

ABSTRACT

Title of Dissertation: A VERIFIED SOFTWARE TOOLCHAIN
FOR QUANTUM PROGRAMMING

Kesha Hietala
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Michael Hicks
Department of Computer Science

Quantum computing is steadily moving from theory into practice, with small-scale quantum computers available for public use. Now quantum programmers are faced with an old problem: How can they be sure that their code does what they intend it to do? This dissertation presents encouraging results in the application of *formal methods* to the domain of quantum programming, aiming to *provide a high-assurance software toolchain for quantum programming*. I begin by presenting SQIR, a small quantum intermediate representation deeply embedded in the Coq proof assistant, which has been used to implement and prove correct quantum algorithms such as Grover's search and Shor's factorization algorithm. Next, I present VOQC, a verified optimizer for quantum circuits, which contains state-of-the-art SQIR program transformations and optimizations that I have proved to be semantics-preserving. Finally, I discuss developing work on providing high assurance for a high-level industry quantum programming language, Q#. With this dissertation, I aim to paint a picture of a verified toolchain for quantum programming, where quantum programs can be written and formally verified at the source level, and then compiled to an optimized form that can be executed on a machine, with the guarantee that the compiled code has the same behavior as the source.

A VERIFIED SOFTWARE TOOLCHAIN FOR QUANTUM PROGRAMMING

by

Kesha Hietala

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:

Professor Michael Hicks, Chair/Advisor

Professor Lawrence Washington, Dean's Representative

Professor Andrew Childs

Professor Leonidas Lampropoulos

Professor Robert Rand

Professor Xiaodi Wu

Acknowledgements

I would like to thank my coauthors Mike Hicks, Shih-Han Hung, Liyi Li, Sarah Marshall, Yuxiang Peng, Robert Rand, Kartik Singhal, Runzhou Tao, Finn Voichick, Xiaodi Wu, and Shaopeng Zhu, whose work is represented in this dissertation. I am especially grateful to Mike (my advisor) for his help with improving my writing and presentation skills, Xiaodi for the inspiration to work in the intersection of quantum computing and programming languages, and Robert Rand and Jennifer Paykin for their initial work on the *QWIRE* language, which started me down the path of verifying quantum software in Coq. In each chapter, I emphasize the parts of the work that are mine and give credit for work that I did not do. None of this would have been possible alone.

I would also like to thank the programming languages communities at the University of Minnesota, the University of Maryland, and Microsoft Research for their support. In particular, I benefitted a lot from my interactions with: Arvind Arasu, Dan DaCosta, Sankha Guria, Stephen McCamant, Gopalan Nadathur, Andrew Ruef, Vaibhav Sharma, Mary Southern, Nik Swamy, Ian Sweet, and Eric Van Wyk. Finally, I'd like to thank my husband and my father for their unconditional support.

Table of contents

Acknowledgements	ii
Table of contents	iii
1 Introduction	1
1.1 SQIR: A Small Quantum Language Supporting Verification	2
1.2 voQC: A Verified Optimizer for Quantum Circuits	3
1.3 Q*: Formal Verification for a High-level Quantum Language	5
1.4 Summary	5
2 Background	7
2.1 Formal Verification	7
2.2 Quantum Computing	7
2.2.1 Preliminaries	7
2.2.2 Quantum Programs	9
2.2.3 Compiling Quantum Programs	9
3 A Quantum Intermediate Representation for Verification	11
3.1 Overview	11
3.1.1 Acknowledgements	12
3.2 Syntax and Semantics	12
3.2.1 Unitary SQIR: Syntax	12
3.2.2 Unitary SQIR: Semantics	13
3.2.3 Full SQIR: Adding Measurement	15
3.3 Design Considerations	18
3.3.1 Related Approaches	18
3.3.2 Concrete Indices into a Global Register	20
3.3.3 Extensible Language around a Unitary Core	21
3.3.4 Semantics of Ill-typed Programs	21
3.3.5 Automation for Matrix Expressions	22
3.3.6 Vector State Abstractions	22
3.3.7 Measurement Predicates	25
3.4 Proofs of Quantum Algorithms	25
3.4.1 Grover’s Algorithm	26
3.4.2 Quantum Phase Estimation	27

3.4.3	Shor’s Prime Factorization Algorithm	30
4	A Verified Optimizer for Quantum Circuits	31
4.1	Overview	31
4.1.1	Acknowledgements	32
4.2	A Verified Framework for Optimizing Quantum Programs	33
4.2.1	VOQC Program Representation	33
4.2.2	Program Equivalence	33
4.2.3	Supported Gate Sets	34
4.2.4	Extraction to Executable Code	37
4.3	Optimizations	37
4.3.1	Optimization by Propagation and Cancellation	38
4.3.2	Circuit Replacement	39
4.3.3	Unitary Optimization Scheduling	42
4.3.4	Optimizing Non-Unitary Programs	42
4.4	Circuit Mapping	44
4.4.1	Verified Layout	45
4.4.2	Verified Routing	45
4.4.3	Verified Translation Validation	47
4.4.4	Mapping with Optimization	48
4.5	Experimental Evaluation	48
4.6	Related Work	48
5	Applying Formal Verification to Q#	50
5.1	Overview	50
5.1.1	Acknowledgements	52
5.2	Background	52
5.2.1	Q#: A High-Level Quantum Programming Language	52
5.2.2	F*: A Proof Oriented Programming Language	53
5.3	Quantum Instruction Trees	54
5.3.1	Semantics	56
5.3.2	Q* Combinator Language	56
5.3.3	Translation from Q#	57
5.4	Correctness Properties	59
5.4.1	Correct Qubit Usage	59
5.4.2	Discard Safety	60
5.4.3	Custom Properties	62
6	Conclusion	65
	Bibliography	67

Chapter 1

Introduction

Quantum computers may soon deliver revolutionary improvements in algorithmic performance, but this is only useful if the answers computed by quantum programs are correct. While hardware-level decoherence errors have garnered significant attention, a less recognized obstacle to quantum program correctness is that of human programming errors—bugs. This dissertation aims to provide a solution for how quantum programmers can be assured that their code is bug-free.

Standard approaches from the classical domain for assuring correctness, such as unit testing and runtime debugging, are of limited use in the quantum setting. For example, consider runtime debugging via print statements: In a quantum program, printing the value of a quantum bit requires measuring it (an effectful operation) and printing the returned value. This is akin to randomly and irreversibly coercing a floating point number to a nearby integer—it will give a weakly informative answer and corrupt the rest of the program. Unit tests are of similarly limited value when a program is probabilistic: Many quantum algorithms generate samples over an exponentially large output domain, so characterizing the output distribution may require exponentially many samples. Matters are further complicated by the fact that near-term quantum machines suffer from high error rates, which makes it difficult to distinguish between an unexpected output due to machine error from one due to programmer error. Simulating quantum programs on a classical computer holds some promise (and simulators are bundled with most quantum software packages) but it requires resources up to exponential in the number of qubits being simulated.

An attractive solution to these challenges is to use techniques from the field of programming languages, such as induction, algebraic reasoning and equational rewriting, to *mathematically prove* that a quantum program is correct. This approach is called *formal verification*. Unlike simulation, in formal verification the state of the quantum system is represented *symbolically*, meaning that proofs are often independent of the number of qubits referenced in the program.

This dissertation aims to demonstrate that:

Techniques for classical program verification can be adapted to the quantum setting, allowing for the development of high-assurance quantum software, without sacrificing performance or programmability.

1.1 SQIR: A Small Quantum Language Supporting Verification

This dissertation begins by presenting SQIR (pronounced “squire”), a *small quantum intermediate representation* deeply embedded in the Coq proof assistant [Coq19]. The SQIR language is not too different from quantum assembly languages like OpenQASM 2.0 [Cro+17] and Quil [SCZ16], but has a carefully defined formal semantics in terms of linear algebra that allows us to prove properties about SQIR programs.

SQIR’s design has several key features. First, it uses natural numbers in place of variables so that we can naturally index into the program’s vector or matrix state. Using variables directly (e.g., with higher-order abstract syntax [PE88], as in the *QWIRE* [PRZ17] and *Quipper* [Gre+13] languages) necessitates a map from variables to indices, which we find confounds proof automation. Second, SQIR provides two semantics for quantum programs. We express the semantics of a general program as a function between *density matrices*, as is standard (e.g., in QPL [Sel04] and *QWIRE*), since density matrices can represent the *mixed states* that arise when a program applies a measurement operator. However, measurement typically occurs at the end of a computation, rather than within it, so we also provide a simpler *unitary semantics* for (sub-)programs that do not measure their inputs. In this case, a program’s semantics corresponds to a restricted class of matrices that are much easier to work with, especially when employing automation. Other features of SQIR’s design, like assigning an ill-typed program the denotation of the zero-matrix, are similarly intended to ease proof.

We have used SQIR to implement verified versions of several textbook quantum algorithms including *quantum teleportation*, *superdense coding*, *GHZ state preparation* [GHZ89], the *Deutsch-Jozsa algorithm* [DJ92], *Simon’s algorithm* [Sim94], *Grover’s algorithm* [Gro96], *quantum phase estimation* (QPE), and, most recently, *Shor’s factorization algorithm* [Sho94]. These are the most sophisticated quantum algorithms that have been formally verified in any tool to date, providing encouraging evidence that our design is productive.

Related Work Several prior works have had the goal of formally verifying quantum programs. In 2010, Green [Gre10] developed an Agda implementation of the Quantum IO Monad, and in 2015 Boender et al. [BKN15] produced a small Coq quantum library for reasoning about quantum “programs” directly via their matrix semantics. These were both proofs of concept, and were only capable of verifying basic protocols. Rand, Paykin, and Zdancewic [RPZ18] embedded the *QWIRE* programming language in the Coq proof assistant, and used it to verify a variety of simple programs and assertions regarding ancilla qubits [Ran+19]. SQIR reuses parts of *QWIRE*’s Coq development, and takes inspiration and lessons from its design. Concurrently with our work, Chareton et al. [Cha+21] introduced *QBRICKS*, a tool implemented in Why3 [FP13] whose aim is to support mostly-automated verification of quantum algorithms. Their design in many ways mirrors SQIR’s: both tools provide special support for reasoning about unitary programs and the languages are simplified so that

programs have a straightforward translation to their semantics. SQIR and *QBRICKS* have been used to verify similar algorithms (in particular, Grover’s search and QPE), but *QBRICKS* does not support measurement and has not been applied to verify hybrid classical/quantum algorithms like Shor’s factorization algorithm.

1.2 VOQC: A Verified Optimizer for Quantum Circuits

As the name suggests, the initial intended use of SQIR was not as a source language, but as an intermediate language in a compiler. Compilers are an especially important part of the quantum software toolchain because near-term quantum machines are extremely resource limited and require input programs to be aggressively optimized and transformed to satisfy hardware constraints.

Such sophisticated optimizations are hard to get right: Kissinger and Wetering [KW19] discovered mistakes in the optimized outputs produced by the circuit optimizer of Nam et al. [Nam+18], and Nam et al. themselves found that the optimization library they compared against (Amy, Maslov, and Mosca [AMM13]) sometimes produced incorrect results; Amy [Amy18] similarly discovered an optimizer he had recently developed produced buggy results [AAM18]. Each of these bugs was found via *translation validation* (i.e., comparing the semantics of the compiler’s target to its source) well after the original publication, suggesting that testing for semantics-preservation is hard. This is true in the classical setting too [Yan+11], but it is especially challenging in the quantum setting, where determining the semantics of a quantum program by running it on a quantum machine may require exponentially many runs, and simulating it on a classical machine may require storing an exponential-sized state.

As before, an appealing solution is to apply rigorous formal methods to prove that an optimization always preserves the source program’s semantics. For example, CompCert [Ler09] is a compiler for C programs that is written and proved correct using the Coq proof assistant. CompCert includes sophisticated optimizations whose proofs of correctness are verified to be valid by Coq’s type checker, and has been empirically demonstrated to be more robust than similar unverified tools: Using sophisticated testing techniques, researchers found hundreds of bugs in the popular C compilers `gcc` and `clang`, but none in CompCert’s verified core [Yan+11].

We have extended CompCert’s approach to the quantum setting with VOQC (pronounced “vox”), the first *verified optimizer for quantum circuits*. VOQC takes as input a SQIR program and applies a series of optimizations, ultimately producing a result that is compatible with the specified quantum architecture. Importantly, all optimizations in VOQC are guaranteed to be semantics-preserving, meaning that any properties proved about a SQIR program will still hold after VOQC’s optimizations have been applied. Many of VOQC’s optimization are based on those in an optimizer developed by Nam et al. [Nam+18], but we also take inspiration from the Qiskit compiler [Ale+19]. Along with optimizations, we also provide verified utilities for circuit

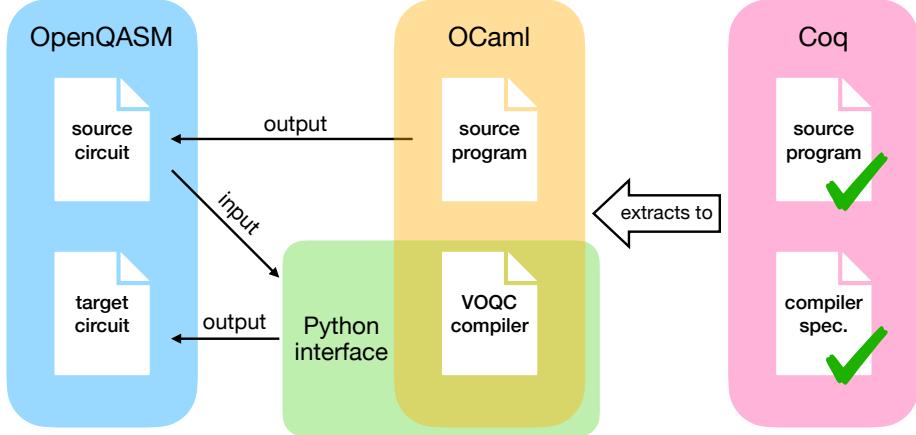


Figure 1.1: Overview of SQIR/VOQC toolchain

mapping, which transforms a SQIR program to satisfy constraints on how qubits may interact on a target architecture.

The SQIR/VOQC toolchain is summarized in Figure 1.1. VOQC optimizations and quantum source programs are specified and formally verified in Coq. Source programs are written in a combination of SQIR, which allows applying quantum operations, and Coq, which supports classical circuit parameters and arbitrary classical computation. Using Coq’s standard code extraction mechanism, we extract VOQC into a standalone OCaml library and source programs into OCaml code that generates SQIR circuits, which we then convert to OpenQASM 2.0 [Cro+17], a standard representation for quantum circuits. We provide a Python wrapper around the OCaml VOQC library, which takes as input an OpenQASM circuit and produces an optimized circuit as output. Our support for Python and OpenQASM allows us to integrate with other Python-based quantum programming frameworks that use OpenQASM, including Qiskit [Ale+19], $t|ket\rangle$ [Cam19], Quil [Rig19b], and Cirq [Dev21].

Using our Python and OCaml libraries, we evaluate the quality of VOQC’s verified optimizations by measuring how well they optimize a large set of benchmark programs. We find that VOQC has competitive performance with comparable unverified tools like Qiskit and $t|ket\rangle$.

Related Work VOQC is the first fully verified optimizer for general quantum programs. Amy, Roetteler, and Svore [ARS17] developed a verified optimizing compiler from source Boolean expressions to reversible circuits and Fagan and Duncan [FD18] verified an optimizer for ZX-diagrams representing Clifford circuits; however, neither of these tools handle general quantum programs. In concurrent work, Shi et al. [Shi+19] developed CertiQ, which uses symbolic execution and SMT solving to verify circuit transformations in the Qiskit compiler. CertiQ is limited to verifying correct application of local equivalences and does not provide a way to describe general quantum states (a key feature of SQIR), which limits the types of optimizations that it can reason about. This also means that it cannot be used as a general tool for verifying quantum programs.

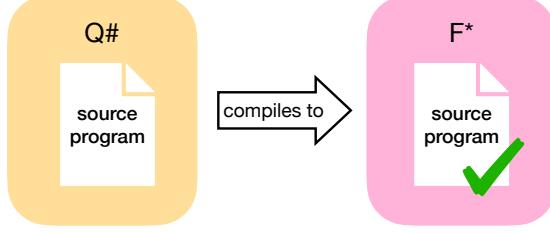


Figure 1.2: Overview of Q^* toolchain

1.3 Q^* : Formal Verification for a High-level Quantum Language

SQIR and VOQC comprise the core of a high-assurance toolchain for quantum programming, supporting verification and optimization of quantum circuits. However, SQIR is a low-level language that does not make plain the high-level structure of quantum algorithms. Meta-programming algorithms in Coq recovers some structure, but Coq was not designed to be a source programming tool (it emphasizes proof) and has no features to support quantum programming, like libraries for common quantum algorithm components or a simulator for quantum programs. Q# [Svo+18; Hei20] is a recent quantum programming language from Microsoft that includes extensive libraries for quantum computing and comes equipped with state-of-the-art simulators, but has no support for formal verification. In an effort to extend our work on quantum formal verification to this higher-level setting, we developed Q^* , a language with a formal semantics for representing Q# programs.

As shown in Figure 1.2, developers write programs in Q#, taking advantage of the many features available in Microsoft’s Quantum Development Kit (QDK). The Q# programs are then translated into Q^* , our novel high-level quantum programming language embedded in the F* proof assistant [Dev22]. We develop a semantics for Q^* programs, allowing us to prove functional correctness, as well as simpler some well-formedness properties not currently enforced by the Q# compiler. Q^* is the first attempt to fit formal verification into an industrial quantum programming language. Prior work on verifying quantum source programs, like SQIR, QWIRE, and QBRICKS, focuses on research languages that have no clear path for integration into an industry setting.

1.4 Summary

This dissertation aims to show that techniques for classical program verification can be adapted to the quantum setting, allowing for the development of high-assurance quantum software, without sacrificing performance or programmability. Towards this goal, we make the following contributions:

1. We present SQIR, a small quantum language designed for proof. We discuss verified SQIR implementations of sophisticated quantum algorithms including

quantum phase estimation, Grover’s search, and Shor’s factorization algorithm. (Chapter 3.)

2. We present VOQC, a verified compiler for quantum circuits that contains implementations of state-of-the-art circuit optimizations. Evaluating on a large set of benchmark programs, we find that VOQC achieves performance comparable with popular unverified tools. (Chapter 4.)
3. We present Q*, the first effort to add the benefits of formal verification to a widely-used industrial quantum programming framework, allowing users to take advantage of convenient high-level programming language features, while still providing correctness guarantees. (Chapter 5.)

Chapter 2

Background

2.1 Formal Verification

Formal methods are techniques to mathematically *prove* that software does what it should, for all inputs; the proved-correct artifact is referred to as *formally certified*. The development of formal methods began in the 1960s when classical computers were in a state similar to quantum computers today: Computers were rare, expensive to use, and had relatively few resources, e.g., memory and processing power. Then, programmers would be expected to do proofs of their programs' correctness by hand. Automating and confirming such proofs has, for more than 50 years now, been a grand challenge for computing research [Hoa03].

While early developments of formal methods led to disappointment [DLP79], the last two decades have seen remarkable progress. Notable successes include the development of the seL4 microkernel [Kle+09] and the CompCert C compiler [Ler09]. For the latter, the benefits of formal methods have been demonstrated empirically: Using sophisticated testing techniques, researchers found hundreds of bugs in the popular mainstream C compilers `gcc` and `clang`, but none in CompCert's verified core [Yan+11]. Formal methods have also been successfully deployed to prove major mathematical theorems (e.g., the Four Color theorem [Gon+08]) and build computer-assisted proofs in the grand unification theory of mathematics [Cas21; Har21].

[**TODO: Add refs to the Verified Software Toolchain project (<https://vst.cs.princeton.edu/>); brief background on Coq**]

2.2 Quantum Computing

2.2.1 Preliminaries

Quantum programs operate over *quantum states*, which consist of one or more *quantum bits* (aka, *qubits*). A single qubit is represented as a vector of complex numbers $\langle \alpha, \beta \rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$. The vector $\langle 1, 0 \rangle$ represents the state $|0\rangle$ while vector $\langle 0, 1 \rangle$ represents the state $|1\rangle$. A state written $|\psi\rangle$ is called a *ket*, following Dirac's notation. We say a qubit is in a *superposition* of $|0\rangle$ and $|1\rangle$ when both α and β are

$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
(a) H matrix	(b) $Rz(\theta)$ matrix	(c) $CNOT$ matrix

Figure 2.1: Matrix representation of common quantum gates

non-zero. Just as Schrodinger’s cat is both dead and alive until the box is opened, a qubit is only in superposition until it is *measured*, at which point the outcome will be 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Measurement is not passive: it has the effect of collapsing the state to match the measured outcome, i.e., either $|0\rangle$ or $|1\rangle$. As a result, all subsequent measurements return the same answer.

