional extensionality is assumed. Therefore, the axiom of functional extensionality is essentially required.

Classical Logic

Coq is based on the *constructive logic* (also known as *intuitionistic logic*). In constructive logic, to prove propositions like $\exists x, P(x)$, the only acceptable way is to construct such an x in which proposition $P(x)$ holds. This usually suffices in theories discussing finite objects. However, in mathematical theories, some existential propositions are often proved by showing the opposite assumption is *absurd*. That is to say, instead of proving $\exists x, P(x)$, mathematicians tend to show that $\forall x, \neg P(x)$ leads to a contradiction. This reasoning is supported by the *Law of Excluded Middle*, which is the principle of *Classical Logic*. It says that any proposition is either true or false:

$$\forall P, P \vee \neg P \tag{117}$$

To see this, remember that

$$\neg(\exists x, P(x)) = \forall x, \neg P(x) \tag{118}$$

By the De Morgan law over quantifiers.

An Example of Axiom Usage

We now give a concrete example in Coq-QECC to show why these two axioms are necessary: functional extensionality and classical logic.[51]

```
Lemma inequal_f_2: ∀ {n m : ℕ} (A B: Matrix n m),
  A ≠ B → ∃ x y, A x y ≠ B x y.
```

This lemma states that for all $n \times m$ matrices A and B, if they are not equal, there must exist x, y the cell $A[x][y]$ is not $B[x][y]$. This lemma looks intuitive, but to prove this lemma in constructive logic, the only way is to supply a concrete i and a j. But one will soon find themself stuck — Although we know $A \neq B$, there is no information about their cells. Then how can one supply the x and y? Fortunately, with classical logic, we can do it in the reverse way. i.e. to show that its negation leads to a contradiction. i.e.

$$\forall (x, y : \mathbb{N}), A[x][y] = B[x][y] \to \bot \tag{119}$$

This is provable by using functional extensionality: if for every x, y, $A[x][y]$ and $B[x][y]$ are equal, then we conclude $A = B$ by functional extensionality. Then, $A = B$ is a contradiction of $A \neq B$ in the hypothesis.

In conclusion, to support the reasoning like above, we choose to include functional extensionality and classical logic. QuantumLib and SQIR also include these two axioms. The root cause is how matrices are defined in QuantumLib — a matrix is simply a function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{C}$.[52]

[51] The lemma inequal_f_2 is in ExtraSpecs.v in the Coq-QECC repository. And being used to prove the theorem that all Pauli operators have the eigenvalue ± 1.

[52] see Section 3.1.2 for a detailed explanation of its matrix library

7.2 An Early Unsuccessful Attempt with Coq.Vectors

The formalism discussed in Chapter 5 and Chapter 6 can be referred to as the *MathComp-based version*, as it employs MathComp extensively together with QuantumLib. In fact, the MathComp-based version is not the initial attempt. We have gone through an unsuccessful initial attempt, which we named *QuantumLib-only version*. As the name suggests, we attempt to formalise Pauli groups using only QuantumLib with the base Coq library. The development of the QuantumLib-only version can be found under the `quantumlib-only` folder in the Coq repository.

The main challenge we encountered in the QuantumLib-only version was the use of Coq.Vectors, a dependently-typed vector, to represent n-qubit Pauli groups. In contrast, the MathComp-based version uses the n.-tuple T data type for this purpose. While Coq.Vectors encode the list length directly into the type via dependent typing, MathComp's tuple type takes a different approach: It pairs a standard list with a proof that the list has the expected length. The difference is illustrated in the following definitions[53]:

```
(* Coq.Vectors *)
Inductive vector A : N → Type :=
  | nil : vector A 0
  | cons : ∀ (h:A) (n:N), vector A n → vector A (S n).

(* QuantumLib.ssreflect.tuple *)
Record tuple_of (n : N) T := Tuple
  { tval :> list T; tsize : length tval = n }.
```

Observe that although they serve as the same purpose, they employ different approaches:

- The `vector` type enforces the length constraint through type-checking.
- In contrast, `tuple_of` does not involve type-checking on length directly. Instead, it requires the user to supply an explicit proof (the `tsize`) about the length.