Operators on quantum states are linear mappings. These mappings can be expressed as matrices, and their application to a state expressed as matrix multiplication. For example, the matrix representation of the *Hadamard* (H) operator is shown in Figure 2.1(a) and the matrix representation of a z -axis rotation by θ ($Rz(\theta)$) is shown in Figure 2.1(b). Applying H to state $|0\rangle$ yields state $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle$, also written as $|+\rangle$, while applying H to $|1\rangle$ yields $\langle \frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}} \rangle$, also written $|-\rangle$. Many quantum operators are not only linear, they are also *unitary*—the conjugate transpose (or adjoint) of their matrix is its own inverse. This ensures that multiplying a qubit by the operator preserves the qubit’s sum of norms squared. All of the matrices in Figure 2.1 are unitary. Since a Hadamard is its own adjoint, it is also its own inverse: hence $H|+\rangle = |0\rangle$ and $H|-\rangle = |1\rangle$.

A quantum state with n qubits is represented as vector of length 2^n . For example, a 2-qubit state is represented as a vector $\langle \alpha, \beta, \gamma, \delta \rangle$ where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. Because of the exponential size of the complex quantum state space, it is not possible to simulate a 100-qubit quantum computer using even the most powerful classical computer!

n -qubit operators are represented as $2^n \times 2^n$ matrices. For example, the *CNOT* operator over two qubits is represented by the matrix shown in Figure 2.1(c). It expresses a *controlled not* operation—if the first qubit (called the *control*) is $|0\rangle$ then both qubits are mapped to themselves, but if the first qubit is $|1\rangle$ then the second qubit (called the *target*) is negated, e.g., $CNOT|00\rangle = |00\rangle$ while $CNOT|10\rangle = |11\rangle$.

n -qubit operators can be used to create *entanglement*, which is a situation where two qubits cannot be described independently. For example, while the vector $\langle 1, 0, 0, 0 \rangle$ can be written as $\langle 1, 0 \rangle \otimes \langle 1, 0 \rangle$ where \otimes is the tensor product, the state $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}} \rangle$ cannot be similarly decomposed. We say that $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}} \rangle$ is an entangled state.

An important non-unitary quantum operator is *projection* onto a subspace. For example, $|0\rangle\langle 0|$ (in matrix notation $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$) projects a qubit onto the subspace where that qubit is in the $|0\rangle$ state. Projections are useful for describing quantum states after measurement has been performed. We sometimes use $|i\rangle_q\langle i|$ as shorthand for

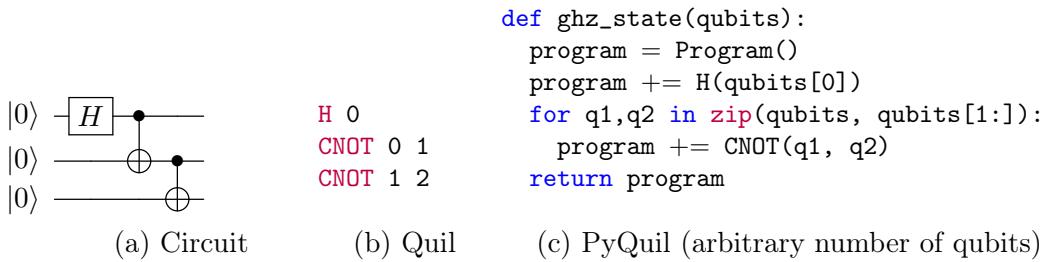


Figure 2.2: Example quantum program: GHZ state preparation

applying the projection $|i\rangle\langle i|$ to qubit q and an identity operation to every other qubit in the state.

2.2.2 Quantum Programs

Quantum programs are typically expressed as circuits, as shown in Figure 2.2(a). In these circuits, each horizontal wire represents a *qubit* and boxes on these wires indicate quantum operators, or *gates*. The circuit in Figure 2.2(a) uses three qubits and applies three gates: the *Hadamard* (H) gate and two *controlled-not* ($CNOT$) gates. Gates can either be unitary (e.g., H and $CNOT$) or non-unitary (e.g., measurement). In software, quantum programs are often represented using lists of instructions that describe the different gate applications. For example, Figure 2.2(b) is the Quil [SCZ16] representation of the circuit in Figure 2.2(a).

In the *QRAM model* [Kni96] quantum computers are used as co-processors to classical computers. The classical computer generates descriptions of circuits to send to the quantum computer and then processes the measurement results. Many high-level quantum programming languages are designed to follow this model. For example, Figure 2.2(c) shows a program in PyQuil [Rig19a], a quantum programming framework embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [GHZ89], which is an n -qubit entangled quantum state of the form

$$|\text{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}).$$

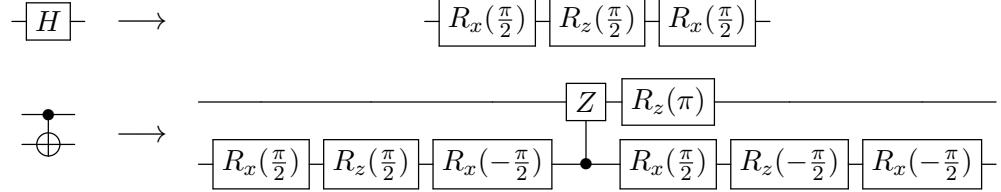
Calling `ghz_state([0,1,2])` returns the Quil program in Figure 2.2(b), which could subsequently be compiled and run on a quantum machine.

2.2.3 Compiling Quantum Programs

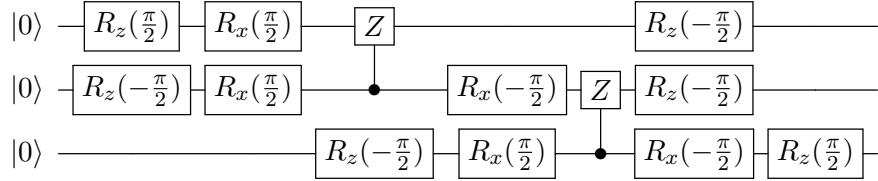
Before a quantum program can be run on a machine, it must be compiled to hardware-specific code that accounts for details like the number of available qubits, native gate set, or connectivity between physical qubits. It is also often desirable to optimize the resulting circuit to reduce gate count, circuit depth, qubit usage, etc.

PyQuil provides access to the quilec compiler [Rig19b], which targets Rigetti's

quantum machines. Rigetti’s hardware natively supports controlled- Z , $R_x(\pm\pi/2)$, and $R_z(\lambda)$ gates. So before the program in Figure 2.2(b) can be executed on a quantum machine, its gates must be translated to this native set. quic uses the following translation rules.



Along with performing gate set translation, quic may also need to account for connectivity constraints, which restrict which qubits can be used together (“interact”) in two-qubit gates. In the program in Figure 2.2, qubits 0 and 1 interact and qubits 1 and 2 interact. Assume that this interaction pattern is allowed by the machine so no additional instructions need be inserted. Finally, quic will perform optimizations that rewrite multiple R_x or R_z instructions acting on the same qubit into fewer instructions, producing the following program¹, which is sent to the hardware for execution.



¹<https://pyquil-docs.rigetti.com/en/v3.1.0/compiler.html>

Chapter 3

A Quantum Intermediate Representation for Verification

3.1 Overview

SQIR is a *small quantum intermediate representation* for describing and verifying quantum programs, embedded in the Coq proof assistant [Coq19]. We initially developed SQIR to be a compiler intermediate representation for our formally verified optimizer for quantum programs VOQC (Chapter 4). However, we quickly realized that it was not so different from languages used to write *source* quantum programs, and that the design choices that eased proving optimizations correct could ease proving source programs correct, too.

To date, we have proved the correctness of implementations of a number of quantum algorithms, including quantum teleportation, Greenberger–Horne–Zeilinger (GHZ) state preparation [GHZ89], the Deutsch–Jozsa algorithm [DJ92], Simon’s algorithm [Sim94], the quantum Fourier transform (QFT), quantum phase estimation (QPE), Grover’s algorithm [Gro96], and, most recently, Shor’s factorization algorithm [Sho94]. Our implementations can be extracted to code that can be executed on quantum hardware or simulated classically, depending on the problem size and hardware limitations.

This chapter presents a detailed discussion of how SQIR’s design supports proofs of correctness. Section 3.2 presents SQIR’s syntax and semantics. Section 3.3 discusses key elements of SQIR’s design and compares and contrasts them to design decisions made in the related tools *QWIRE* [PRZ17], *QBRICKS* [Cha+20], and the Isabelle implementation of quantum Hoare logic [Liu+19a]. SQIR’s overall benefit over these tools is its flexibility, supporting multiple semantics and approaches to proof. Section 3.4 presents the code, formal specification, and proof sketch of Grover’s algorithm, QPE, and Shor’s algorithm, which are the most sophisticated algorithms that we have verified so far.

SQIR is freely available at <https://github.com/inQWIRE/SQIR>.

3.1.1 Acknowledgements

This chapter is primarily based on two papers [Hie+21; Hie+20], which were joint work with Robert Rand, Shih-Han Hung, Liyi Li, Xiaodi Wu, and Michael Hicks. Section 3.4.3 describes joint work with Yuxiang Peng, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu [Pen+22] on verifying Shor’s algorithm. Robert initially conceived the idea of a simplified form of the *QWIRE* language, which developed into SQIR. Shih-Han led our initial effort to verify a quantum program (the Deutsch-Jozsa algorithm) and Liyi contributed to the proof of Simon’s algorithm. Yuxiang led the effort to implement and verify the modular multiplication oracle used in Shor’s algorithm, and he and Runzhou developed the number theory framework needed to prove that factorization reduces to order finding. I developed the bulk of SQIR and the implementation and proofs of quantum phase estimation and Grover’s algorithm. For Shor’s, I developed the infrastructure to extract our SQIR definitions to simulatable OpenQASM code and, along with Robert and Yuxiang, developed the framework to reason about the classical probabilistic components of the algorithm.

3.2 Syntax and Semantics

SQIR is a simple quantum language deeply embedded in the Coq proof assistant. This section presents SQIR’s syntax and semantics. We defer a detailed discussion of SQIR’s design rationale to the next section.

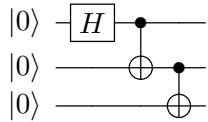
3.2.1 Unitary SQIR: Syntax

SQIR’s *unitary fragment* is a sub-language of full SQIR for expressing programs consisting of unitary gates. (The full SQIR language extends unitary SQIR with measurement.) A program in the unitary fragment has type `ucom` (for “unitary command”), which we define in Coq as follows:

```
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 `useq` constructor sequences two commands; we use notational shorthand `p1 ; p2` for `useq p1 p2`. The two `uappn` constructors indicate the application of a quantum gate to n qubits, where n is 1 or 2. Qubits are identified as numbered indices into a *global qubit register* of size d , which stores the quantum state. Gates are drawn from parameter `U`, which is indexed by a gate’s size. For writing and verifying programs, we use the following `base` set for `U`, inspired by IBM’s OpenQASM 2.0 [Cro+17]:¹

¹It is helpful for proofs to keep `U` small because the number of cases in the proof about a value of type `ucom U d` will depend on the number of gates in `U`. In our work on VOQC (Chapter 4), we define optimizations over a larger gate set that includes common gates like Hadamard, but convert these gates to our `base` set for proof.



(a) Quantum circuit

```
H 0;
CNOT 0 1;
CNOT 1 2
```

```
Fixpoint ghz (n :  $\mathbb{N}$ ) : ucom base n :=
  match n with
  | 0 => I 0
  | 1 => H 0
  | S n' => ghz n'; CNOT (n'-1) n'
  end.
```

(b) SQIR assembly

(c) SQIR meta-program

Figure 3.1: GHZ state preparation in SQIR

```
Inductive base :  $\mathbb{N} \rightarrow \text{Set} :=$ 
| U_R ( $\theta \phi \lambda : \mathbb{R}$ ) : base 1
| U_CNOT : base 2.
```

That is, we have a one-qubit gate U_R (which we write U_R when using math notation), which takes three real-valued arguments, and the standard two-qubit *controlled-not* gate, U_{CNOT} (written $CNOT$ in math notation), which negates the second qubit wherever the first qubit is $|1\rangle$, making it the quantum equivalent of a *xor* gate. The U_R gate can be used to express any single-qubit gate (see Section 3.2.2). Together, U_R and U_{CNOT} form a *universal* gate set, meaning that they can be composed to describe any unitary operation [Bar+95].

Example: SWAP The following Coq function produces a unitary SQIR program that applies three controlled-not gates in a row, with the effect of exchanging two qubits in the register. We define CNOT as shorthand for $\text{uapp2 } U_{\text{CNOT}}$.

```
Definition SWAP d a b : ucom base d := CNOT a b; CNOT b a; CNOT a b.
```

Example: GHZ Figure 3.1(b) is the SQIR representation of the circuit in Figure 3.1(a), which prepares the three-qubit GHZ state [GHZ89]. We describe *families* of SQIR circuits by meta-programming in the Coq host language. The Coq function in Figure 3.1(c) produces a SQIR program that prepares the n -qubit GHZ state, producing the program in Figure 3.1(b) when given input 3. In Figure 3.1(b–c), H and I apply the U_R encodings of the Hadamard and identity gates.

3.2.2 Unitary SQIR: Semantics

Each k -qubit quantum gate corresponds to a $2^k \times 2^k$ unitary matrix. The matrices for our `base` set are:

$$\llbracket U_R(\theta, \phi, \lambda) \rrbracket = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i(\phi+\lambda)} \cos(\theta/2) \end{pmatrix}, \quad \llbracket CNOT \rrbracket = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Conveniently, the U_R gate can encode any single-qubit gate [NC10, Chapter 4]. For instance, two commonly-used single-qubit gates are X (“not”) and H (“Hadamard”). The former has the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ and serves to flip a qubit’s α and β amplitudes; it can be encoded as $U_R(\pi, 0, \pi)$. The H gate has the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, and is often used to put a qubit into superposition (it takes $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$); it can be encoded as $U_R(\pi/2, 0, \pi)$. Multi-qubit gates are easily produced by combinations of $CNOT$ and U_R ; we show the definition of the three-qubit “Toffoli” gate in Section 3.3.6. Keeping our gate set small simplifies the language and enables easy case analysis—and does not complicate proofs. We rarely unfold the definition of gates like X or the three-qubit Toffoli, instead providing automation to directly translate these gates to their intended denotations. Hence, X is translated directly to $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Users can thereby easily extend SQIR with new gates and denotations.

A unitary SQIR program operating on a size- d register corresponds to a $2^d \times 2^d$ unitary matrix. Function `uc_eval` denotes the matrix corresponding to program `c`.

```
Fixpoint uc_eval (d : ℕ) (c : ucom base d) : Matrix (2^d) (2^d) := ...
```

We write $\llbracket c \rrbracket_d$ for `uc_eval d c`. The denotation of composition is simple matrix multiplication: $\llbracket U_1; U_2 \rrbracket_d = \llbracket U_2 \rrbracket_d \times \llbracket U_1 \rrbracket_d$. The denotation of `uapp1` is the denotation of its argument gate, but padded with the identity matrix so it has size $2^d \times 2^d$. To be precise, we have:

$$\llbracket uapp1 \; U \; q \rrbracket_d = \begin{cases} I_{2^q} \otimes \llbracket U \rrbracket \otimes I_{2^{d-q-1}} & q < d \\ 0_{2^d} & \text{otherwise} \end{cases}$$

where I_n is the $n \times n$ identity matrix. In the case of our `base` gate set, $\llbracket U \rrbracket$ is the U_R matrix shown above. The denotation of any gate applied to an out-of-bounds qubit is the zero matrix, ensuring that a circuit corresponds to a zero matrix if and only if it is ill-formed. We likewise prove that every well-formed circuit corresponds to a unitary matrix.

As our only two-qubit gate in the `base` set is `U_CNOT`, we specialize our semantics for `uapp2` to this gate. To compute $\llbracket CNOT \; q_1 \; q_2 \rrbracket_d$, we first decompose the $CNOT$ matrix into $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes I_2 + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes X$. We then pad the expression appropriately, obtaining the following when $q_1 < q_2 < d$:

$$I_{2^{q_1}} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes I_{2^{q_2-q_1-1}} \otimes I_2 \otimes I_{2^{d-q_2-1}} + I_{2^{q_1}} \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes I_{2^{q_2-q_1-1}} \otimes X \otimes I_{2^{d-q_2-1}}.$$

When $q_2 < q_1 < d$, we obtain a symmetric expression, and when either qubit is out of bounds, we get the zero matrix. Additionally, since the two inputs to $CNOT$ cannot be the same, if $q_1 = q_2$ we also obtain the zero matrix.

Example: Verifying SWAP We can prove in Coq that `SWAP 2 0 1`, which swaps the first and second qubits in a two-qubit register, behaves as expected on two unentangled qubits:

```
Lemma swap2: ∀ (φ ψ : Vector 2), WF_Matrix φ → WF_Matrix ψ →
  \llbracket SWAP 2 0 1 \rrbracket_2 × (φ ⊗ ψ) = ψ ⊗ φ.
```

`WF_Matrix` says that ϕ and ψ are well-formed vectors [Ran18, Section 2]. This proof can be completed by simple matrix multiplication. In the full development we prove the correctness of `SWAP d a b` for arbitrary dimension `d` and qubits `a` and `b`.

Example: Verifying GHZ The GHZ state is an n -qubit entangled quantum state of the form $\frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$. In order to verify the `ghz` program, our goal is to show that for any $n > 0$ the circuit generated by `ghz n` produces the corresponding $GHZ(n)$ vector when applied to $|0\rangle^{\otimes n}$:

Lemma `ghz_correct` : $\forall n : \mathbb{N}, n > 0 \rightarrow [\![\text{ghz } n]\!]_n \times |0\rangle^{\otimes n} = GHZ(n)$.

The proof proceeds by induction on n . The $n = 0$ case is trivial as it contradicts the hypothesis. For $n = 1$ we show that H applied to $|0\rangle$ produces the $|+\rangle$ state, which is $GHZ(1)$. In the inductive step, the induction hypothesis says that the result of applying `ghz n'` to the input state `nket n' |0⟩` is the state $(\frac{1}{\sqrt{2}} * |0\rangle^{\otimes n'} + \frac{1}{\sqrt{2}} * |1\rangle^{\otimes n'}) \otimes |0\rangle$. By applying `CNOT (n' - 1) n'` to this state, we show that `ghz (n' + 1) = GHZ(n' + 1)`.

3.2.3 Full SQIR: Adding Measurement

The full SQIR language adds a branching measurement construct inspired by Selinger’s QPL [Sel04]. This construct permits measuring a qubit, taking one of two branches based on the measurement outcome. Full SQIR defines “commands” `com` as either a unitary sub-program, a no-op `skip`, branching measurement, or a sequence of these.

```
Inductive com (U:  $\mathbb{N} \rightarrow \text{Set}$ ) (d :  $\mathbb{N}$ ) : Set :=
| uc : ucom U d → com U d
| skip : com U d
| meas :  $\mathbb{N} \rightarrow \text{com } U \text{ d} \rightarrow \text{com } U \text{ d} \rightarrow \text{com } U \text{ d}$ 
| seq : com U d → com U d → com U d.
```

The command `meas q P1 P2` measures qubit `q` and performs P_1 if the outcome is 1 and P_2 if it is 0. We define non-branching measurement and resetting to a zero state in terms of branching measurement:

```
Definition measure q := meas q skip skip.  

Definition reset q := meas q (X q) skip.
```

As before, we use our `base` set of unitary gates for full SQIR.

Example: Flipping a Coin It is simple to generate a random coin flip with a quantum computer: Use the Hadamard gate to put a qubit into equal superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then measure it.

```
Definition coin : com base 1 := H 0; measure 0.
```

Density Matrix Semantics As discussed in Section 2.2.1, measurement induces a probabilistic transition, so the semantics of a program with measurement is a probability distribution over states, called a mixed state. As is standard [PRZ17; Yin12], we represent such a state using a *density matrix*. The density matrix of a pure state $|\psi\rangle$ is $|\psi\rangle\langle\psi|$ where $\langle\psi| = |\psi\rangle^\dagger$ is the conjugate transpose of $|\psi\rangle$. The density matrix of a mixed state is a sum over its constituent pure states. For example, the density matrix corresponding to the uniform distribution over $|0\rangle$ and $|1\rangle$ is $\frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$.

The semantics $\{\mathbf{P}\}_d$ of a full SQIR program \mathbf{P} is a function from density matrices to density matrices. Naturally, $\{\mathbf{skip}\}_d \rho = \rho$ and $\{\mathbf{P1} ; \mathbf{P2}\}_d = \{\mathbf{P2}\}_d \circ \{\mathbf{P1}\}_d$. For unitary subroutines, we have $\{\mathbf{uc U}\}_d \rho = [\mathbf{U}]_d \rho [\mathbf{U}]_d^\dagger$: Applying a unitary matrix to a state vector is equivalent to applying it to both sides of its density matrix. Finally, using $|i\rangle_q\langle j|$ for $I_{2^q} \otimes |i\rangle\langle j| \otimes I_{2^{d-q-1}}$, the semantics for $\{\mathbf{meas q P1 P2}\}_d \rho$ is

$$\{\mathbf{P1}\}_d(|1\rangle_q\langle 1| \rho |1\rangle_q\langle 1|) + \{\mathbf{P2}\}_d(|0\rangle_q\langle 0| \rho |0\rangle_q\langle 0|)$$

which corresponds to probabilistically applying $\mathbf{P1}$ to ρ with the specified qubit projected to $|1\rangle\langle 1|$ or applying $\mathbf{P2}$ to a similarly altered ρ .

Example: A Provably Random Coin We can now prove that our `coin` circuit above produces the $|1\rangle\langle 1|$ or $|0\rangle\langle 0|$ density matrix (corresponding to the $|1\rangle$ or $|0\rangle$ pure state), each with probability $\frac{1}{2}$.

Lemma `coin_dist` : $\{\mathbf{coin}\}_1 |0\rangle\langle 0| = \frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$.

The proof proceeds by simple matrix arithmetic. $\{\mathbf{H}\} |0\rangle\langle 0|$ is $H|0\rangle\langle 0|H^\dagger = \frac{1}{2}(1 \ 1)$. Calling this ρ_{12} , applying `measure` yields $|1\rangle\langle 1| \rho_{12} |1\rangle\langle 1| + |0\rangle\langle 0| \rho_{12} |0\rangle\langle 0|$, which can be further simplified using the fact $\langle 1| \rho_{12} |1\rangle = \langle 0| \rho_{12} |0\rangle = (\frac{1}{2})$, yielding $\frac{1}{2}|1\rangle\langle 1| + \frac{1}{2}|0\rangle\langle 0|$, as desired.

Nondeterministic Semantics In addition to the density matrix-based semantics, SQIR also supports a *nondeterministic semantics* in which evaluation is expressed as a relation. Given a state $|\psi\rangle$, a unitary program \mathbf{u} will (deterministically) evaluate to $[\mathbf{u}]_d \times |\psi\rangle$. However, `meas q p1 p2` may evaluate to either $\mathbf{p1}$ applied to $|1\rangle\langle 1| \times |\psi\rangle$ or $\mathbf{p2}$ applied to $|0\rangle\langle 0| \times |\psi\rangle$. We use notation $\mathbf{p} / \psi \Downarrow \psi'$ to say that on input ψ program \mathbf{p} nondeterministically evaluates to ψ' .

The advantage of the nondeterministic semantics is that state is represented using a vector $|\psi\rangle$ rather than a density matrix ρ , which makes proofs easier (see Section 3.3.3). However, because the nondeterministic semantics only describes one possible measurement outcome, it is only useful for proving certain types of properties. For example, it can be used to prove the existence of a possible output state or to show that all execution paths result in the same outcome. The following two examples share the latter property.

Example: Resetting a Qubit Consider the following SQIR program, which resets qubit \mathbf{q} to the $|0\rangle$ state.

Definition `reset q = meas q (X q) skip`.

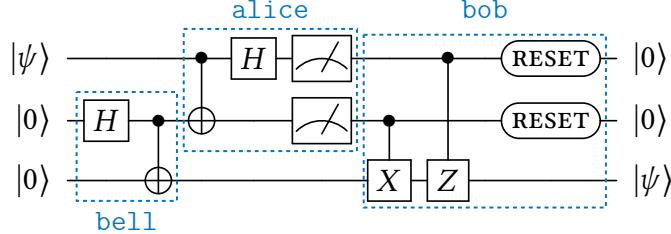


Figure 3.2: Circuit for quantum teleportation. In the standard presentation Bob only acts on the last qubit, given two classical bits as input. In our presentation, Bob (equivalently) performs operations controlled by the first two qubits, which are in a post-measurement classical state. We include the reset operations to simplify our statement of correctness.

Using the density matrix-based semantics, we can prove the following, which says that for any valid density matrix ρ , applying `reset` to ρ will produce the density matrix corresponding to the $|0\rangle$ state.

Lemma `reset_to_zero`: $\forall (\rho : \text{Density } 2), \text{Mixed_State } \rho \rightarrow \{\text{reset}\}_1 \rho = |0\rangle\langle 0|$.

The proof is straightforward:

$$\begin{aligned} [\text{reset}] \rho &= X(|1\rangle\langle 1| \rho |1\rangle\langle 1|)X + I_2(|0\rangle\langle 0| \rho |0\rangle\langle 0|)I_2 \\ &= |0\rangle\langle 1| \rho |1\rangle\langle 0| + |0\rangle\langle 0| \rho |0\rangle\langle 0| \\ &= |0\rangle (\langle 1| \rho |1\rangle + \langle 0| \rho |0\rangle) \langle 0| = |0\rangle (I_1) \langle 0| = |0\rangle\langle 0| \end{aligned}$$

The last line uses the fact that ρ is a valid density matrix (`Mixed_State`), which implies that the entries along its diagonal sum to 1.

Although the proof above is straightforward, it does not give a clear intuition for *why* the program is correct. The simple explanation for why this program is correct is as follows: There are two cases, depending on the result of `meas`. In the case where measurement outputs 0, the remainder of the program is the no-op `skip`, so the output state is $|0\rangle$. In the case where measurement outputs 1, the program applies an `X` gate, which flips the qubit's value, leaving it in final state $|0\rangle$.

The proof using the nondeterministic semantics closely follows this argument: It considers both possible measurement transitions and inspects the output state. The correctness property for the nondeterministic semantics is stated as follows.

Lemma `reset_to_zero`: $\forall (\psi \psi' : \text{Vector } 2), \text{WF_Matrix } \psi \rightarrow \text{reset} / \psi \Downarrow \psi' \rightarrow \psi' \propto |0\rangle$.

This says that *any* output state ψ' is proportional (\propto) to $|0\rangle$.

Example: Quantum Teleportation The goal of quantum teleportation is to transmit a state $|\psi\rangle$ from one party (Alice) to another (Bob) using a shared entangled state. The circuit for quantum teleportation is shown in Figure 3.2 and the corresponding SQIR program is given below.

Definition `bell` : ucom base 3 := H 1; CNOT 1 2.

Definition `alice` : com base 3 := CNOT 0 1 ; H 0; measure 0; measure 1.

```

Definition bob : com base 3 := CNOT 1 2; CZ 0 2; reset 0; reset 1.
Definition teleport : com base 3 := bell; alice; bob.

```

The `bell` circuit prepares a Bell pair on qubits 1 and 2, which are respectively sent to Alice and Bob. Alice applies `CNOT` from qubit 0 to qubit 1 and then measures both qubits and (implicitly) sends them to Bob. Finally, Bob performs operations controlled by the (now classical) values on qubits 0 and 1 and then resets them to the zero state.

Using the density matrix-based semantics, the correctness property for this program says that for any (well-formed) density matrix ρ , `teleport` takes the state $\rho \otimes |0\rangle\langle 0| \otimes |0\rangle\langle 0|$ to the state $|0\rangle\langle 0| \otimes |0\rangle\langle 0| \otimes \rho$.

```

Lemma teleport_correct : ∀ (ρ : Density 2),
WF_Matrix ρ → {teleport}₃ (ρ ⊗ |0⟩⟨0| ⊗ |0⟩⟨0|) = |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ ρ

```

The proof is simple: We perform (automated) arithmetic to show that the output matrix has the desired form.

Under the nondeterministic semantics, we aim to prove that on input $|\psi\rangle \otimes |0,0\rangle$, `teleport` will produce a state that is proportional to $|0,0\rangle \otimes |\psi\rangle$.

```

Lemma teleport_correct : ∀ (ψ : Vector (2^1)) (ψ' : Vector (2^3)),
WF_Matrix ψ → teleport / (ψ ⊗ |0,0⟩) ↓ ψ' → ψ' ∝ |0,0⟩ ⊗ ψ.

```

The first half of the circuit is unitary, so the proof simply computes the effect of applying a H gate, two $CNOT$ gates and another H gate to the input vector state. The two measurement steps then leave four different cases to consider. In each of the four cases, we can use the outcomes of measurement to correct the final qubit, putting it into the state $|\psi\rangle$. Finally, resetting the already-measured qubits is deterministic and leaves us with the desired state.

3.3 Design Considerations

This section describes key elements in the design of SQIR and its infrastructure for verifying quantum programs. To place those decisions in context, we first introduce several related verification frameworks and contrast SQIR’s design with theirs. In summary, SQIR benefits from the use of *concrete indices into a global register* (a common feature in the tools we looked at), support for *reasoning about unitary programs in isolation* (supported by one other tool), and the *flexibility to allow different semantics and approaches to proof* (best supported in SQIR).

3.3.1 Related Approaches

Several prior works have had the goal of formally verifying quantum programs. In 2010, Green [Gre10] developed an Agda implementation of the Quantum IO Monad, and in 2015 Boender et al. [BKN15] produced a small Coq quantum library for reasoning about quantum “programs” directly via their matrix semantics (e.g. Figure 3.3(d)). These were both proofs of concept, and were only capable of verifying basic protocols. More recently, Bordg et al. [BLH20] took a step further in verifying

$\text{box } (\mathbf{x}, \mathbf{y}, \mathbf{z}) \Rightarrow$ $\text{gate } \mathbf{x} \leftarrow H \mathbf{x};$ $\text{gate } (\mathbf{x}, \mathbf{y}) \leftarrow \text{CNOT } (\mathbf{x}, \mathbf{y});$ $\text{gate } (\mathbf{y}, \mathbf{z}) \leftarrow \text{CNOT } (\mathbf{y}, \mathbf{z});$ $\text{output } (\mathbf{x}, \mathbf{y}, \mathbf{z}).$	$\text{SEQ}(\text{SEQ}(\text{PAR}(H, \text{PAR}(I, I)),$ $\text{PAR}(\text{CNOT}, I)),$ $\text{PAR}(I, \text{CNOT}))$	$q_1 := H q_1;$ $q_1, q_2 := \text{CNOT } q_1, q_2;$ $q_2, q_3 := \text{CNOT } q_2, q_3$
(a) $\mathcal{Q}\text{WIRE}$	(b) $\mathcal{Q}\text{BRICKS-DSL}$	(c) QWhile
$(I \otimes \text{CNOT}) \times (\text{CNOT} \otimes I) \times (H \otimes I \otimes I)$		
(d) Matrix expression		

Figure 3.3: Alternate descriptions of the GHZ program in Figure 3.1(a–b)

quantum programs expressed as matrix products (Figure 3.3(d)), providing a library for reasoning about quantum computation in Isabelle/HOL and verifying more interesting protocols like the n -qubit Deutsch-Jozsa algorithm.

In this section, we compare SQIR’s design against three other tools for verified quantum programming that have been used to verify interesting, parameterized quantum programs: $\mathcal{Q}\text{WIRE}$ [Ran18] (implemented in Coq [Coq19]); quantum Hoare logic [Liu+19b] (in Isabelle/HOL [NWP02]); and $\mathcal{Q}\text{BRICKS}$ [Cha+20] (in Why3 [FP13]). We do not include Bordg et al. [BLH20], despite its recency, because it operates one level below the surface programming language, so many issues considered here do not apply. Bordg et al.’s library is similar to the quantum libraries developed for $\mathcal{Q}\text{WIRE}$ and the quantum Hoare logic. All matrix formalisms provided by Bordg et al. are also available in $\mathcal{Q}\text{WIRE}$ ’s library, which we re-use and extend in SQIR.

QWIRE The $\mathcal{Q}\text{WIRE}$ language [PRZ17; RPZ18] originated as an embedded circuit description language in the style of Quipper [Gre+13] but with a more powerful type system. Figure 3.3(a) shows the $\mathcal{Q}\text{WIRE}$ equivalent of the SQIR program in Figure 3.1(b). $\mathcal{Q}\text{WIRE}$ uses variables from the host language Coq to reference qubits, an instantiation of higher-order abstract syntax [PE88]. To describe the GHZ circuit, the $\mathcal{Q}\text{WIRE}$ program uses variables \mathbf{x} , \mathbf{y} , and \mathbf{z} , while the SQIR program uses indices 0, 1, and 2 to refer to the first, second, and third qubits in the global register. $\mathcal{Q}\text{WIRE}$ does not distinguish between unitary and non-unitary programs, and thus uses density matrices for its semantics. $\mathcal{Q}\text{WIRE}$ has been used to verify simple randomness generation circuits and a few textbook examples [Ran18].

QBRICKS $\mathcal{Q}\text{BRICKS}$ [Cha+20] is a quantum proof framework implemented in Why3 [FP13], developed concurrently with SQIR. $\mathcal{Q}\text{BRICKS}$ provides a domain-specific language (DSL) for constructing quantum circuits using combinators for parallel and sequential composition (among others). Figure 3.3(b) presents the GHZ example written in $\mathcal{Q}\text{BRICKS}$ ’s DSL. The semantics of $\mathcal{Q}\text{BRICKS}$ are based on the *path-sums* formalism by Amy [Amy19a; Amy19b], which can express the semantics of unitary

programs in a form amenable to proof automation. \mathcal{Q} BRICKS extends path-sums to support parameterized circuits. \mathcal{Q} BRICKS has been used to verify a variety of quantum algorithms, including Grover’s algorithm and QPE.

QHL Quantum Hoare logic (QHL) [Yin12] has been formalized in the Isabelle/HOL proof assistant [Liu+19a]. QHL is built on top of the quantum while language (QWhile), which is the quantum analog of the classical while language, allowing looping and branching on measurement results. Figure 3.3(c) presents the GHZ example written in QHL. QWhile does not use a fixed gate set; gates are instead described directly by their unitary matrices. As such, the program in Figure 3.3(c) could instead be written as the application of a single gate that prepares the 3-qubit GHZ state. Given that measurement is a core part of the language, QWhile’s semantics are given in terms of (partial) density matrices. A density matrix is *partial* when it may represent a sub-distribution—that is, a subset of the outcomes of measurement.

QHL has been used to verify Grover’s algorithm [Liu+19a]. An earlier effort by Liu et al. [Liu+16] to formalize QHL claimed to prove correctness of QPE, too. However, the approach used a combination of Isabelle/HOL and Python, calling out to Numpy to solve matrix (in)equalities; as such, we consider this only a partial verification effort. We cannot find a proof of QPE in the associated Github repository² and believe that this approach was abandoned in favor of Liu et al. [Liu+19a].

3.3.2 Concrete Indices into a Global Register

The first key element of SQIR’s design is its use of concrete indices into a fixed-sized global register to refer to qubits. For example, in our `SWAP` program (end of Section 3.2.1), `a` and `b` are natural numbers indexing into a global register of size `d`. Expressing the semantics of a program that uses concrete indices is simple because concrete indices map directly to the appropriate rows and columns in the denoted matrix. Moreover, it is easy to check relationships between operations—`x a` and `x b` act on the same qubit if and only if `a = b`. Keeping the register size fixed means that the denoted matrix’s size is known, too.

On the other hand, concrete indices hamper programmability. The `ghz` example in Figure 3.1(c) only produces circuits that occupy global qubits `0...n`; we could imagine further generalizing it to add a lower bound `m` (so the circuit uses qubits `m ... n`), but it is not clear how it could be generalized to use non-contiguous wires. A natural solution, employed by \mathcal{Q} WIRE, is to use host-level variables to refer to *abstract* qubits that can be freely introduced and discarded, simplifying circuit construction and sub-program composition. Unfortunately, abstract qubits significantly complicate formal verification. To translate circuits to operations on density matrices, variables must be mapped to concrete matrix indices. Each time a qubit is discarded, indices undergo a de Bruijn-style shifting.

Similar to SQIR’s use of concrete indices, \mathcal{Q} BRICKS-DSL’s compositional structure makes it easy to map programs to their denotation: The “index” of a gate applica-

²<https://github.com/ijcar2016/propitious-barnacle>

tion can be computed by its nested position in the program. However, this syntax is even less convenient than SQIR’s for programming: Although *QBRICKS* provides a utility function for defining CNOT gates between non-adjacent qubits, their underlying syntax does not support this, meaning that expressions like CNOT 7 2 are translated into large sequences of CNOT gates. QHL is presented as having variables (e.g. `q1` in Figure 3.3(c)), but these variables are fixed before a program executes and persist throughout the program. In the Isabelle formalization, they are represented by natural numbers, making them comparable to SQIR concrete indices.

3.3.3 Extensible Language around a Unitary Core

Another key aspect of SQIR’s design is its decomposition into a unitary sub-language and the non-unitary full language. While the full language (with measurement) is more powerful, its density matrix-based semantics adds unneeded complication to the proof of unitary programs. For example, given the program $U_1; U_2; U_3$, its unitary semantics is a matrix $U_3 \times U_2 \times U_1$ while its density matrix semantics is a function $\rho \mapsto U_3 \times U_2 \times U_1 \times \rho \times U_1^\dagger \times U_2^\dagger \times U_3^\dagger$. The latter is a larger term, with a type that is harder to work with. This added complexity, borne by *QWIRE* and QHL, lacks a compelling justification given that many algorithms can be viewed as unitary programs with measurement occurring implicitly at their conclusion (see Section 3.3.7).

On the other hand, *QBRICKS*’ semantics is based on (higher-order) path-sums, which cannot describe mixed states, and thus cannot give a semantics to measurement. SQIR’s design allows for a “best of both worlds,” utilizing a unitary semantics when possible, but supporting non-unitary semantics when needed. Furthermore, as we show in section 3.3.6, abstractions like path-sums can be easily defined on top of SQIR’s unitary semantics.

3.3.4 Semantics of Ill-typed Programs

We say that a SQIR program is well-typed if every gate is applied to indices within range of the global register and indices used in each multi-qubit gate are distinct. This second condition enforces quantum mechanics’ *no-cloning theorem*, which disallows copying an arbitrary quantum state, as would be required to evaluate an expression like CNOT `q q`. For example, SWAP `d a b` is well-typed if $a < d$, $b < d$, and $a \neq b$.

QWIRE addresses this issue through its linear type system, which also guarantees that qubits are never reused. However, well-typedness is a (non-trivial) extrinsic proposition in *QWIRE*, meaning that many proofs require an assumption that the input program is well-typed and must manipulate this typing judgment within the proof. *QBRICKS* avoids the issue of well-typedness through its language design: It is not possible to construct an ill-typed circuit using sequential and parallel composition. The Isabelle implementation of QHL uses a well-typedness predicate to enforce some program restrictions (e.g. the gate in a unitary application is indeed a unitary matrix), but the issue of gate argument validity is enforced by Isabelle’s type system: Gate arguments are represented as a set (disallowing duplicates) where all elements are valid variables.

In SQIR, ill-typed programs are denoted by the zero matrix. This often means that we do not need to explicitly assume or prove that a program is well-typed in order to state a property about its semantics, thereby removing clutter from theorems and proofs. For example, we can prove symmetry of `SWAP`, i.e. $\text{SWAP } d \ a \ b \equiv \text{SWAP } d \ b \ a$, without any well-typedness constraint because either both sides of the equation are well-typed or both are ill-typed. However, we cannot always avoid well-typedness preconditions. Say we want to prove transitivity of `SWAP`, i.e. $\text{SWAP } d \ a \ c \equiv \text{SWAP } d \ a \ b ; \text{SWAP } d \ b \ c ; \text{SWAP } d \ a \ b$. In this case the left-hand side may be well-typed while the right-hand side is ill-typed. To verify this equivalence, we (minimally) need the precondition $b < d \wedge b \neq a \wedge b \neq c$. We capture these in our `uc_well_typed` predicate, which resembles the `WF_Matrix` predicate (used in the `SWAP` example in Section 3.2.2) that guarantees that a matrix's non-zero entries are all within its bounds. Both conditions are easily checked via automation.

3.3.5 Automation for Matrix Expressions

The SQIR development provides a variety of automation techniques for dealing with matrix expressions. Most of this automation is focused on simplifying matrix terms to be easier to work with. The best example of this is our `gridify` tactic, which rewrites matrix terms into *grid normal form* where matrix addition is on the outside, followed by tensor product, with matrix multiplication on the inside, i.e., $((.. \times ..) \otimes (.. \times ..)) + ((.. \times ..) \otimes (.. \times ..))$. Most of the circuit equivalences available in SQIR (e.g. $\forall a, b, c. \text{CNOT } a \ c ; \text{CNOT } b \ c \equiv \text{CNOT } b \ c ; \text{CNOT } a \ c$) are proved using `gridify`. This style of automation is available in other verification tools too; `gridify` is similar to Liu et al.'s Isabelle tactic for matrix normalization [Liu+19a, Section 5.1]. `QBRICKS` avoids the issue by using path-sums; they provide a matrix semantics for comparison's sake, but do not discuss automation for it.

Some of our automation is aimed at alleviating difficulties caused by our use of *phantom types* [Ran18] to store the dimensions of a matrix, following the approach taken in `QWIRE` [PRZ17]. In our development, matrices have the type `Matrix m n`, where `m` is the number of rows and `n` is the number of columns. One challenge with this definition is that the dimensions stored in the type may be “out of sync” with the structure of the expression itself. For example, due to simplification, rewriting, or declaration, the expression $|0\rangle \otimes |0\rangle$ may be annotated with the type `Vector 4`, although rewrite rules expect it to be of the form `Vector (2 * 2)`. We provide a tactic `restore_dims` that analyzes the structure of a term and rewrites its type to the desired form, allowing for more effective automated simplification.

3.3.6 Vector State Abstractions

To verify that the `SWAP` program has the intended semantics, we can unfold its definition (`CNOT a b; CNOT b a; CNOT a b`) and compute the associated matrix expression. However, while this proof is made simpler by automation like `gridify`, it is still fairly complicated considering that `SWAP` has a simple classical (non-quantum) specification. In fact, this operation is much more naturally analyzed using its action on basis states.

A (*computational*) *basis state* is any state of the form $|i_1 \dots i_d\rangle$ for $i_1, \dots, i_d \in \{0, 1\}$ (so $|00\rangle$ and $|11\rangle$ are basis states, while $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is not). The set of all d -qubit basis states form a basis for the underlying d -dimensional vector space, meaning that any $2^d \times 2^d$ unitary operation can be uniquely described by its action on those basis states.

Using basis states, the reasoning for our `SWAP` example proceeds as follows, where we use $|\dots x \dots y \dots\rangle$ as informal notation to describe the state where the qubit at index a is in state x and the qubit at index b is in state y .

1. Begin with the state $|\dots x \dots y \dots\rangle$.
2. $CNOT a b$ produces $|\dots x \dots (x \oplus y) \dots\rangle$.
3. $CNOT b a$ produces $|\dots (x \oplus (x \oplus y)) \dots (x \oplus y) \dots\rangle = |\dots y \dots (x \oplus y) \dots\rangle$.
4. $CNOT a b$ produces $|\dots y \dots (y \oplus (x \oplus y)) \dots\rangle = |\dots y \dots x \dots\rangle$.

In our development, we describe basis states using `f_to_vec d f` where $d : \mathbb{N}$ and $f : \mathbb{N} \rightarrow \mathbb{B}$. This describes a d -qubit quantum state where qubit i is in the basis state $f(i)$, and `false` corresponds to 0 and `true` to 1. We also sometimes describe basis states using `basis_vector d i` where $i < 2^d$ is the index of the only 1 in the vector. We provide methods to translate between the two representations (simply converting between binary and decimal encodings). For the remainder of the chapter, we will write $|f\rangle$ for `f_to_vec n f` and $|i\rangle$ for `basis_vector n i`, omitting the `n` parameter when it is clear from the context.

We prove a variety of facts about the actions of gates on basis states. For example, the following succinctly describe the behavior of the $CNOT$ and $Rz(\theta)$ gates, where $Rz(\theta) = U_R(0, 0, \theta)$:

```
Lemma f_to_vec_CNOT :  $\forall (d \ i \ j : \mathbb{N}) \ (f : \mathbb{N} \rightarrow \mathbb{B})$ ,  
   $i < d \rightarrow j < d \rightarrow i \neq j \rightarrow$   
  let  $f' := \text{update } f \ j \ (f \ j \oplus f \ i)$  in  
   $\llbracket CNOT \ i \ j \rrbracket_d \times |f\rangle = |f'\rangle$ .
```

```
Lemma f_to_vec_Rz :  $\forall (d \ j : \mathbb{N}) \ (\theta : \mathbb{R}) \ (f : \mathbb{N} \rightarrow \mathbb{B})$ ,  
   $j < d \rightarrow$   
   $\llbracket Rz \ \theta \ j \rrbracket_d \times |f\rangle = e^{i\theta(f \ j)} * |f\rangle$ .
```

Above, `update f i v` updates the value of f at index i to be v (i.e., the resulting function f' satisfies $f'(i) = v$ and $f'(j) = f(j)$ for all $j \neq i$). So $CNOT i j$ has the effect of updating the j^{th} entry of the input state to the exclusive-or of its i^{th} and j^{th} entries. $Rz \theta j$ updates the *phase* associated with the input state.

There are several advantages to applying these rewrite rules instead of unfolding the definitions of $\llbracket CNOT \ i \ j \rrbracket_d$ and $\llbracket Rz \ \theta \ j \rrbracket_d$. For example, these rewrite rules assume well-typedness and do not depend on the ordering of qubit arguments, avoiding the case analysis needed in tactics like `gridify`. In addition, the rule for $CNOT$ above is simpler to work with than the general unitary semantics ($CNOT \mapsto _ \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes _ \otimes I_2 \otimes _ + _ \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes _ \otimes \sigma_x \otimes _$).

As a concrete example of where vector-based reasoning was critical, consider the three-qubit Toffoli gate, which implements a *controlled-controlled-not*, and can be thought of as the quantum equivalent of an *and* gate. It is frequently used in algorithms, but (like all n -qubit gates with $n > 2$) rarely supported in hardware, meaning that it must be decomposed into more basic gates before execution. In practice, we found `gridify` too inefficient to verify the standard decomposition of the gate [NC10, Chapter 4], shown below.

```
Definition TOFF {d} a b c : ucom base d :=
  H c ; CNOT b c ; T† c ; CNOT a c ; T c ; CNOT b c ; T† c ;
  CNOT a c ; CNOT a b ; T† b ; CNOT a b ; T a ; T b ; T c ; H c.
```

However, like `SWAP`, the semantics of the Toffoli gate is naturally expressed through its action on basis states:

```
Lemma f_to_vec_TOFF : ∀ (d a b c : N) (f : N → B),
  a < d → b < d → c < d →
  a ≠ b → a ≠ c → b ≠ c →
  let f' := update f c (f c ⊕ (f a && f b)) in
  [TOFF a b c]d × |f⟩ = |f'⟩.
```

The proof of `f_to_vec_TOFF` is almost entirely automated using a tactic that rewrites using the `f_to_vec` lemmas shown above, since `T` and `T†` are `Rz(PI / 4)` and `Rz(-PI / 4)`, respectively.

The `f_to_vec` abstraction is simple and easy to use, but not universally applicable: Not all quantum algorithms produce basis states, or even sums over a small number of basis states, and reasoning about 2^d terms of the form $|i_1 \dots i_d\rangle$ is no easier than reasoning directly about matrices. To support more general types of quantum states we define indexed sums and tensor (Kronecker) products of vectors.

```
Fixpoint vsum {d} n (f: N → Vector d) : Vector d := ...
Fixpoint vkron n (f: N → Vector 2) : Vector 2n := ...
```

As an example of a state that uses these constructs, consider the output of n parallel Hadamard gates applied to the state $|f\rangle$, which can be written as

$$\begin{aligned} \text{vkron } n (\text{fun } i \Rightarrow \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(i)}|1\rangle)) \quad \text{or} \\ \frac{1}{\sqrt{2^n}} * (\text{vsum } 2^n (\text{fun } i \Rightarrow (-1)^{\text{to_int}(f) \bullet i} * |i\rangle)), \end{aligned}$$

both commonly-used facts in quantum algorithms. For the remainder of the chapter, we will write $\sum_{i=0}^{n-1} f(i)$ for `vsum n (fun i => f i)` and $\bigotimes_{i=0}^{n-1} f(i)$ for `vkron n (fun i => f i)`.

Relationship with Path-sums Our `vsum` and `vkron` definitions share similarities with the *path-sums* [Amy19a; Amy19b] semantics used by `QBRICKS`. In the path-sums formalism, every unitary transformation is represented as a function of the form

$$|x\rangle \rightarrow \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{2\pi i P(x,y)/2^m} |f(x,y)\rangle$$

where $m \in \mathbb{N}$, P is an arithmetic function over x and y , and f is of the form $|f_1(x, y)\rangle \otimes \cdots \otimes |f_m(x, y)\rangle$ where each f_i is a Boolean function over x and y . For instance, the Hadamard gate H has the form $|x\rangle \rightarrow \frac{1}{\sqrt{2}} \sum_{y=0}^1 e^{2\pi i xy/2} |y\rangle$. Path-sums provide a compact way to describe the behavior of unitary matrices and are closed under matrix and tensor products, making them well-suited for automation. They can be naturally described in terms of our `vkron` and `vsum` vector-state abstractions:

```
Definition path_sum (m :  $\mathbb{N}$ ) P f x :=
  vsum  $2^m$  (fun y  $\Rightarrow$   $e^{2\pi i P(x,y)/2^m} * (\text{vkron } m (\text{fun } i \Rightarrow f i x y))$ ).
```

As above, P is an arithmetic function over x and y and $f i$ is a Boolean function over x and y for any i .

3.3.7 Measurement Predicates

The proofs in Section 3.4 do not use the non-unitary semantics directly, but instead describe the probability of different measurement outcomes using predicates `probability_of_outcome` and `prob_partial_meas`.

```
(* Probability of measuring  $\varphi$  given input  $\psi$ . *)
Definition probability_of_outcome {n} ( $\varphi \psi$  : Vector n) : R :=
  let c :=  $(\varphi^\dagger \times \psi)$  0 0 in  $|c|^2$ .

(* Probability of measuring  $\varphi$  on the first n qubits given (n+m) qubit input  $\psi$ . *)
Definition prob_partial_meas {n m} ( $\varphi$  : Vector  $2^n$ ) ( $\psi$  : Vector  $2^{n+m}$ ) :=
   $\|(\varphi^\dagger \otimes I_{2^m}) \times \psi\|^2$ .
```

Above, $\|v\|$ is the 2-norm of vector v and $|c|$ is the complex norm of c . In formal terms, the “probability of measuring φ ” is the probability of outcome φ when measuring a state in the basis $\{\varphi \times \varphi^\dagger, I_{2^n} - \varphi \times \varphi^\dagger\}$.

The *principle of deferred measurement* [NC10, Chapter 4] says that measurement can always be deferred until the end of a quantum computation without changing the result. However, we included measurement in SQIR because it is an important feature of quantum programming languages that is used in a variety of constructs like repeat-until-success loops [PS14] and error-correcting codes [Got10]. QBRICKS also uses measurement predicates, but unlike SQIR does not support a general measurement construct.

3.4 Proofs of Quantum Algorithms

In this section we discuss the formal verification of three influential quantum algorithms: Grover’s algorithm [NC10, Chapter 6], quantum phase estimation [NC10, Chapter 5], and Shor’s factorization algorithm [Sho94]. The proofs and specifications follow the standard textbook arguments.

```

(* Controlled-X with target (n-1) and controls 0, 1, ..., n-2. *)
Fixpoint generalized_Toffoli' n0 : ucom base n :=
  match n0 with
  | 0 | S 0 => X (n - 1)
  | S n0' => control (n - n0) (generalized_Toffoli' n0')
  end.
Definition generalized_Toffoli := generalized_Toffoli' n.

(* Diffusion operator. *)
Definition diff : ucom base n :=
  npar n H; npar n X ;
  H (n - 1) ; generalized_Toffoli ; H (n - 1) ;
  npar n X; npar n H.

(* Main program (iterates applying U_f and diff). *)
Definition body := U_f ; cast diff (S n).
Definition grover i := X n ; npar (S n) H ; niter i body.

```

Figure 3.4: Grover’s algorithm in SQIR. `control` performs a unitary program conditioned on an input qubit, `npar` performs copies of a unitary program in parallel, `cast` is a no-op that changes the dimension in a `ucom`’s type, and `niter` iterates a unitary program.

3.4.1 Grover’s Algorithm

Overview Given a circuit implementing Boolean oracle $f : \{0, 1\}^n \rightarrow \{0, 1\}$, the goal of Grover’s algorithm is to find an input x satisfying $f(x) = 1$. Suppose that $n \geq 2$. In the classical (worst-)case where $f(x) = 1$ has a unique solution, finding this solution requires $O(2^n)$ queries to the oracle. However, the quantum algorithm finds the solution with high probability using only $O(\sqrt{2^n})$ queries.

The algorithm alternates between applying the oracle and a “diffusion operator.” Individually, these operations each perform a reflection in the two-dimensional space spanned by the input vector (a uniform superposition) and a uniform superposition over the solutions to f . Together, they perform a rotation in the same space. By choosing an appropriate number of iterations i , the algorithm will rotate the input state to be suitably close to the solution vector. The SQIR definition of Grover’s algorithm is shown in Figure 3.4.

The SQIR version of Grover’s algorithm is 15 lines, excluding utility definitions like `control` and `npar`. The specification and proof are around 770 lines. The proof took approximately one person-week.

Proof Details The statement of correctness says that after i iterations, the probability of measuring a solution is $\sin^2((2i + 1)\theta)$ where $\theta = \arcsin(\sqrt{k/2^n})$ and k is the number of satisfying solutions to f . Note that this implies that the optimal number of iterations is $\frac{\pi}{4}\sqrt{\frac{2^n}{k}}$.

We begin the proof by showing that the uniform superposition can be rewritten as a sum of “good” states (ψg) that satisfy f and “bad” states (ψb) that do not.

Definition $\psi := \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle$.

Definition $\theta := \arcsin(\sqrt{k/2^n})$.

Lemma `decompose_ψ` : $\psi = (\sin \theta) \psi g + (\cos \theta) \psi b$.

We then prove that U_f and `diff` perform the expected reflections (e.g. $\llbracket \text{diff} \rrbracket_n = -2|\psi\rangle\langle\psi| + I_{2^n}$) and the following key lemma, which shows the output state after i iterations of `body`.

Lemma `loop_body_action_on_unif_superpos` : $\forall i,$

$$\llbracket \text{body} \rrbracket_{n+1}^i (\psi \otimes |- \rangle) = (-1)^i (\sin((2*i+1)*\theta) \psi g + \cos((2*i+1)*\theta) \psi b) \otimes |- \rangle.$$

This property is straightforward to prove by induction on i , and implies the desired result, which specifies the probability of measuring a solution to f .

Lemma `grover_correct` : $\forall i,$

$$\begin{aligned} \text{Rsum } 2^n (\text{fun } z \Rightarrow \text{if } f z \\ \text{then prob_partial_meas } |z\rangle (\llbracket \text{grover } i \rrbracket_{n+1} \times |0\rangle^{n+1}) \\ \text{else } 0) = \\ (\sin((2*i+1)*\theta))^2. \end{aligned}$$

That is, the sum over the probabilities of outcomes z such that $f(z)$ is true is $\sin^2((2i+1)\theta)$. Above, `Rsum` is a sum over real numbers.

3.4.2 Quantum Phase Estimation

Overview Given a unitary matrix U and eigenvector $|\psi\rangle$ such that $U|\psi\rangle = e^{2\pi i \theta} |\psi\rangle$, the goal of quantum phase estimation (QPE) is to find a k -bit representation of θ . In the case where θ can be exactly represented using k bits (i.e. $\theta = z/2^k$ for some $z \in \mathbb{Z}$), QPE recovers θ exactly. Otherwise, the algorithm finds a good k -bit approximation with high probability. QPE is often used as a subroutine in quantum algorithms, most famously Shor’s factoring algorithm [Sho94].

The SQIR program for QPE is shown in Figure 3.5. For comparison, the standard circuit diagrams for QPE and the quantum Fourier transform (QFT), which is used as a subroutine in QPE, are shown in Figure 3.6. The SQIR version of QPE is around 40 lines and the specification and proof in the simple case ($\theta = z/2^k$) is around 800 lines. The fully general case ($\theta \neq z/2^k$) adds about 250 lines. The proof of the simple case was completed in about two person-weeks. When working out the proof of the general case, we found that we needed some non-trivial bounds on trigonometric functions (for $x \in \mathbb{R}$, $|\sin(x)| \leq |x|$ and if $|x| \leq \frac{1}{2}$ then $|2*x| \leq |\sin(\pi x)|$). Laurent Théry kindly provided proofs of these facts using the Coq Interval package [Mel20].

Proof Details The correctness property for QPE in the case where θ can be described exactly using k bits ($\theta = z/2^k$) says that the QPE program will exactly recover z . It can be stated as follows.

```

(* Controlled rotation cascade on n qubits. *)
Fixpoint controlled_rotations n : ucom base n :=
  match n with
  | 0 | 1 => SKIP
  | S n' => controlled_rotations n' ; control n' (Rz (2π / 2n) 0)
  end.

(* Quantum Fourier transform on n qubits. *)
Fixpoint QFT n : ucom base n :=
  match n with
  | 0 => SKIP
  | S n' => H 0 ; controlled_rotations n ; map_qubits (fun q => q + 1) (QFT n')
  end.

(* The output of QFT needs to be reversed before further processing. *)
Definition reverse_qubits n := ...
Definition QFT_w_reverse n := QFT n ; reverse_qubits n.

(* Controlled powers of u. *)
Fixpoint controlled_pow' n (u : ucom base n) k kmax : ucom base (kmax+n) :=
  match k with
  | 0 => SKIP
  | S k' => controlled_pow' n u k' kmax ; niter 2k' (control (kmax - k' - 1) u)
  end.
Definition controlled_powers n (u : ucom base n) k := controlled_pow' n u k k.

(* QPE circuit for program u.
   k = number of bits in resulting estimate
   n = number of qubits in input state *)
Definition QPE k n (u : ucom base n) : ucom base (k + n) :=
  npar k H ;
  controlled_powers n (map_qubits (fun q => k + q) u) k;
  invert (QFT_w_reverse k).

```

Figure 3.5: SQIR definition of QPE. Some type annotations and calls to `cast` have been removed for clarity. `control`, `map_qubits`, `niter`, `npar`, and `invert` are Coq functions that transform SQIR programs; we have proved that they have the expected behavior (e.g. $\forall u. \llbracket \text{invert } u \rrbracket_n = \llbracket u \rrbracket_n^\dagger$).

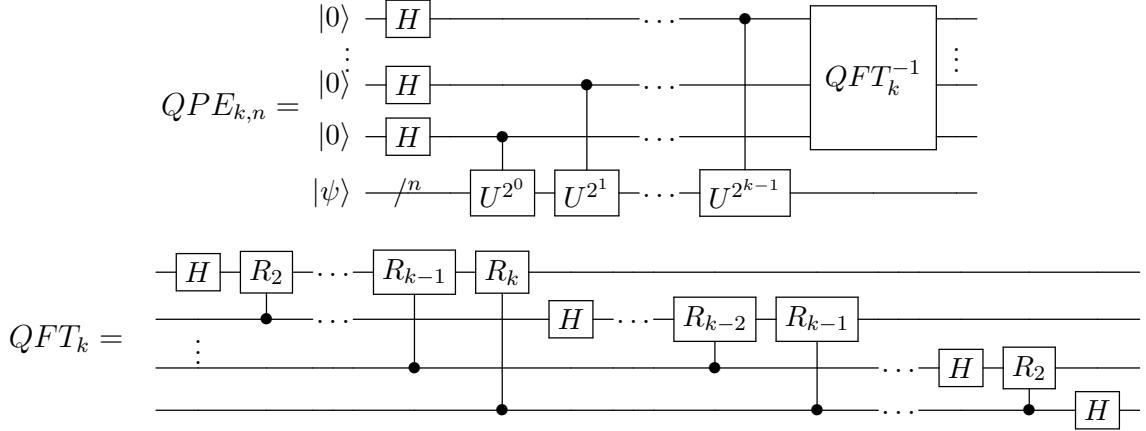


Figure 3.6: Circuit for quantum phase estimation (QPE) with k bits of precision and an n -qubit input state (top) and quantum Fourier transform (QFT) on k qubits (bottom). $|\psi\rangle$ and U are inputs to QPE. R_m is a z -axis rotation by $2\pi/2^m$.

```


Lemma QPE_correct_simplified: ∀ k n (u : ucom base n) z (ψ : Vector 2n),
  n > 0 → k > 1 → uc_well_typed u → WF_Matrix ψ →
  let θ := z / 2k in
  [u]n × ψ = e2πiθ * ψ →
  [QPE k n u]k+n × (|0k⟩ ⊗ ψ) = |z⟩ ⊗ ψ.


```

The first four conditions ensure well-formedness of the inputs. The fifth condition enforces that input ψ is an eigenvector of c . The conclusion says that running the QPE program computes the value z , as desired.

In the general case where θ cannot be exactly described using k bits, we instead prove that QPE recovers the best k -bit approximation with high probability (in particular, with probability $\geq 4/\pi^2$).