From the perspective of the Curry-Howard correspondence [28], the construction of `v: vector A n` acts as a proof that vector `v` is of length `n`. However, this proof is implicit: Coq does not provide a means to allow the user to build the proof, because the type inference of Coq almost works internally. Instead, for `tuple_of`, users can use Coq's Tactic language to prove the length of a tuple. This provides greater flexibility in proof development.

To illustrate the difficulty, consider defining the multiplication function using Coq.Vectors. i.e. the function `mul_pstring: forall n, PauliString n → PauliString n → PauliString n`. One intuitive recursive definition is

[53] Coq.Vectors type is defined in the Coq standard library, see the official Coq documentation: https://rocq-prover.org/doc/V8.21%2Balpha/stdlib/Coq.Vectors.Vector.html.

```
Notation PauliString n := vector PauliBase n.

Fail Fixpoint mul_pstring {n: N} (a b: PauliString n): PauliString n :=
  match a, b with
  | ha :: ta, hb :: tb ⇒ ha * hb :: mul_pstring(ta tb)
  | [], []             ⇒ []
  end.
```

where notation `[]` represents the `nil` constructor in the inductive definition of vector, and `::` represents `cons` constructor. This definition fails with error message:

```
Error: Non exhaustive pattern-matching: no clause found for patterns [], _ :: _
```

Indicating that the pattern matching is incomplete: The case where `a` to be empty, and `b` is a non-empty vector is missed. However, this branch of case analysis is absurd because we define `a` and `b` to be two vectors of the same size. Although it is easy for humans to tell this fact, it's not the case for Coq's type checker, and more importantly, there is no means to prove it to Coq.

In conclusion, this unsuccessful attempt indicates that, although Coq.Vectors is a popular and standard Coq library, it is not suitable for formalising structures with non-trivial operations defined over it, such as the group multiplication. Understanding this limitation can greatly inform and guide future formalisation efforts.

7.3 limitations and Future Work

Coq-QECC is shown to be capable of verifying quantum error correction codes. But there is still work to be done.

Reducing Proof Obligations The first issue that we have noticed is that verifying that each error has a unique syndrome (see Section 5.5) is rather tedious. This is because the formalism requires evidence that every pair of errors has a different syndrome. This means if a code has n correctable errors, we require users to provide $\binom{n}{2}$ pieces of evidence, by combining every pair of errors. It might be promising to improve this tedious work by taking advantage of advanced operations on sets in MathComp to automate and simplify this checking.[54] In addition to leveraging MathComp, *stabiliser generators* in the stabiliser formalism offer a promising approach to mitigate this issue. Instead of reasoning over the entire stabiliser group, stabiliser generators use a compact set of group generators to describe it. Since the number of generators is typically much smaller than the full group, this approach can significantly reduce the verification burden.

Full Measurement Formalism We also acknowledge several limitations in theory in our work. First, our framework focuses exclusively on stabiliser codes, leaving non-stabiliser codes unsupported. Extending the formalism to handle non-stabiliser codes would be a valuable direction for future research. In addition, certain foundational theories in quantum computing could further enhance the trustworthiness of the framework. For example, in Section 5.1, we formalise only a special case of

[54]Here, *set* means the type `{set T}` in MathComp, which represents sets in mathematics. Not to be confused with the typing term `Set` of Coq.

projective measurement, where the measured states are eigenstates of the observable. This is because in stabiliser formalism, this type of measurement is the only form of measurement directly involved. We did this because this special case is widely accepted as a known fact in the literature (e.g., Proposition 3.11 in [21] and Section 10.5.3 in [37]); And other celebrated verification frameworks like SQIR [25] and CoqQ Section 3.2 also avoid introducing the full measurement postulate. However, a valuable extension would be to formalise the full projective measurement postulate and verify that our case is indeed a valid instance of it.