```


Lemma QPE_semantics_full : ∀ k n (u: ucom base n) z (ψ : Vector 2n) (δ : ℝ),
  n > 0 → k > 1 → uc_well_typed u → Pure_State_Vector ψ →
  -1 / 2k+1 ≤ δ < 1 / 2k+1 → δ ≠ 0 →
  let θ := z / 2k + δ in
  [u]n × ψ = e2πiθ * ψ →
  prob_partial_meas |z⟩ ([QPE k n u]k+n × (|0k⟩ ⊗ ψ)) ≥ 4 / π2.


```

`Pure_State_Vector` is a restricted form of `WF_Matrix` that requires a vector to have norm 1.

As an example of the reasoning that goes into proving these properties, consider the QFT subroutine of QPE. The correctness property for `controlled_rotations` says that evaluating the program on input $|x\rangle$ will produce the state $e^{2\pi i(x_0 \cdot x_1 x_2 \dots x_{n-1})/2^n} |x\rangle$ where x_0 is the highest-order bit of x represented as a binary string and $x_1 x_2 \dots x_{n-1}$ are the lower-order $n - 1$ bits.

```


Lemma controlled_rotations_correct : ∀ n x,
  n > 1 → [controlled_rotations n]n × |x⟩ = e2πi(x_0 · x_1 x_2 ... x_{n-1})/2n |x⟩.


```

$$\begin{aligned}
& \llbracket \text{controlled_rotations } (n+1) \rrbracket_{n+1} \times |x\rangle \\
&= \llbracket \text{control } x_n (\text{Rz } (2\pi/2^{n+1}) 0) \rrbracket_{n+1} \times \llbracket \text{controlled_rotations } n \rrbracket_{n+1} \times |x\rangle \\
&= \llbracket \text{control } x_n (\text{Rz } (2\pi/2^{n+1}) 0) \rrbracket_{n+1} \times e^{2\pi i(x_0 \cdot x_1 x_2 \dots x_{n-1})/2^n} |x_1 x_2 \dots x_{n-1} x_n\rangle \\
&= e^{2\pi i(x_0 \cdot x_n)/2^{n+1}} e^{2\pi i(x_0 \cdot x_1 x_2 \dots x_{n-1})/2^n} |x_1 x_2 \dots x_{n-1} x_n\rangle \\
&= e^{2\pi i(x_0 \cdot x_1 x_2 \dots x_n)/2^{n+1}} |x_1 x_2 \dots x_{n-1} x_n\rangle
\end{aligned}$$

Figure 3.7: Reasoning used in the proof of `controlled_rotations`. The first step unfolds the definition of `controlled_rotations`; the second step applies the inductive hypothesis; the third step evaluates the semantics of `control`; and the fourth step combines the exponential terms.

We can prove this property via induction on n . In the base case ($n = 2$) we have that x is a 2-bit string $x_0 x_1$. In this case, the output of the program is $e^{2\pi i(x_0 \cdot x_1)/2^2} |x_0 x_1\rangle$, as desired. In the inductive step, we assume that:

$$\llbracket \text{controlled_rotations } n \rrbracket_n \times |x_1 x_2 \dots x_{n-1}\rangle = e^{2\pi i(x_0 \cdot x_1 x_2 \dots x_{n-1})/2^n} |x_1 x_2 \dots x_{n-1}\rangle.$$

We then perform the simplifications shown in Figure 3.7, which complete the proof.

3.4.3 Shor's Prime Factorization Algorithm

[TODO: Overview of Shor's algorithm + proof sketch. I expect this will be 3–4 pages.]

Chapter 4

A Verified Optimizer for Quantum Circuits

4.1 Overview

Compilers are an especially important part of the quantum software toolchain because near-term quantum machines are resource limited: qubits are scarce and have restrictions on how they can interact, and gate pipelines must be short to prevent decoherence. A *verified* compiler guarantees that executable code output by the compiler, despite the optimizations and transformations applied, behaves as specified by the input source program (we say that such a compiler is “semantics preserving”). Similar to the case of determining correctness of quantum programs, semantics-preservation is a difficult property to test for [Yan+11], making formal verification an appealing alternative. The most notable example of a verified compiler (for classical programs) is CompCert [Ler09], an optimizing compiler for C proved correct using the Coq proof assistant.

In this chapter, we apply CompCert’s approach to the quantum setting. We present VOQC (pronounced “vox”), a *verified optimizer for quantum circuits*. VOQC applies a series of optimizations to SQIR programs, ultimately producing a result that is compatible with a specified quantum architecture. For added convenience, VOQC provides translators between SQIR and OpenQASM 2.0 [Cro+17].

At the core of VOQC is a framework for writing transformations of SQIR programs and verifying their correctness (Section 4.2). To ensure that the framework is suitably expressive, we have used it to develop verified versions of a variety of optimizations (Section 4.3). Many are based on those used in an optimizer developed by Nam et al. [Nam+18], a recent, state-of-the-art circuit optimizer. We abstract these optimizations into a couple of different classes, and provide library functions, lemmas, and automation to simplify their construction and proof. We have also verified circuit mapping routines that transform SQIR programs to satisfy constraints on how qubits may interact on a specified target architecture (Section 4.4).

We evaluated the quality of the optimizations we verified in VOQC, and by extension the quality of our framework, by measuring how well it optimizes a set of

benchmark programs, compared to Nam et al. and several other optimizing compilers (Section 4.5). The results are encouraging. On a benchmark of 26 circuit programs developed by Amy, Maslov, and Mosca [AMM13] we find that VOQC reduces total gate count on average by 17.8% compared to 10.1% for IBM’s Qiskit compiler [Ale+19], 10.6% for CQC’s $t|ket\rangle$ [Siv+20], and 24.8% for the cutting-edge research optimizer by Nam et al. [Nam+18]. On the same benchmarks, VOQC reduces T -gate count (an important measure when considering fault tolerance) on average by 41.4% compared to 39.7% by Amy, Maslov, and Mosca [AMM13], 41.4% by Nam et al., and 42.6% by the PyZX optimizer. In sum, VOQC and SQIR are expressive enough to verify a range of useful optimizations, yielding performance competitive with leading unverified compilers.

VOQC is the first fully verified optimizer for general quantum programs. Amy, Roetteler, and Svore [ARS17] developed a verified optimizing compiler from source Boolean expressions to reversible circuits and Fagan and Duncan [FD18] verified an optimizer for ZX-diagrams representing Clifford circuits; however, neither of these tools handle general quantum programs. In concurrent work, Shi et al. [Shi+19] developed CertiQ, which uses symbolic execution and SMT solving to verify circuit transformations in the Qiskit compiler. CertiQ is limited to verifying correct application of local equivalences and does not provide a way to describe general quantum states (a key feature of SQIR), which limits the types of optimizations that it can reason about. This also means that it cannot be used as a tool for verifying general quantum programs. Burgholzer, Raymond, and Wille [BRW20], Kissinger and Wetering [KW19], and Smith and Thornton [ST19] use equivalence checking to compare the semantics of a compiled circuit against its source. Unlike these tools, which add extra overhead and the possibility of failure at runtime, VOQC optimizations are guaranteed to produce a semantically equivalent output for any input.

All code is freely available online:

- Coq definitions and proofs: <https://github.com/inQWIRE/SQIR>
- OCaml library: <https://github.com/inQWIRE/mlvoqc>
- Python bindings and tutorials: <https://github.com/inQWIRE/pyvoqc>
- Benchmarking scripts: <https://github.com/inQWIRE/VOQC-benchmarks>

4.1.1 Acknowledgements

This chapter is primarily based on joint work with Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks [Hie+21]. Xiaodi originally proposed the idea of implementing a verified version of the circuit optimizations in Nam et al. [Nam+18]. My and Robert’s attempts to implement his vision led to the development of SQIR, and eventually VOQC. I implemented and verified all optimizations in VOQC and handled extraction to OCaml/Python and benchmarking. Liyi contributed to the proof of Euler rotations used in single-qubit gate merging (Section 4.3.2).

Table 4.1: Gate sets used in VOQC. r is a real parameter and q is a rational parameter.

Full	Single-qubit gates:	$I, X, Y, Z, H, S, T, S^\dagger, T^\dagger,$ $R_x(r), R_y(r), R_z(r), R_zQ(q),$ $U_1(r), U_2(r, r), U_3(r, r, r)$
	Two-qubit gates:	$CNOT, CZ, SWAP$
	Three-qubit gates:	$CCNOT, CCZ$
RzQ	Single-qubit gates:	$X, H, R_zQ(q)$
	Two-qubit gates:	$CNOT$
IBM	Single-qubit gates:	$U_1(r), U_2(r, r), U_3(r, r, r)$
	Two-qubit gates:	$CNOT$
Mapping	Single-qubit gates:	∞
	Two-qubit gates:	$CNOT, SWAP$

4.2 A Verified Framework for Optimizing Quantum Programs

This section introduces general features of VOQC’s design. We discuss specific optimizations in Section 4.3 and circuit mapping routines in Section 4.4.

4.2.1 VOQC Program Representation

To ease the implementation of and proofs about SQIR program transformations, we developed a framework of supporting library functions that operate on SQIR programs as lists of gate applications, rather than on the native SQIR representation. The conversion code takes a sequence of gate applications in the original SQIR program and *flattens* it so that a program like $(G_1 p; G_2 q); G_3 r$ is represented as the Coq list $[G_1 p; G_2 q; G_3 r]$. The denotation of the list representation is the denotation of its corresponding SQIR program. Examples of the list operations VOQC provides include:

- Finding the next gate acting on a qubit that satisfies some predicate f .
- Propagating a gate using a set of cancellation and commutation rules (see Section 4.3.1).
- Replacing a sub-program with an equivalent program (see Section 4.3.2).
- Computing the maximal matching prefix of two programs.

We verify that these functions have the intended behavior (e.g., in the last example, that the returned sub-program is indeed a prefix of both input programs).

4.2.2 Program Equivalence

The VOQC optimizer takes as input a SQIR program and attempts to reduce its total gate count by applying a series of optimizations. For each optimization, we verify that

it is *semantics preserving* (or *sound*), meaning that the output program is guaranteed to be equivalent to the input program. We say that two unitary programs of dimension d are equivalent, written $U_1 \equiv U_2$, if their denotation is the same, i.e., $\llbracket U_1 \rrbracket_d = \llbracket U_2 \rrbracket_d$. We can then write our soundness condition for optimization function `optimize` as follows.

Definition `sound {G}` (`optimize` : $\forall \{d : \mathbb{N}\}, \text{ucom } G \ d \rightarrow \text{ucom } G \ d$) :=
 $\forall (d : \mathbb{N}) (u : \text{ucom } G \ d), \llbracket \text{optimize } u \rrbracket_d \equiv \llbracket u \rrbracket_d$.

This property is quantified over G , d , and u , meaning that the property holds for *any program that uses any set of gates and any number of qubits*. The optimizations in our development are defined over particular gate sets, described below, but still apply to programs that use any number of qubits. Our statements of soundness also occasionally have an additional precondition that requires program u to be well typed.

We also support two more general versions of equivalence: We say that two circuits are *equivalent up to a global phase*, written $U_1 \cong U_2$, when there exists a θ such that $\llbracket U_1 \rrbracket_d = e^{i\theta} \llbracket U_2 \rrbracket_d$; We say that two circuits are *equivalent up to permutation* if there exist permutation matrices P_1, P_2 such that $\llbracket U_1 \rrbracket_d = P_1 \times \llbracket U_2 \rrbracket_d \times P_2$.¹ Equivalence up to a global phase is useful in the quantum setting because $|\psi\rangle$ and $e^{i\theta} |\psi\rangle$ (for $\theta \in \mathbb{R}$) represent the same physical state. Equivalence up to permutation is useful in the context of circuit mapping (Section 4.4) where inserted *SWAP* gates may change the positions of qubits in the system.

4.2.3 Supported Gate Sets

VOQC supports arbitrary gate sets; the utility functions and properties described above are all parameterized by choice of gate set. The program transformations in Sections 4.3 and 4.4 are defined over the gate sets listed in Table 4.1. Using a custom gate set for each transformation makes writing the transformation cleaner and simplifies the proof of soundness (typically, each gate corresponds to one case in the proof). We summarize which transformations are defined over which gate sets in Figure 4.1.

The *full gate set* consists of a variety of standard quantum gates; it is used for parsing and aims for completeness: Instead of having to translate a T gate in the source OpenQASM program to the semantically equivalent $U_1(\pi/4)$, we can translate it directly to T . Likewise, we can translate the three-qubit *CCNOT* gate directly to *CCNOT*, rather than decomposing it into a series of one- and two-qubit gates (potentially incorrectly). As shown in Figure 4.1, although internally optimizations are defined over different gate sets, in the interface we expose, all functions are defined over the full gate set. We convert between the different gate sets using the rules in Tables 4.2 and 4.3.

The *RzQ gate set*, inspired by the one used by Nam et al. [Nam+18], consists of $\{H, X, R_z Q, CNOT\}$ where $R_z Q(q)$ describes rotation about the z -axis by $q\pi$ for

¹A permutation matrix is a square binary matrix with a single 1 entry in each row and column and 0s elsewhere. Left-multiplying a permutation matrix P by a matrix A (i.e., PA) permutes A 's rows, and right-multiplying (AP) permutes A 's columns.

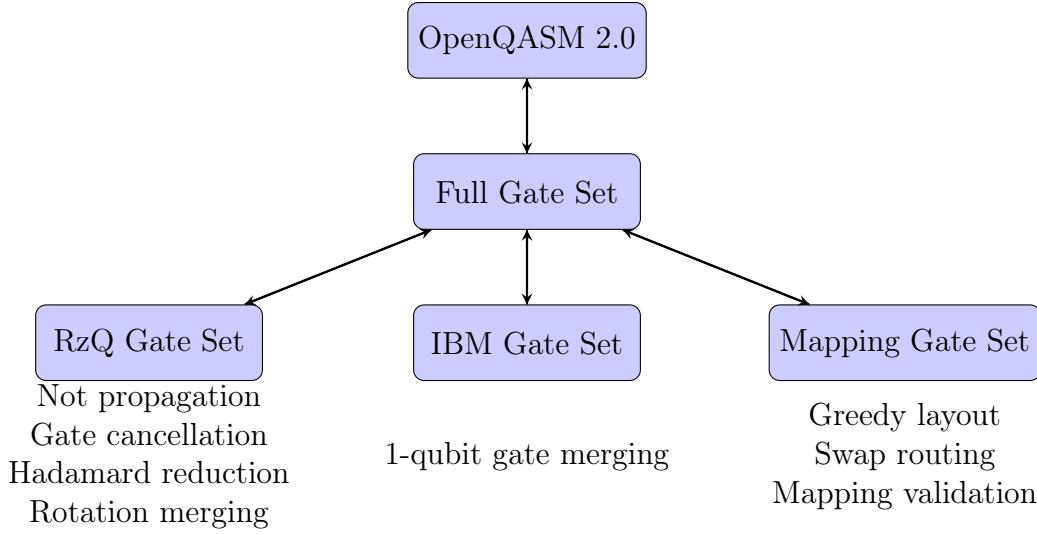


Figure 4.1: Summary of features available in VOQC

Table 4.2: Decompositions of single-qubit gates in the full gate set into gates in the RzQ and IBM gate sets.

Input	RzQ Decomp.	IBM Decomp.
I	$R_z Q(0)$	$U_1(0)$
X	X	$U_3(\pi, 0, \pi)$
Y	$R_z Q(\frac{3}{2})$; X ; $R_z Q(\frac{1}{2})$	$U_3(\pi, \frac{\pi}{2}, \frac{\pi}{2})$
Z	$R_z Q(1)$	$U_1(\pi)$
H	H	$U_2(0, \pi)$
S	$R_z Q(\frac{1}{2})$	$U_1(\frac{\pi}{2})$
T	$R_z Q(\frac{1}{4})$	$U_1(\frac{\pi}{4})$
S^\dagger	$R_z Q(\frac{3}{2})$	$U_1(-\frac{\pi}{2})$
T^\dagger	$R_z Q(\frac{7}{4})$	$U_1(-\frac{\pi}{4})$
$R_x(r)$	H ; $R_z Q(\frac{r}{\pi})$; H	$U_3(r, -\frac{\pi}{2}, \frac{\pi}{2})$
$R_y(r)$	$R_z Q(\frac{3}{2})$; H ; $R_z Q(\frac{r}{\pi})$; H ; $R_z Q(\frac{1}{2})$	$U_3(r, 0, 0)$
$R_z(r)$	$R_z Q(\frac{r}{\pi})$	$U_1(r)$
$R_z Q(q)$	$R_z Q(q)$	$U_1(q\pi)$
$U_1(r)$	$R_z Q(\frac{r}{\pi})$	$U_1(r)$
$U_2(r_1, r_2)$	$R_z Q(\frac{r_2}{\pi} - 1)$; H ; $R_z Q(\frac{r_1}{\pi})$	$U_2(r_1, r_2)$
$U_3(r_1, r_2, r_3)$	$R_z Q(\frac{r_3}{\pi} - 1/2)$; H ; $R_z Q(\frac{r_1}{\pi})$; H ; $R_z Q(\frac{r_2}{\pi} + 1/2)$	$U_3(r_1, r_2, r_3)$

Table 4.3: Decompositions of multi-qubit gates in the full gate set into simpler gates in the full gate set. Decomposition to the RzQ or IBM gate sets can be performed by further applying the rules in Table 4.2. Note that $CNOT$ is primitive in every gate set we support.

Input Gate	Decomposition
$CNOT a b$	$CNOT a b$
$CZ a b$	$H b; CNOT a b; H b$
$SWAP a b$	$CNOT a b; CNOT b a; CNOT a b$
$CCZ a b c$	$CNOT b c; T^\dagger c; CNOT a c; T c; CNOT b c; T^\dagger c;$ $CNOT a c; CNOT a b; T^\dagger b; CNOT a b; T a; T b; T c$
$CCNOT a b c$	$H c; CCZ a b c; H c$

$q \in \mathbb{Q}$. We use a rational parameter for the $R_z Q$ gate instead of a real parameter in an effort to avoid unsound extraction to OCaml. Our Coq formalization relies on an axiomatized definition of real numbers [INR22], so there is no way to extract Coq definitions using reals to OCaml without providing an implementation of real arithmetic. One option (used for the IBM gate set below) is to extract Coq reals to OCaml floats, although this leads to the possibility of floating-point error not accounted for in our proofs.

The *IBM gate set* is the default basis for the Qiskit compiler, and is supported in many quantum compilers. It includes the two-qubit $CNOT$ gate, along with three parameterized single-qubit gates:

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}, \quad U_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix},$$

$$U_3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i(\phi+\lambda)} \cos(\theta/2) \end{pmatrix}.$$

U_3 gates are the most general², and require two quantum ‘‘pulses’’ to implement on hardware. U_2 and U_1 gates are more specialized, but require one and zero pulses, respectively. One interesting property of this gate set (which is not true of the RzQ gate set) is that any sequence of single-qubit gates can be combined into a single gate (see Section 4.3.2). However, during combination, it is not possible to stay in the domain of rational numbers, which forces us to change the parameter type to be real, leading to potential unsoundness in extraction (discussed below).

The *mapping gate set* is used for circuit mapping (Section 4.4). It is parameterized by a choice of single-qubit gates and includes multi-qubit gates $CNOT$ and $SWAP$. In our implementation, we instantiate the mapping gate set using the single-qubit gates from the full gate set.

To compute a program’s denotation, VOQC’s gates must be translated into the $CNOT$ and $R_{\theta, \phi, \lambda}$ gates in SQIR’s base set. In the RzQ set, H , X , and $RzQ(q)$ are

² U_1 and U_2 gates can both be written in terms of U_3 : $U_1(\lambda) = U_3(0, 0, \lambda)$ and $U_2(\phi, \lambda) = U_3(\frac{\pi}{2}, \phi, \lambda)$.

translated into $R_{\pi/2,0,\pi}$, $R_{\pi,0,\pi}$, and $R_{0,0,q\pi}$ respectively. In the IBM set, $U_3(\theta, \phi, \lambda)$ is translated into $R_{\theta,\phi,\lambda}$. We compute the denotation of a program in the full gate set using the rules in Tables 4.2 and 4.3.

4.2.4 Extraction to Executable Code

We use Coq’s standard code extraction mechanism to extract VOQC into a standalone OCaml library. For performance, our library uses OCaml primitives for describing multi-precision rational numbers, maps and sets, rather than the code generated from Coq. We thus implicitly trust that the OCaml implementation of these data types is consistent with Coq’s; we believe that this is a reasonable assumption. A more problematic assumption is that the behavior of OCaml’s 64-bit float type matches the behavior of Coq’s mathematical reals. As mentioned above, we extract the real parameters of the IBM and full gate sets to floats, which may allow for floating-point error not accounted for in our soundness proofs. We would prefer to use a full-precision datatype, like rationals, but the trigonometric functions used to optimize U_2 and U_3 gates are not defined over rationals. We note that existing quantum compilers also use float parameters, so they are equally susceptible to floating-point errors. It is typically assumed that these errors are insignificant in practice due to the fact that near-term machines only support limited precision for input parameters.

In order to make VOQC compatible with existing Python-based frameworks for compiling quantum programs (e.g., Qiskit [Ale+19], pytket [Cam19], Quilc [Rig19b], Cirq [Dev21]), we provide a Python wrapper around the VOQC OCaml library. To interface between Python and OCaml, we wrap the OCaml code in a C library (following standard conventions [INR21]) and call to this C library using Python’s `ctypes` [Pyt21]. For convenience, we have written Python code that makes VOQC look like an optimization pass in IBM’s Qiskit, allowing us to take advantage of this framework’s utilities for quantum programming (e.g., constructing and printing circuits, unverified optimizations and mapping routines).

4.3 Optimizations

VOQC primarily implements optimizations inspired by the state-of-the-art circuit optimizer of Nam et al. [Nam+18]. As such, we do not claim credit for the optimizations themselves. Rather, our contribution is a framework that is sufficiently flexible that it can be used to prove such state-of-the-art optimizations correct. VOQC implements two basic kinds of optimizations: *replacement* and *propagate-cancel*. The former simply identifies a pattern of gates and replaces it with an equivalent pattern. The latter works by commuting sets of gates when doing so produces an equivalent quantum program—often with the effect of “propagating” a particular gate rightward in the program—until two adjacent gates can be removed because they cancel out.

$$\begin{aligned}
X q; H q &\equiv H q; Z q \\
X q; R_z Q(k) q &\cong R_z Q(2-k) q; X q \\
X q_1; CNOT q_1 q_2 &\equiv CNOT q_1 q_2; X q_1; X q_2 \\
X q_2; CNOT q_1 q_2 &\equiv CNOT q_1 q_2; X q_2
\end{aligned}$$

Figure 4.2: Equivalences used in not propagation.

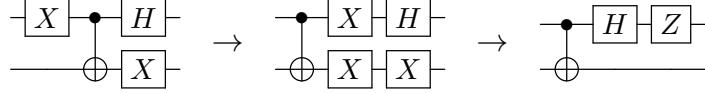


Figure 4.3: An example of not propagation. In the first step the leftmost X gate propagates through the $CNOT$ gate to become two X gates. In the second step the upper X gate propagates through the H gate and the lower X gates cancel.

4.3.1 Optimization by Propagation and Cancellation

Our *propagate-cancel* optimizations have two steps. First we localize a set of gates by repeatedly applying commutation rules. Then we apply a circuit equivalence to replace that set of gates. In VOQC, most optimizations of this form use a library of code patterns, but one—*not propagation*—is slightly different, so we discuss it first.

Not Propagation

The goal of not propagation is to remove cancelling X (“not”) gates. Two X gates cancel when they are adjacent or they are separated by a circuit that commutes with X . We find X gates separated by commuting circuits by repeatedly applying the propagation rules in Figure 4.2. An example application of the not propagation algorithm is shown in Figure 4.3. This implementation may introduce extra X gates at the end of a circuit or extra Z gates in the interior of the circuit. Extra Z gates are likely to be cancelled by the gate cancellation and rotation merging passes that follow, and moving X gates to the end of a circuit makes the rotation merging optimization more likely to succeed. We note that our version of this optimization is a simplification of Nam et al.’s, which is specialized to a three-qubit $CCNOT$ gate; this gate can be decomposed into a $\{H, R_z Q, CNOT\}$ program per Table 4.3. In our experiments, we did not observe any difference in performance between VOQC and Nam et al. due to this simplification.

Gate Cancellation

The single- and two-qubit gate cancellation optimizations rely on the same propagate-cancel pattern used in not propagation, except that gates are returned to their original location if they fail to cancel. To support this pattern, we provide a general `propagate` function in VOQC. This function takes as inputs (i) an instruction list, (ii) a gate to propagate, and (iii) a set of rules for commuting and cancelling that gate. At each

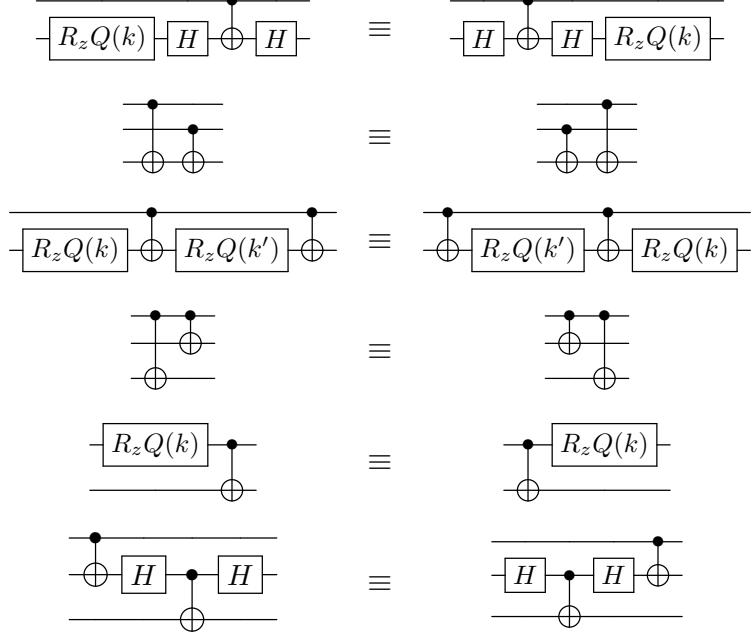


Figure 4.4: Commutation equivalences for single- and two-qubit gates adapted from Nam et al. [Nam+18, Figure 5]. We use the second and third rules for propagating both single- and two-qubit gates.

iteration, `propagate` performs the following actions:

1. Check if a cancellation rule applies. If so, apply that rule and return the modified list.
2. Check if a commutation rule applies. If so, commute the gate and recursively call `propagate` on the remainder of the list.
3. Otherwise, return the gate to its original position.

We have proved that our `propagate` function is sound when provided with valid commutation and cancellation rules. Each commutation or cancellation rule is implemented as a partial Coq function from an input circuit to an output circuit. A common pattern in these rules is to identify one gate (e.g., an X gate), and then to look for an adjacent gate it might commute with (e.g., $CNOT$) or cancel with (e.g., X). For commutation rules, we use the rewrite rules shown Figure 4.4. For cancellation rules, we use the fact that H , X , and $CNOT$ are all self-cancelling and $R_zQ(k)$ and $R_zQ(k')$ combine to become $R_zQ(k + k')$.

4.3.2 Circuit Replacement

We have implemented three optimizations that work by replacing one pattern of gates with an equivalent one; no preliminary propagation is necessary. These aim either to reduce the gate count directly, or to set the stage for additional optimizations.

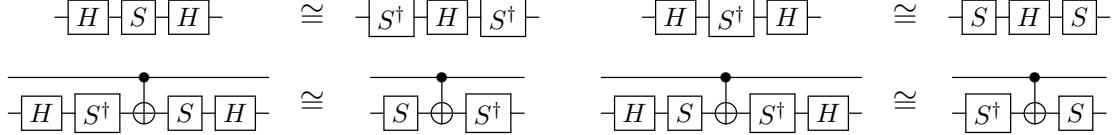


Figure 4.5: Equivalences for removing Hadamard gates adapted from Nam et al. [Nam+18, Figure 4]. S is the phase gate $R_zQ(\frac{1}{2})$ and S^\dagger is its inverse $R_zQ(\frac{3}{2})$.

Hadamard Reduction

The Hadamard reduction routine employs the equivalences shown in Figure 4.5 to reduce the number of H gates in the program. Removing H gates is useful because H gates limit the size of the $\{R_zQ, CNOT\}$ subcircuits used in the rotation merging optimization.

Rotation Merging

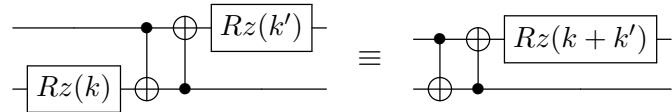
The rotation merging optimization allows for combining R_zQ gates that are not physically adjacent in the circuit. This optimization is more sophisticated than the previous optimizations because it does not rely on small structural patterns (e.g., that adjacent X gates cancel), but rather on more general (and non-local) circuit behavior. The basic idea behind rotation merging is to (i) identify subcircuits consisting of only $CNOT$ and R_zQ gates and (ii) merge R_zQ gates within those subcircuits that are applied to qubits in the same logical state. The argument for the correctness of this optimization relies on the *phase polynomial* representation of a circuit. Let C be a circuit consisting of $CNOT$ gates and rotations about the z -axis. Then on basis state $|x_1, \dots, x_n\rangle$ for $x_i \in \{0, 1\}$, C will produce the state

$$e^{ip(x_1, \dots, x_n)} |h(x_1, \dots, x_n)\rangle$$

where $h : \{0, 1\}^n \rightarrow \{0, 1\}^n$ is an affine reversible function and

$$p(x_1, \dots, x_n) = \sum_{i=1}^l (\theta_i \bmod 2\pi) \cdot f_i(x_1, \dots, x_n)$$

is a linear combination of affine boolean functions. $p(x_1, \dots, x_n)$ is called the phase polynomial of circuit C . Each rotation gate in the circuit is associated with one term of the sum and if two terms of the phase polynomial satisfy $f_i(x_1, \dots, x_n) = f_j(x_1, \dots, x_n)$ for some $i \neq j$, then the corresponding i and j rotations can be merged. As an example, consider the two circuits shown below.



To prove that these circuits are equivalent, we can consider their behavior on basis state $|x_1, x_2\rangle$. Applying $R_zQ(k)$ to the basis state $|x\rangle$ produces the state $e^{ik\pi x}|x\rangle$

$$\begin{aligned}
U_1(\lambda_1) ; U_1(\lambda_2) &= U_1(\lambda_1 + \lambda_2) \\
U_1(\lambda_1) ; U_2(\phi_2, \lambda_2) &= U_2(\phi_2, \lambda_1 + \lambda_2) \\
U_1(\lambda_1) ; U_3(\theta_2, \phi_2, \lambda_2) &= U_3(\theta_2, \phi_2, \lambda_1 + \lambda_2) \\
U_2(\phi_1, \lambda_1) ; U_1(\lambda_2) &= U_2(\phi_1 + \lambda_2, \lambda_1) \\
U_2(\phi_1, \lambda_1) ; U_2(\phi_2, \lambda_2) &= U_3(\pi - \phi_1 - \lambda_2, \phi_2 + \frac{\pi}{2}, \lambda_1 + \frac{\pi}{2}) \\
U_3(\theta_1, \phi_1, \lambda_1) ; U_1(\lambda_2) &= U_3(\theta_1, \phi_1 + \lambda_2, \lambda_1)
\end{aligned}$$

Figure 4.6: Rules for single-qubit gate merging.

and $CNOT |x, y\rangle$ produces the state $|x, x \oplus y\rangle$ where \oplus is the xor operation. Thus evaluation of the left-hand circuit proceeds as follows:

$$\begin{aligned}
|x_1, x_2\rangle &\rightarrow e^{ik\pi x_2} |x_1, x_2\rangle \\
&\rightarrow e^{ik\pi x_2} |x_1, x_1 \oplus x_2\rangle \\
&\rightarrow e^{ik\pi x_2} |x_2, x_1 \oplus x_2\rangle \\
&\rightarrow e^{ik\pi x_2} e^{ik'\pi x_2} |x_2, x_1 \oplus x_2\rangle.
\end{aligned}$$

Whereas evaluation of the right-hand circuit produces

$$|x_1, x_2\rangle \rightarrow |x_1, x_1 \oplus x_2\rangle \rightarrow |x_2, x_1 \oplus x_2\rangle \rightarrow e^{i(k+k')\pi x_2} |x_2, x_1 \oplus x_2\rangle.$$

The two resulting states are equal because $e^{ik\pi x_2} e^{ik'\pi x_2} = e^{i(k+k')\pi x_2}$. This implies that the unitary matrices corresponding to the two circuits are the same. We can therefore replace the circuit on the left with the one on the right, removing one gate from the circuit.

Our rotation merging optimization follows the reasoning above for arbitrary $\{R_z Q, CNOT\}$ circuits. For every gate in the program, it tracks the Boolean function associated with every qubit (the Boolean functions above are $x_1, x_2, x_1 \oplus x_2$), and merges $R_z Q$ rotations when they are applied to qubits associated with the same Boolean function. To prove equivalence over $\{R_z Q, CNOT\}$ circuits, we show that the original and optimized circuits produce the same output on every basis state. We have found evaluating behavior on basis states to be useful for proving equivalences that are not as direct as those listed in Figures 4.4 and 4.5. Although our merge operation is identical to Nam et al.'s, our approach to constructing $\{R_z Q, CNOT\}$ subcircuits differs. We construct a $\{R_z Q, CNOT\}$ subcircuit beginning from a $R_z Q$ gate whereas Nam et al. begin from a $CNOT$ gate. The result of this simplification is that we may miss some opportunities for merging. However, in our experiments we found that this choice impacted only one benchmark.

Single-qubit Gate Merging

In the IBM gate set, any two single-qubit gates can be combined into one gate. This allows us to implement an optimization over programs in the IBM gate set (which Qiskit calls `Optimize1qGates` [Qis21]) that merges all adjacent single-qubit gates by applying the rules in Figure 4.6, along with a more complex rule for combining a U_2 and U_3 gate or two U_3 gates. In the more complex rule, the two gates are first

converted into a sequence of Euler rotations about the y - and z -axes: $U_3(\theta, \phi, \lambda) \rightarrow R_z(\phi) \cdot R_y(\theta) \cdot R_z(\lambda)$. Call this a ZYZ rotation. Next, local identities are applied to combine the two ZYZ rotations into a single ZYZYZ rotation. Then the interior YZY rotation is converted to a new ZYZ rotation, yielding a ZZYZZ rotation. Finally, this is simplified to a ZYZ rotation, which can be represented as a U_3 gate. For example, here is the process for combining two U_3 gates:

$$\begin{aligned} U_3(\theta_1, \phi_1, \lambda_1) ; U_3(\theta_2, \phi_2, \lambda_2) &= R_z(\phi_2) \cdot R_y(\theta_2) \cdot R_z(\lambda_2) \cdot R_z(\phi_1) \cdot R_y(\theta_1) \cdot R_z(\lambda_1) \\ &= R_z(\phi_2) \cdot [R_y(\theta_2) \cdot R_z(\lambda_2 + \phi_1) \cdot R_y(\theta_1)] \cdot R_z(\lambda_1) \\ &= R_z(\phi_2) \cdot [R_z(\gamma) \cdot R_y(\beta) \cdot R_z(\alpha)] \cdot R_z(\lambda_1) \\ &= R_z(\phi_2 + \gamma) \cdot R_y(\beta) \cdot R_z(\alpha + \lambda_1) \\ &= U_3(\beta, \phi_2 + \gamma, \alpha + \lambda_1) \end{aligned}$$

where α, β, γ satisfy $R_y(\theta_2) \cdot R_z(\lambda_2 + \phi_1) \cdot R_y(\theta_1) = R_z(\gamma) \cdot R_y(\beta) \cdot R_z(\alpha)$. The angles α, β, γ can be generated using arithmetic over trigonometric functions sin, cos, arccos, and arctan [Ebe99], as shown in Figure 4.7. Proving the generation of α, β, γ correct was the most difficult part of verifying soundness for this optimization; to our knowledge, we are the first to formally verify this method in a proof assistant.

4.3.3 Unitary Optimization Scheduling

The VOQC `optimize` function applies each of the optimizations we have discussed one after the other, in the following order (due to Nam et al.):

0, 1, 3, 2, 3, 1, 2, 4, 3, 2

where 0 is not propagation, 1 is Hadamard reduction, 2 is single-qubit gate cancellation, 3 is two-qubit gate cancellation, and 4 is rotation merging. Nam et al. justify this ordering at length, though they do not prove that it is optimal. In brief, removing X and H gates (0,1) allows for more effective application of the gate cancellation (2,3) and rotation merging (4) optimizations. In our experiments (Section 4.5), we observed that single-qubit gate cancellation and rotation merging were the most effective at reducing gate count.

We conclude by converting the gates to the IBM gate set and performing single-qubit gate merging in order to produce output in the $\{U_1, U_2, U_3, CNOT\}$ gate set for fair comparison with other tools.

4.3.4 Optimizing Non-Unitary Programs

We have implemented and verified two optimizations for non-unitary programs in VOQC, inspired by optimizations in IBM’s Qiskit compiler [Ale+19]: removing pre-measurement z -rotations, and classical state propagation. For these optimizations, we represent a non-unitary program P as a list of *blocks*. A block is a binary tree whose leaves are unitary programs (in list form) and nodes are measurements `meas q P1 P2` whose children P_1 and P_2 are lists of blocks. Since the density matrix semantics denotes programs as functions over matrices, we say that programs P_1 and P_2 of dimension d are equivalent if for every input ρ , $\{\!P_1\!\}_d(\rho) = \{\!P_2\!\}_d(\rho)$.

```

Definition rm02 (x y z : R) : R := sin x * cos z + cos x * cos y * sin z.
Definition rm12 (x y z : R) : R := sin y * sin z.
Definition rm22 (x y z : R) : R := cos x * cos z - sin x * cos y * sin z.
Definition rm10 (x y z : R) : R := sin y * cos z.
Definition rm11 (x y z: R) : R := cos y.
Definition rm20_minus (x y z : R) : R := cos x * sin z + sin x * cos y * cos z.
Definition rm21 (x y z : R) : R := sin x * sin y.

Definition atan2 (y x : R) : R :=
  if 0 <? x then atan (y/x)
  else if x <? 0 then if negb (y <? 0) then atan (y/x) + PI else atan (y/x) - PI
    else if 0 <? y then PI/2 else if y <? 0 then -PI/2 else 0.

Definition yzy_to_zyz (x y z : R) : R * R * R :=
  if rm22 x y z <? 1
  then if -1 <? rm22 x y z
    then (atan2 (rm12 x y z) (rm02 x y z),
          acos (rm22 x y z),
          atan2 (rm21 x y z) (rm20_minus x y z))
    else (- atan2 (rm10 x y z) (rm11 x y z), PI, 0)
  else (atan2 (rm10 x y z) (rm11 x y z), 0, 0).