Verifying More Properties of Interests In this thesis, we solely focus on the error-detecting and correcting properties. But some other properties of quantum codes are also important and worth verifying. One interesting candidate is the *logical operators* of a quantum code. Logical operators are unitary transformations that map one codeword to another within the code space [19]. To implement quantum algorithms on encoded qubits, such operators are essential. We are also interested in verifying more abstract properties in coding theory. The *code distance* is such an abstract property that can compare the general fault-tolerance capability of two different codes [31].

Further Investigation on Scalability Another interesting direction is to verify a larger quantum code to further test the scalability of Coq-QECC. Practical experiments of quantum codes usually use codes with tens of qubits. For example, Google experimented with a 49-qubit stabiliser code on a quantum device [1]. It would be great if we could show Coq-QECC is capable of verifying codes of similar size.

Incorporating Quantum Computing Platforms Finally, we would like to consider combining Coq-QECC with executable quantum programs as another promising direction [10]. For example, we can provide certified translation of a certified quantum code in Coq-QECC directly into an executable OpenQASM program. This would help to reduce the potential human-made error in translation and implementation.

7.4 Summary

In this thesis, we presented Coq-QECC, a Coq formalisation of the quantum stabiliser code formalism. The formalism mainly contains two parts: the formalism of Pauli groups (Chapter 4), and the formalism of quantum stabiliser codes (Chapter 5). Both are made readily reusable as open source software. Coq-QECC is shown to be capable as a tool to verify properties of quantum error correction codes (Chapter 6). Compared to existing work, Coq-QECC is more principled as we take advantage on *Mathematical Components*, which offers us higher-level abstraction and automation based on canonical structures. Coq-QECC is also more scalable for verification tasks. For example, we have verified a critical program property that has been left unverified due the the scalability problem. In addition, we also point out some limitation on current formalisation and propose them as future work (Section 7.3).

Moreover, we believe our investigation provides insight not only for verifying quantum codes, but also for the more general area on applying formal verification to quantum programs.

Bibliography

[1] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681. https://doi.org/10.1038/s41586-022-05434-1

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, and others. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[3] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. https://doi.org/https://doi.org/10.1002/smr.2419

[4] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[5] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. 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. 286–300.

[6] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Math. Comput. Sci.* 9, 1 (2015), 41–62. https://doi.org/10.1007/S11786-014-0181-1

[7] Nicholas Chancellor, Aleks Kissinger, Stefan Zohren, Joschka Roffe, and Dominic Horsman. 2023. Graphical structures for design and verification of quantum error correction. *Quantum science and technology* 8, 4 (2023), 45028.

[8] Christophe Chareton, Dongho Lee, Benoit Valiron, Renault Vilmart, Sébastien Bardin, and Zhaowei Xu. 2023. Formal methods for quantum algorithms. *Handbook of Formal Analysis and Verification in Cryptography*, 319–422.

[9] Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, 1997. 54–56.

[10] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. (2017). Retrieved from https://arxiv.org/abs/1707.03429

[11] Wang Fang and Mingsheng Ying. 2024. Symbolic execution for quantum error correction programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1040–1065.

[12] Marco Fellous-Asiani, Jing Hao Chai, Robert S Whitney, Alexia Auffèves, and Hui Khoon Ng. 2021. Limitations in quantum computing from resource constraints. *PRX Quantum* 2, 4 (2021), 40335.

[13] Qiuyi Feng. 2025. Formal Verification of Quantum Stabilizer Code. In *CoqPL 2025: The Tenth International Workshop on Coq for Programming Languages*, 2025. Retrieved from https://popl25.sigplan.org/details/CoqPL-2025-papers/7/Formal-Verification-of-Quantum-Stabilizer-Code

[14] Alessio Ferrari and Maurice H. Ter Beek. 2022. Formal Methods in Railways: A Systematic Mapping Study. *ACM Comput. Surv.* 55, 4 (November 2022). https://doi.org/10.1145/3520480

[15] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2017. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science* 239, (January 2017), 15–27. https://doi.org/10.4204/eptcs.239.2

[16] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Quantum random access memory. *Physical review letters* 100, 16 (2008), 160501. https://doi.org/10.1103/PhysRevLett.100.160501

[17] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, and others. 2013. A machine-checked proof of the odd order theorem. In *International conference on interactive theorem proving*, 2013. 163–179.