(* Correctness property: *)
Lemma yzy_to_zyz_correct : ∀ θ₁ ⋯ θ₂ ⋯ ξ₁ ⋯ ξ₂,
  yzy_to_zyz θ₁ ⋯ θ₂ = (ξ₁, θ, ξ₂) →
  y_rotation θ₂ × phase_shift ξ × y_rotation θ₁
  × phase_shift ξ₂ × y_rotation θ × phase_shift ξ₁.

```

Figure 4.7: Code for converting a YZY rotation to a ZYZ rotation

z-rotations Before Measurement

z-axis rotations (or, more generally, diagonal unitary operations) before a measurement will have no effect on the measurement outcome, so they can safely be removed from the program. This optimization locates R_z gates before measurement operations and removes them. It was inspired by Qiskit’s RemoveDiagonalGatesBeforeMeasure [Qis21] pass.

Classical State Propagation

Once a qubit has been measured, the subsequent branch taken provides information about the qubit’s (now classical) state, which may allow pre-computation of some values. For example, in the branch where qubit q has been measured to be in the $|0\rangle$ state, any $CNOT$ with q as the control will be a no-op and any subsequent measurements of q will still produce zero.

In detail, given a qubit q in classical state $|i\rangle$, our optimization applies the following

propagation rules:

- $Rz(k)$ q preserves the classical state of q .
- X q flips the classical state of q .
- If $i = 0$ then $CNOT q q'$ is removed.
- If $i = 1$ then $CNOT q q'$ becomes $X q'$.
- $\text{meas } q P_1 P_0$ becomes P_i .
- $H q$ and $CNOT q' q$ make q non-classical and terminate analysis.

Our statement of correctness for one round of propagation says that if qubit q is in a classical state in the input, then the optimized program will have the same denotation as the unoptimized original. We express the requirement that qubit q be in classical state $i \in \{0, 1\}$ with the condition $|i\rangle_q \langle i| \times \rho \times |i\rangle_q \langle i| = \rho$, which says that projecting state ρ onto the subspace where q is in state $|i\rangle$ results in no loss of information.

This optimization is not implemented directly in Qiskit, but Qiskit contains passes that have a similar effect. For example, `RemoveResetInZeroState` [Qis21] removes adjacent reset gates, as the second has no effect.

4.4 Circuit Mapping

Similar to how optimization aims to reduce qubit and gate usage to make programs more feasible to run on near-term machines, *circuit mapping* aims to address connectivity constraints of near-term machines [SWD11; ZPW17]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of the underlying architecture. Consider the connectivity constraints of IBM's 16-qubit Guadalupe machine [IBM22], shown in Figure 4.8. This is a representative example of a superconducting qubit system, where qubits are laid out in a 2-dimensional grid and possible interactions are described by edges between qubits. For instance, a $CNOT$ gate may be applied between qubits 1 and 4, but not between qubits 1 and 3.

Circuit mapping typically consists of two stages: *layout* (or placement), which associates each logical qubit in the program with some physical qubit on the machine; and *routing*, which, given an initial layout, transforms a program to satisfy connectivity constraints. Routing is often performed by inserting $SWAP$ gates to “move” qubits to compatible locations when they are used together in a two-qubit gate. This approach is used by the routing routines in Qiskit [Ale+19] and `tket` [Siv+20]. Another approach is to compute the unitary matrix corresponding to the input (sub)circuit and then *resynthesize* the circuit in a way that satisfies connectivity constraints; Staq [AG20] implements this approach.

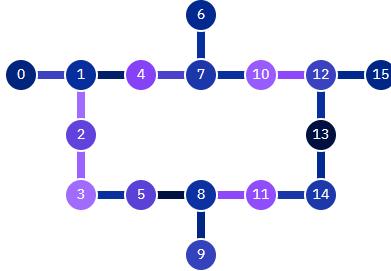


Figure 4.8: IBM’s Guadalupe machine [IBM22]. The different colors on nodes and connections reflect different error rates (darker means lower error and lighter means higher error).

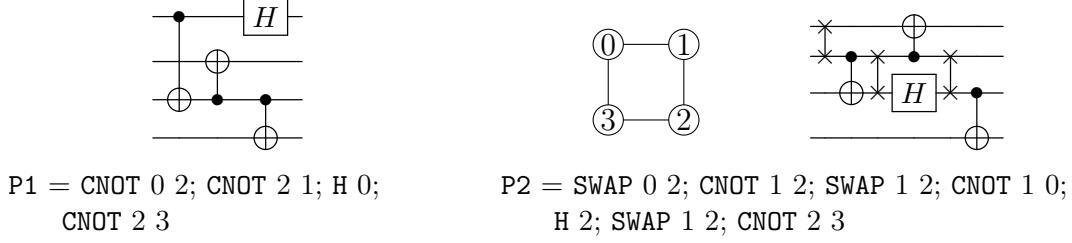
4.4.1 Verified Layout

We provide two layout functions in VOQC: *trivial layout*, which maps logical qubit i to physical qubit i , and *greedy layout* which takes into account the program and architecture characteristics, allocating logical qubits to nearby physical qubits when they are used together in a two-qubit gate. Figure 4.9(a) shows an example quantum circuit that uses four qubits. Imagine that we would like to map this circuit to an architecture with four qubits, connected in a ring. Figure 4.9(b) shows the result of arranging the circuit’s qubits according to a trivial layout, and Figure 4.9(c) shows the result of arranging the qubits according to our greedy layout routine. The greedy layout allocates logical qubits 0 and 2 (and 1 and 2) to adjacent physical qubits since they are used together in a *CNOT* gate.

We verify that both the trivial and greedy layout functions produce well-formed layouts, i.e., one-to-one mappings between logical and physical qubits. We also provide a function to convert a list to a layout; we prove that this function produces a well-formed layout if the input list has no duplicates and every element in the list is less than the length of the list. We use this function to generate safe layouts for our translation validation routine in Section 4.4.3.

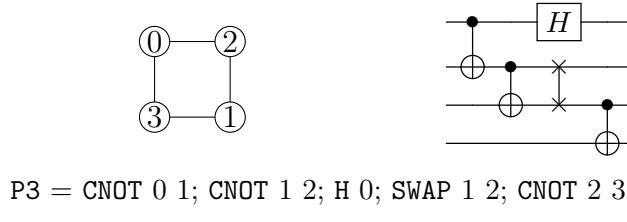
4.4.2 Verified Routing

We have implemented a simple *SWAP*-based routing method for unitary SQIR programs and verified that it is sound (up to a permutation of qubits) and produces programs that satisfy the relevant hardware constraints. Our routing method is parameterized by a description of the connectivity of an architecture, which includes a function to check whether an edge is in the connectivity graph and a function to find a path between two nodes. Given an initial layout, our implementation iterates through the gates of the input program and, every time a *CNOT* occurs between two logical qubits whose corresponding physical qubits are not adjacent in the underlying architecture, inserts *SWAPs* to move the control adjacent to the target. Figure 4.9(b–c) show the result of applying our routing routine to the circuit in Figure 4.9(a) starting from the trivial and greedy layouts. In Figure 4.9(b), three *SWAP* gates are inserted



(a) Input program

(b) Trivial layout



(c) Greedy layout

Figure 4.9: Circuit mapping example

to ensure that the circuit can execute on the target hardware, while in Figure 4.9(c) only one *SWAP* is inserted. In both produced circuits, the wires are ordered: top left physical qubit, top right, lower right, lower left.

Although our routing algorithm is simple (it is equivalent to Qiskit’s `BasicSwap` pass [Qis21]), it allows for some flexibility in design because we do not specify the method for finding paths in the connectivity graph, which allows, for example, strategies that take into account error characteristics of the machine [TQ19]. We have built-in connectivity graphs for the linear nearest neighbor (LNN), LNN ring, and 2D nearest neighbor architectures pictured in Figure 4.10.

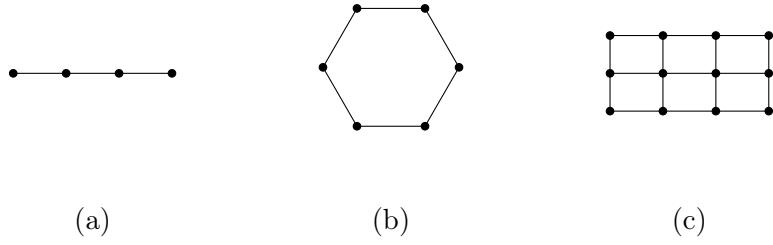


Figure 4.10: Architectures supported in voQC. From left to right: LNN, LNN ring, and 2D grid. Each architecture is shown with a fixed number of vertices, but in our implementation the number of vertices is a parameter.

4.4.3 Verified Translation Validation

There are a wide variety of proposed layout and routing techniques, many of which involve complex search algorithms and heuristic techniques [SWD11; ZPW17; TQ19; CSU19; LDX19]. Rather than aiming to verify all of these different approaches in Coq, we provide a *verified translation validation* function to check the correctness of circuit mapping on particular inputs. Unlike our verified optimizations and mapping routines, the verified translation validator may fail at runtime, indicating a potential bug in the circuit mapper. However, if translation validation succeeds, then our proofs guarantee that the input and output programs are mathematically equivalent up to permutation.

Our translation validator works by removing *SWAP* gates, performing a logical relabelling of qubits, and then checking for equality modulo reordering of gates that respects gate dependencies. This approach to equivalence checking will only work for routing routines that insert *SWAP* gates while leaving the rest of the program’s structure unchanged (e.g., those in Qiskit and $t|ket\rangle$)³. It will not be able to validate circuits generated using resynthesis (e.g., using Staq’s `steiner` routing). However, it is possible to develop polynomial-time translation validation functions for these cases too since the input and output circuits have a restricted form (e.g., Staq performs routing for $\{CNOT, X, R_z\}$ sub-circuits, which can be easily analyzed using phase polynomials per our discussion of rotation merging in Section 4.3.2).

Translation validation is popular for providing assurance for quantum compilers: Amy [Amy18] checks for equivalence of optimized and unoptimized programs using the path-sums semantics; PyZX [KW19] performs translation validation by checking if the result of optimizing a circuit followed by its optimized adjoint produces an identity program; and Smith and Thornton [ST19] provide a compiler with built-in translation validation via QMDD equivalence checking [MT06]. Most recently, Burgholzer, Raymond, and Wille [BRW20] presented a technique for equivalence checking, specialized to validating results of the Qiskit compiler, that relies on the fact that the identity matrix (which should be the result of composing a circuit with its optimized adjoint) can be efficiently represented using decision diagrams [BW20]; this observation allows them to perform equivalence checking on circuit that use tens of thousands of operations. But none of these other tools has been formally verified, and all aside from Burgholzer, Raymond, and Wille [BRW20] are more computationally heavy than our mapping validation, which simply requires a linear scan through the input, because they aim to detect equivalence between a more general class of programs.

³ $t|ket\rangle$ may choose to compile a distance-2 *CNOT* gate to a distributed *CNOT*, rather than a *SWAP* followed by a *CNOT*, if the heuristics find that this improves the final result [Siv+20, Section 7.2]. This optimization will cause our translation validator to fail; a simple fix would be to decompose *SWAP* gates before validation and then remove select *CNOT* gates from the mapped circuits.

4.4.4 Mapping with Optimization

Circuit mapping increases the size of the program, typically adding many *CNOT* gates to perform *SWAPs* between qubits. It is desirable to reduce this overhead by applying optimization after mapping, but this is only worthwhile if optimization preserves the guarantee from mapping that all *CNOT* gates are allowed by the connectivity graph. We have verified that all of the optimizations in Section 4.3 preserve connectivity guarantees, allowing us to apply optimization before and/or after mapping.

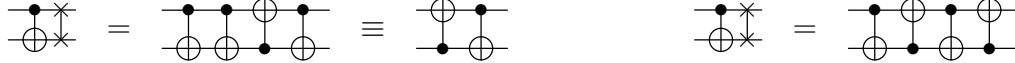
We also apply some light mapping-specific optimizations. For example, after mapping we carefully decompose *SWAP* gates to enable further optimization. *SWAP* gates have two natural decompositions in terms of *CNOT* gates:

$$\text{SWAP } a \ b = \text{CNOT } a \ b; \text{CNOT } b \ a; \text{CNOT } a \ b$$

and

$$\text{SWAP } a \ b = \text{CNOT } b \ a; \text{CNOT } a \ b; \text{CNOT } b \ a.$$

We choose the decomposition that will allow *CNOT* gates to be removed during gate cancellation (Section 4.3.1). For example, the subcircuit in Figure 4.9(b) and Figure 4.9(c) that consists of a *CNOT* followed by *SWAP* will be decomposed as shown on the left below, rather than the right, since this will save two gates in the optimized form.



4.5 Experimental Evaluation

[TODO: Copy from journal version, once finished. Should be 4–5 pages]

4.6 Related Work

Quantum compilation is an active area. In addition to Qiskit, $t|\text{ket}\rangle$, Staq, PyZX, and Nam et al. [Nam+18] (discussed in Section 4.5), other recent compiler efforts include quic [Rig19b], ScaffCC [Jav+15], and Project Q [SHT18]. Due to resource limits on near-term quantum machines, most compilers for quantum programs contain some degree of optimization, and nearly all place an emphasis on satisfying architectural requirements, like mapping to a particular gate set or qubit topology. None of the optimization or mapping code in these compilers is formally verified.

However, VOQC is not the only quantum compiler to which automated reasoning or formal verification has been applied. Amy, Roetteler, and Svore [ARS17] developed a certified optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs. Rand, Paykin, and Zdancewic [RPZ18] developed a similar compiler for quantum circuits but without optimizations (using the *QWIRE* language).

The problem of optimization verification has also been considered in the context of the ZX-calculus [CD11], which is a formalism for describing quantum tensor networks (which generalize quantum circuits) based on categorical quantum mechanics [AC09]. The ZX-calculus is characterized by a small set of rewrite rules that allow translation of a diagram to any other diagram representing the same computation [JPV18]. Fagan and Duncan [FD18] verified an optimizer for ZX diagrams representing Clifford circuits (which use the non-universal gate set $\{CNOT, H, S\}$) in the Quantomatic graphical proof assistant [KZ15]. PyZX [KW19] uses ZX diagrams as an intermediate representation for compiling quantum circuits, and generally achieves performance comparable to leading compilers [Kv19]. While PyZX is not verified in a proof assistant like Coq (the “Py” stands for Python), it does rely on a small, well-studied equational theory. Additionally, PyZX can perform translation validation to check if a compiled circuit is equivalent to the original. However, PyZX’s translation validator is not guaranteed to succeed for any two equivalent circuits.

Smith and Thornton [ST19] and Burgholzer, Raymond, and Wille [BRW20] also use translation validation to check whether a compiled program is semantically equivalent to its source. Smith and Thornton [ST19] rely on QMDD equivalence checking [MT06], which scales poorly with increasing number of qubits. Burgholzer, Raymond, and Wille [BRW20] use a more efficient approach to check equivalence of decision diagrams [BW20], but will still exhibit poor scaling depending on the optimizations applied. Both tools have been tested on optimizations simpler than those available in VOQC.

Concurrently with our work, Shi et al. [Shi+20] developed CertiQ, an approach to verifying properties of circuit transformations in the Qiskit compiler, which is implemented in Python. Their approach has two steps. First, it uses matrix multiplication to check that the unitary semantics of two concrete gate patterns are equivalent. Second, it uses symbolic execution to generate verification conditions for parts of Qiskit that manipulate circuits. These are given to an SMT solver to verify that pattern equivalences are applied correctly according to programmer-provided function specifications and invariants. That CertiQ can analyze Python code directly in a mostly automated fashion is appealing. However, it is limited in the optimizations it can verify. For example, equivalences that range over arbitrary indices, like $CNOT m x; CNOT n x \equiv CNOT n x; CNOT m x$ cannot be verified by matrix multiplication; CertiQ checks a concrete instance of this pattern and then applies it to more general circuits. More complex optimizations like rotation merging (the most powerful optimization in our experiments) cannot be generalized from simple, concrete circuits. CertiQ may also fail to prove an optimization correct, e.g., because of complicated control code; in this case it falls back to translation validation, which adds extra cost and the possibility of failure at run-time. By contrast, every optimization in VOQC has been proved correct. Finally, CertiQ does not directly represent the semantics of quantum programs, so unlike SQIR it cannot be used as a tool for verifying general properties of a program’s semantics.

Chapter 5

Applying Formal Verification to Q#

5.1 Overview

SQIR and VOQC comprise a key part of a high-assurance toolchain for quantum programming, supporting verification and optimization of quantum circuits. However, SQIR is a low-level language not suited for writing high-level quantum algorithms. We can get around this to some extent by meta-programming in Coq, but Coq was not designed to be a source programming tool (it emphasizes proof) and has no features to support quantum programming, like libraries for common quantum algorithm components or a simulator for quantum programs. Q# [Svo+18; Hei20] is a recent quantum programming language from Microsoft that includes extensive libraries for quantum computing and comes equipped with state-of-the-art simulators, but has no support for formal verification. In this chapter, we discuss our efforts to adapt our work on quantum formal verification to the Q# programming framework.

Figure 5.1 presents our vision for a verified software toolchain for Q#: users write their programs in Q#, taking advantage of the many features available in Microsoft’s Quantum Development Kit (QDK); the Q# program is then translated into Q^* , our novel high-level quantum programming language embedded in the F^* proof assistant [Dev22]; various properties are proved about the Q^* program in F^* , and then the original Q# program is compiled to Microsoft’s QIR [Gel20]; then the quantum/-classical QIR is compiled to (potentially multiple) SQIR circuits, which are optimized with VOQC; and finally the SQIR circuits are compiled to executable code. To provide the benefits of verified compilation, described in previous chapters, we envision that compilation from Q# to QIR to SQIR to executable code is verified to be semantics-preserving, guaranteeing that properties proved of the Q^* source also hold for code executed on the machine.

To achieve this ambitious goal requires a semantics for Q#, QIR, and “executable code”, as well as verified compilers between them and SQIR—a daunting task. As an initial step toward this goal, we present Q^* , a quantum programming language shallowly embedded in F^* that closely resembles Q#. We provide a plugin for the Q# compiler to translate Q# into Q^* , allowing us to ascribe a semantics to Q# programs and enforce properties not currently enforced by the Q# compiler—in effect, using

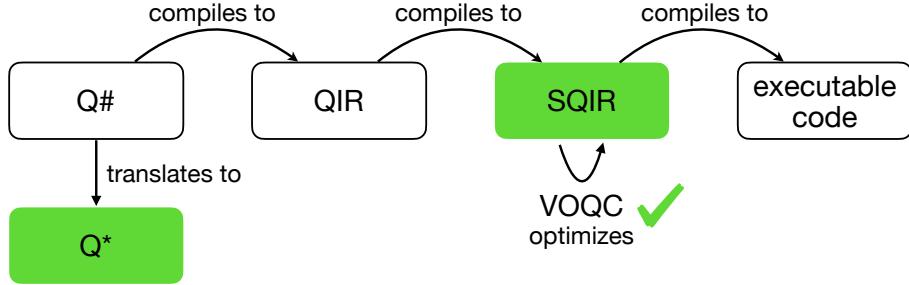


Figure 5.1: Overview of (proposed) toolchain for Q#. Green marks verified components.

F^* as an extra type checking pass.

Under the hood, Q^* is a simple language for constructing *quantum instruction trees*, a simple monadic representation of quantum programs based on SteelCore’s *indexed action trees* [Swa+20]. Pre and postconditions within the instruction trees guarantee well-formedness, ensuring properties like qubits being allocated before use, and distinct arguments in multi-qubit gates (which we call *correct qubit usage*). We further define a semantics for instruction trees in terms of a nondeterministic relation between vector states, and use this semantics to prove more fine-grained properties like *discard safety* (which guarantees that qubits are unentangled when they are discarded) and functional correctness. Q^* differs from prior work in the space in several respects:

- *Program Representation.* We represent quantum programs using a shallow embedding in F^* , with a focus on supporting the features available in Q#. Prior work has focused on custom research languages (e.g., SQIR and QWIRE in Coq and QBRICKS [Cha+21] in Why3) that do not resemble the high-level languages used in practice.
- *Semantics.* Rather than defining a matrix-based semantics for Q# directly, we define the semantics of a Q# program to be the interpretation of its instruction tree representation. Our interpretation function is defined in terms of vector states and is *non-deterministic* rather than *probabilistic* in its handling of measurement, which makes our semantics less expressive than prior semantics for quantum programs. However, this choice simplifies implementation and proof and is still powerful enough to verify the properties we consider, namely, correct qubit usage and discard safety.
- *Application.* We focus on automated proofs of properties that are useful to Q# developers, like correct qubit usage and discard safety, rather than manual proofs of full functional correctness.

This chapter provides background on the Q# language and F^* proof assistant (Section 5.2), presents our formalism for quantum instruction trees and the Q^* language (Section 5.3), and demonstrates applications of our formalism to verify useful properties of quantum programs (Section 5.4).

5.1.1 Acknowledgements

My work on Q^{*} began out of an internship with Microsoft Quantum in Summer 2021. I was mentored by Sarah Marshall and Nik Swamy and benefitted from conversations with Bettina Heim, Andres Paz, Chris Granade, and others. Content in this chapter has been influenced by ongoing work with Kartik Singhal, Sarah Marshall, and Robert Rand on designing $\lambda_{Q\#}$, a small language with a formal semantics describing the core of Q[#] [Sin+22].

5.2 Background

5.2.1 Q[#]: A High-Level Quantum Programming Language

Q[#] [Svo+18; Hei20] is a recent quantum programming language from Microsoft that encourages thinking about quantum programs as algorithms rather than circuits, allowing quantum operations like gates and measurement to be combined with standard classical control flow like branches and loops. An example Q[#] program, implementing the quantum teleportation protocol, is shown in Listing 5.1; we use this program as a motivating example throughout the chapter.

Q[#] callables are split into two types: *operations* can use effectful quantum features like qubit allocation and gate application, while *functions* consist of pure classical expressions with no quantum effect. Listing 5.1 defines four operations. An operation may have an adjoint or controlled *specialization*, which can be invoked with the combinators `Adjoint` or `Controlled`. Specializations can be automatically generated or manually provided; the *characteristics* `Adj` and `Ctl` indicate which specializations an operation supports. In Listing 5.1, `SendMsg` invokes the adjoint of `Entangle`, which is automatically generated. Q[#] follows the QRAM model of computation [Kni96], which assumes an unbounded supply of logical qubits from which the programmer can obtain a reference to a new qubit by calling `use`. Qubits are allocated and deallocated in a stack-like manner; the lifetime of a qubit is the lexical scope of its defining `use`. Upon allocation, a qubit is guaranteed to be in the $|0\rangle$ state and, upon deallocation, it is expected to either be in the $|0\rangle$ state or to have been measured prior. In Listing 5.1, `qAlice` is allocated in `Teleport`, measured in `SendMsg`, and then deallocated at the end of `Teleport`'s body.

Although Q[#]'s type system is able to distinguish between classical functions and quantum operations and tell whether an operation is adjointable or controllable, it does not enforce some basic requirements on how qubits are used, instead deferring to runtime checks in the simulator. Listings 5.2 to 5.4 show Q[#] programs that pass Q[#]'s type checker, but fail during simulation. In Listing 5.2, the same qubit is used as both inputs to a multi-qubit gate, violating the quantum no cloning theorem. In Listing 5.3, the `Qubit` value returned from `InitQubit` is actually discarded, and unavailable for reuse.

Listing 5.4 contains a more subtle bug. In `PrepareBell`, qubit `q2` is *entangled* with qubit `q1` when it is discarded, meaning that if the logical qubit allocated to `q2` were reallocated in another operation, say, to qubit `q3`, then updates to `q3` may impact the

```

namespace QStar.Telemport {
    open Microsoft.Quantum.Intrinsic;

    operation Entangle (qAlice : Qubit, qBob : Qubit) : Unit is Adj {
        H(qAlice);
        CNOT(qAlice, qBob);
    }

    operation SendMsg (qAlice : Qubit, qMsg : Qubit) : (Bool, Bool) {
        Adjoint Entangle(qMsg, qAlice);
        let m1 = M(qMsg);
        let m2 = M(qAlice);
        return (m1 == One, m2 == One);
    }

    operation DecodeMsg (qBob : Qubit, (b1 : Bool, b2 : Bool)) : Unit {
        if b1 { Z(qBob); }
        if b2 { X(qBob); }
    }

    operation Teleport (qMsg : Qubit, qBob : Qubit) : Unit {
        use qAlice = Qubit();
        Entangle(qAlice, qBob);
        let classicalBits = SendMsg(qAlice, qMsg);
        DecodeMsg(qBob, classicalBits);
    }
}

```

Listing 5.1: Teleportation in Q# (adapted from the Quantum Katas [Myk20])

```

use q1 = Qubit();
let q2 = q1;
CNOT(q1, q2);

```

Listing 5.2: Q# program violating no cloning.

seemingly unrelated q_1 . For example, if q_3 were measured, then q_1 would collapse to $|0\rangle$ or $|1\rangle$, depending on the measured value of q_3 . To avoid this issue, the Q# simulator checks at runtime that any discarded qubit is in a classical state. We can fix the bug in Listing 5.4 by inserting a measurement of q_2 before the end of `PrepareBell`, which will cause q_1 to collapse to $|0\rangle$ or $|1\rangle$ within the operation.

5.2.2 F*: A Proof Oriented Programming Language

[TODO: Talk about Low* [Pro+17] and Steel [Fro+21], differences between F* and Coq]

```

operation InitQubit () : Qubit {
    use q = Qubit();
    return q;
}

operation ApplyX() : Unit {
    let q = InitQubit ();
    X(q);
}

```

Listing 5.3: Q# program using a discarded qubit.