[18] Georges Gonthier. 2008. Formal proof–the four-color theorem. *Notices of the American Mathematical Society* 55, no.11 (January 2008), 1382–1393.

[19] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction*. California Institute of Technology.

[20] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC '96)*, 1996. Association for Computing Machinery, Philadelphia, Pennsylvania, USA, 212–219. https://doi.org/10.1145/237814.237866

[21] Brian Hall. 2013. *Quantum Theory for Mathematicians*. Springer. https://doi.org/10.1007/978-1-4614-7116-5

[22] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

[23] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. Quantum programming languages. *Nature Reviews Physics* 2, 12 (2020), 709–722.

[24] Sascha Heußen, David F. Locher, and Markus Müller. 2024. Measurement-Free Fault-Tolerant Quantum Error Correction in Near-Term Devices. *PRX Quantum* 5, 1 (February 2024), 10333. https://doi.org/10.1103/PRXQuantum.5.010333

[25] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs)*, 2021. Schloss Dagstuhl

– Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–19. https://doi.org/10.4230/LIPIcs.ITP.2021.21

[26] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[27] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[28] William A Howard and others. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44, (1980), 479–490.

[29] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and others. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009. 207–220.

[31] Emanuel Knill and Raymond Laflamme. 1997. Theory of quantum error-correcting codes. *Phys. Rev. A* 55, 2 (February 1997), 900–911. https://doi.org/10.1103/PhysRevA.55.900

[32] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[33] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. 2023. Formal verification of quantum programs: Theory, tools, and challenges. *ACM Transactions on Quantum Computing* 5, 1 (2023), 1–35.

[34] Easwar Magesan, Daniel Puzzuoli, Christopher E Granade, and David G Cory. 2013. Modeling quantum noise for efficient testing of fault-tolerant circuits. *Physical Review A—Atomic, Molecular, and Optical Physics* 87, 1 (2013), 12324.

[35] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[36] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, 2015. 378–388.

[37] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press.

[38] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6, (1976), 319–340. https://doi.org/10.1007/BF00268134

[39] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[40] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017. ACM, 846–858. https://doi.org/10.1145/3009837.3009894

[41] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2025. *Logical Foundations*. Electronic textbook.

[42] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2, (2018), 79. https://doi.org/10.22331/Q-2018-08-06-79

[43] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. Phantom types for quantum programs. In *The Fourth International Workshop on Coq for Programming Languages*, 2018.

[44] Robert Rand. 2018. *Formally Verified Quantum Programming*. University of Pennsylvania.

[45] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[46] Joschka Roffe. 2019. Quantum error correction: an introductory guide. *Contemporary Physics* 60, 3 (2019), 226–245. https://doi.org/10.1080/00107514.2019.1667078

[47] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994. 124–134. https://doi.org/10.1109/SFCS.1994.365700

[48] Peter W. Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A* 52, 4 (October 1995), R2493–R2496. https://doi.org/10.1103/PhysRevA.52.R2493

[49] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29. https://doi.org/10.1145/230514.571645

[50] Peter G. Wolynes. 2009. Some quantum weirdness in physiology. *Proceedings of the National Academy of Sciences* 106, 41 (2009), 17247–17248. https://doi.org/10.1073/pnas.0909421106

[51] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and experience. *ACM computing surveys (CSUR)* 41, 4 (2009), 1–36.

[52] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yuan Xie, and Yufei Ding. 2021. Qecv: Quantum error correction verification. *arXiv preprint arXiv:2111.13728* (2021).

[53] Noson S. Yanofsky and Mirco A. Mannucci. 2008. *Quantum Computing for Computer Scientists*. Cambridge University Press.

[54] Mingsheng Ying. 2012. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49.

[55] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. Coqq: Foundational verification of quantum programs. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 833–865.

[56] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019. 1149–1162.