```

operation PrepareBell (q1 : Qubit) : Unit {
    use q2 = Qubit();
    H(q1);
    CNOT(q1, q2);
}

```

Listing 5.4: Q# program discarding an entangled qubit.

5.3 Quantum Instruction Trees

We describe Q# operations using *instruction trees*, which are a sequence of Action, Weaken, and Return nodes, with state passed in between, as shown in Listing 5.5. Instruction trees are marked with explicit preconditions, which must be true before program execution, and postconditions, which will be true after execution. Action nodes apply an instruction (defined below), Return nodes return a value, and Weaken nodes update a tree’s pre and postconditions. Our representation is inspired by SteelCore’s *indexed action trees* [Swa+20].

An example instruction tree, encoding the `Entangle` operation from Listing 5.1, is shown in Listing 5.6. We leave the pre and postcondition unspecified for brevity; we will discuss their form in Section 5.4.

Instructions are F[★] records that define a quantum operation’s semantics, useful pre and postconditions, and optional implementations of adjoint and controlled variants. Our instructions are roughly equivalent to Q# *intrinsics*, which are predefined primitive operations. The instruction type `inst` is shown in Listing 5.7. A precondition `pre` is some predicate over the input state, and a postcondition `post` relates the output value (of type `a`) to the input and output states. The semantic function `sem` transforms a state satisfying the precondition into one satisfying the postcondition. `adj` and `ctl` store optional implementations of adjoint and controlled variants, and `adj_pf` and `ctl_pf` store proofs that the provided variants match the expected specifications. In particular, we require that `sem` followed by `adj` returns a state to its original value, and that `ctl` is a no-op when the input qubit is in the $|0\rangle$ state and applies `sem` when it is in the $|1\rangle$ state.

```

let pre = state → prop
let post a = a → state → state → prop

type inst_tree (a:Type0) : pre → post a → Type =
| Return : q:post a →
    x:a →
        inst_tree a (fun s0 → q x s0 s0) q

| Action : #b:Type0 →
    #p:(b → pre) →
    #q:(b → post a) →
    i:inst b →
    k:(x:b → inst_tree a (p x) (q x)) →
        inst_tree a
            (fun s0 → i.pre s0 ∧ (∀ x s1. i.post x s0 s1 ==> p x s1))
            (fun x s0 s2 → (∃ y s'. i.post y s0 s' ∧ q y x s' s2))

| Weaken : #p:pre → #q:post a →
    p':pre → q':post a →
    _:squash (∀ s0. p' s0 ==> p s0) →
    _:squash (∀ x s0 s1. p' s0 ∧ q x s0 s1 ==> q' x s0 s1) →
    inst_tree a p q →
    inst_tree a p' q'

```

Listing 5.5: Instruction tree definition

```

let entangle (qAlice : qbit) (qBob : qbit)
: inst_tree unit PRE POST
= Action (had qAlice) (fun _ →
    Action (cnot qAlice qBob) (fun _ →
        Return POST ()))

```

Listing 5.6: Example instruction tree

```

type sem_ty (a:Type) (pre:pre) (post:post a) =
  s0:state{pre m0} →
  v:a & s1:state{post v s0 s1}

type inst (a:Type) = {
  // pre- and post-conditions
  pre: pre;
  post: post a;

  // semantics function
  sem: sem_ty a pre post;

  // optional implementation of adjoint
  adj: option (sem_ty a pre post);
  adj_pf: ...

  // optional implementation of control
  ctl: option (qbit → sem_ty a pre post);
  ctl_pf: ...
}

}

```

Listing 5.7: `inst` type

5.3.1 Semantics

Instruction trees act on a program state consisting of:

- Σ : A set of defined qubits.
- Ψ : A $2^{|\Sigma|}$ -length vector describing the state of the qubits in Σ .

Table 5.1 lists example instructions with their input and output types and semantics. `init` allocates a qubit and returns a reference to that qubit, `disc` deallocates a qubit, `meas` measures a qubit and returns the result, and `had` and `cnot` apply a quantum gate. Subscripts on matrix terms (e.g., q in $|b\rangle_q \langle q|$ and H_q) indicate that the matrix is only applied to a portion of the vector state, extending to the full state via padding.

The semantics of an instruction tree is given by an interpretation function that folds over the semantic function of each instruction, as shown in Listing 5.8. The type of the interpretation function ensures that if the input state satisfies the tree’s precondition, then the output state satisfies the postcondition.

5.3.2 Q^* Combinator Language

To ease translation from Q#, we provide a language shallowly embedded in F^* , called Q^* , that consists of combinators for building instruction trees. For example, we provide a “using” combinator that mimics Q#’s `use` command, which allocates a fresh qubit and discards that qubit at the end of its scope. We also provide combinators

Table 5.1: Example quantum instructions where $\text{disc}(q, b, \Psi) = \frac{\langle b|_q \times \Psi}{\|\langle b|_q \times \Psi\|}$ and $\text{meas}(q, b, \Psi) = \frac{\langle b|_q \langle b| \times \Psi}{\|\langle b|_q \langle b| \times \Psi\|}$, defining division to be zero in case of a zero denominator.

Name	Input	Output	Semantics
init	unit	qbit q	$(\Sigma, \Psi) \mapsto (\Sigma \cup q, 0\rangle_q \otimes \Psi)$
disc	qbit q	bool b	$(\Sigma, \Psi) \mapsto (\Sigma \setminus q, \text{disc}(q, b, \Psi))$
meas	qbit q	bool b	$(\Sigma, \Psi) \mapsto (\Sigma, \text{meas}(q, b, \Psi))$
had	qbit q	unit	$(\Sigma, \Psi) \mapsto (\Sigma, H_q \times \Psi)$
cnot	qbit q_1, q_2	unit	$(\Sigma, \Psi) \mapsto (\Sigma, CNOT_{q_1, q_2} \times \Psi)$

```

let rec eval_inst_tree #a #pre #post (it: inst_tree a pre post) (s0:state{pre s0})
  : Tot (v:a & s1:state{post a s0 s1}) (decreases it)
= match it with
  | Return _ v → (| v, s0 |)

  | Action i k →
    let (| v, s1 |) = i.sem s0 in
    let (| v', s2 |) = eval_inst_tree (k v) s1 in
    (| v', s2 |)

  | Weaken _ _ _ _ f →
    let (| v, s1 |) = eval_inst_tree f s0 in
    (| v, s1 |)
  
```

Listing 5.8: Instruction tree interpretation function. $s0$, $s1$, and $s2$ are states with the form (Σ, Ψ) and v and v' are return values.

for sequential composition, conditionals, and return statements, reducing the need for extraneous Weaken node. Finally, we provide combinators for computing the adjoint and controlled versions of an instruction tree, proving that their semantics match their mathematical definitions.

5.3.3 Translation from Q#

We built a plugin for the Q# compiler that generates a Q* program during compilation using the input Q# program's AST. By default, we choose an initial precondition that says that all input qubits are live (i.e., defined in Σ) and distinct and a post-condition that says that the resulting set of live qubits does not change. We chose these constraints because they are basic well-formedness conditions expected of Q# programs, but not currently enforced by the Q# compiler (see Section 5.4.1).

Listing 5.9 shows the Q* translation of Q# example from Listing 5.1. It uses combinators `cond`, `bind`, `adjoint`, and `using`. We reuse features of F*'s language whenever possible to avoid having to redefine basic features common to Q# and F*. For example, we translate Q#'s Boolean type into F*'s.

```

let decodeMsg (qBob : qbit) (b1 :  $\mathbb{B}$ ) (b2 :  $\mathbb{B}$ )
: inst_tree unit (fun s0 → live s0 qBob) eqpost
= bind (cond b1
         (Action (pauli_x qBob) (fun _ → Return eqpost ()))
         (Return eqpost ())) (fun _ →
cond b2
  (Action (pauli_z qBob) (fun _ → Return eqpost ()))
  (Return eqpost ()))

let entangle (qAlice : qbit) (qBob : qbit)
: inst_tree unit (fun s0 → qAlice ≠ qBob ∧ live s0 qAlice ∧ live s0 qBob)
eqpost
= Action (had qAlice) (fun _ →
  Action (cnot qAlice qBob) (fun _ →
    Return eqpost ()))

let sendMsg (qAlice : qbit) (qMsg : qbit)
: inst_tree ( $\mathbb{B}$  &  $\mathbb{B}$ ) (fun s0 → qAlice ≠ qMsg ∧ live s0 qAlice ∧ live s0 qMsg)
eqpost
= bind (adjoint (entangle qMsg qAlice)) (fun _ →
  Action (meas qMsg) (fun m1 →
    Action (meas qAlice) (fun m2 →
      Return eqpost (m1 = One, m2 = One)))))

let teleport (qMsg : qbit) (qBob : qbit)
: inst_tree unit (fun s0 → qMsg ≠ qBob ∧ live s0 qMsg ∧ live s0 qBob) eqpost
= using (fun s0 → qMsg ≠ qBob ∧ live s0 qMsg ∧ live s0 qBob) eqpost (fun
  qAlice →
    bind (entangle qAlice qBob) (fun _ →
      bind (sendMsg qAlice qMsg) (fun classicalBits →
        bind (decodeMsg qBob classicalBits) (fun _ →
          Return eqpost ())))))

```

Listing 5.9: Q^* translation of `teleport` from Listing 5.1. Weaken nodes omitted for brevity. `eqpost` is the postcondition that says that the set of live qubits does not change.

Table 5.2: Pre and postconditions for enforcing correct qubit usage, using the input and output names from Table 5.1. Preconditions may refer to the input state (Σ_0, Ψ_0) and postconditions may refer to both the input state and output state (Σ_1, Ψ_1) .

Name	Pre	Post
init	\top	$\Sigma_1 = \Sigma_0 \cup q$
disc	$q \in \Sigma_0$	$\Sigma_1 = \Sigma_0 \setminus q$
meas	$q \in \Sigma_0$	$\Sigma_1 = \Sigma_0$
had	$q \in \Sigma_0$	$\Sigma_1 = \Sigma_0$
cnot	$q_1 \in \Sigma_0 \wedge q_2 \in \Sigma_0 \wedge q_1 \neq q_2$	$\Sigma_1 = \Sigma_0$

5.4 Correctness Properties

Once we have converted a Q# program into Q^* , we can prove properties about the program in F^* . Proofs may be completely automated, relying only on the pre and postconditions in instructions, or fully manual; we discuss examples of both below.

5.4.1 Correct Qubit Usage

There are several basic requirements on how qubits can be used in order for the semantics in Table 5.1 to make sense. For example:

- Multi-qubit gates must be applied to distinct qubits,
- Every qubit must be live before use,
- Discarded qubits cannot be reused.

We call this family of requirements “correct qubit usage.” Other quantum languages like Silq [Bic+20a] and *QWIRE* [PRZ17] enforce correct qubit usage using linear type systems, which treat qubits as a resource. This is an attractive solution, but to our knowledge there is currently no plan to add a linear type system to Q#.

To check correct qubit usage, we use pre and postconditions shown in Table 5.2, which reflect the effect of the semantics in Table 5.1 on the set of defined qubits Σ . When translating from Q# to Q^* , we choose an initial precondition that says that all input qubits are live and distinct and a postcondition that says that the resulting set of live qubits does not change. For example, the precondition of the `entangle` function in Listing 5.9 says that `qAlice` and `qBob` are distinct and defined in the input state, which ensures that the `had` and `cnot` instructions are quantum-mechanically valid. The postcondition reflects the fact that Q# programs implicitly discard locally-allocated qubits at the end of their scope. In sum, converting a Q# program to an instruction tree and checking correct qubit usage has the same effect as running the program through a type system that treats qubits linearly.

Table 5.3: Pre and postconditions for enforcing discard safety, using the input and output names from Table 5.1.

Name	Pre	Post
init	emp	$q \mapsto 0\rangle$
disc	$q \mapsto \psi\rangle$	emp
meas	$q, \bar{q} \mapsto \psi\rangle$	$q \mapsto b\rangle \star \bar{q} \mapsto \text{disc}(q, b, \psi\rangle)$
had	$q, \bar{q} \mapsto \psi\rangle$	$q, \bar{q} \mapsto H_q \psi\rangle$
cnot	$q_1, q_2, \bar{q} \mapsto \psi\rangle$	$q_1, q_2, \bar{q} \mapsto CNOT_{q_1, q_2} \psi\rangle$

5.4.2 Discard Safety

It is “safe” to discard a qubit when it is not entangled with any other program qubits. Discarding an entangled qubit results in an implicit measurement of that qubit, which may change the rest of the program state in unintended ways (see Section 5.2.1). To avoid this, the Q# simulator enforces that discarded qubits are unentangled with the rest of the computation (and, additionally, that they are in a classical state).

To enforce discard safety, we use pre and postconditions shown in Table 5.3, which are inspired by recent quantum separation logics [Le+22; Zho+21]. In Table 5.3, \bar{q} refers to a (potentially empty) sequence of distinct qubits and $\bar{q} \mapsto |\psi\rangle$ says that in state (Σ, Ψ) , qubits \bar{q} are live in Σ and the portion of Ψ that corresponds to qubits \bar{q} is in state $|\psi\rangle$. In the preconditions, the predicate is over the input state (in the notation of Table 5.2, (Σ_0, Ψ_0)) and in the postconditions, the predicate is over the output state (Σ_1, Ψ_1) . In the case where \bar{q} is empty, we write **emp**. We use the separating conjunction \star from separation logic [Rey02]; $P_1 \star P_2$ says that we can partition the set of defined qubits Σ and corresponding quantum state Ψ so that P_1 holds of one partition and P_2 holds of the other. \star is commutative and $P \star \text{emp} = P$.

The predicate $\bar{q} \mapsto |\psi\rangle$ implies that qubits in \bar{q} are not entangled with qubits in the rest of the program. So the separating conjunction \star actually describes *separability of quantum states*, as observed by Le et al. [Le+22] and Zhou et al. [Zho+21]. The rules in Table 5.3 say that initialization produces a fresh unentangled qubit, discard requires its input to be unentangled, and measurement unentangles its input from the rest of the system. Gate applications like **had** and **cnot** simply multiply the vector state by the appropriate matrix, as in the original semantics (Table 5.1).

To analyze a program for discard safety, we can use the same initial precondition used to check correct qubit usage (i.e., all input qubits are live and distinct), extended with the condition that $\bar{q} \mapsto |\psi\rangle$ where \bar{q} consists of all input qubits and $|\psi\rangle$ is some appropriately-sized vector state. The rules in Table 5.3 will then enforce that every qubit is unentangled before being discarded. For example, they guarantee that **qAlice** is safely discarded in the teleport example since the last operation applied to **qAlice** is a measurement.

Frame Rule

When reasoning with the separating conjunction \star , a key inference rule is the *frame rule*, which has the form

$$\frac{\{P\} c \{Q\}}{\{P \star R\} c \{Q \star R\}}$$

where no variable occurring free in R is modified by c . This rule says that if we have a proof that program c takes predicate P to Q , we can automatically derive that it takes $P \star R$ to $Q \star R$, assuming that R is “unrelated” to c . This allows us to extend a local specification (like P, Q) to a global one ($P \star R, Q \star R$).

As an example, say that we have a program `applyH` that applies a Hadamard gate to qubit q and we know that q is initially in state $|\psi\rangle$ (i.e., $q \mapsto |\psi\rangle$). After `applyH` executes, we will have that $q \mapsto H \times |\psi\rangle$. We can extend this specification to a larger program that also includes qubits q_1 and q_2 : Say that initially $q_1, q_2 \mapsto |\phi\rangle$, then after `applyH` executes, we can conclude that $q_1, q_2 \mapsto |\phi\rangle \star q \mapsto H \times |\psi\rangle$. In other words, q_1 and q_2 are unaffected by `applyH`.

Entailment Reasoning

Unfortunately, applying the frame rule and the rules in Table 5.3 alone will not be enough to identify all safe discards because the rules do not account for *uncomputation*. It is common practice in quantum computing to use ancilla qubits to store temporary values (e.g., the carry bit in an addition circuit). These ancilla qubits may be entangled with the rest of the state to perform some operation, but they will be uncomputed (and not measured) before the ancilla are discarded.

[TODO: Replace with example from Q# libraries/Quantum Katas.]
For example, consider the following program, which computes the logical AND of qubits q_1, q_2, q_3 , storing the result in qubit out :

```
operation ApplyAnd3 (q1 : Qubit, q2 : Qubit, q3 : Qubit, out : Qubit) : Unit {
    use anc = Qubit();
    within {
        CCNOT(q1, q2, anc);
    } apply {
        CCNOT(q3, anc, out);
    }
}
```

Q#’s `within-apply` construct applies the second operation, conjugated by the first. So the body above unfolds to `CCNOT(q1, q2, anc); CCNOT(q3, anc, out); Adjoint CCNOT(q1, q2, anc);`. The ancilla qubit `anc` is uncomputed by the final `CCNOT`, leaving it unentangled with the rest of the state and safe to discard.

In order to reason about uncomputation, we will need to (manually) manipulate the matrix expressions inside predicates and selectively apply the entailment rule

$$\overline{q}_1, \overline{q}_2 \mapsto |\psi_1\rangle_{\overline{q}_1} \otimes |\psi_2\rangle_{\overline{q}_2} \iff (\overline{q}_1 \mapsto |\psi_1\rangle) \star (\overline{q}_2 \mapsto |\psi_2\rangle).$$

Applying this rule requires reasoning about the state $|\psi\rangle$ to show that it has the form

$|\psi_1\rangle_{\overline{q_1}} \otimes |\psi_2\rangle_{\overline{q_2}}$ for some $\overline{q_1}$ and $\overline{q_2}$. For example, say that `q1`, `q2`, `q3`, and `out` are initially in some vector state $|\psi\rangle$. This means that after the allocation of `anc`,

$$q_1, q_2, q_3, o, a \mapsto |\psi\rangle \otimes |0\rangle$$

where we use `o` for `out` and `a` for `anc`. After the `within-apply` construct, we have that

$$q_1, q_2, q_3, o, a \mapsto CCNOT_{q_1, q_2, a} \times CCNOT_{q_3, a, o} \times CCNOT_{q_1, q_2, a} \times (|\psi\rangle \otimes |0\rangle).$$

We can show that the $CCNOT$ matrix-vector produce can be rewritten as $|\psi'\rangle \otimes |0\rangle$ for some $|\psi'\rangle$, allowing us to derive that $q_1, q_2, q_3, o \mapsto |\psi'\rangle \star a \mapsto |0\rangle$.

Related Work

ScaffCC defines a simple entanglement analysis for circuits consisting of n -controlled X gates (X , $CNOT$, $CCNOT$, ...) [Jav+15, Sec 7.1]. The analysis involves tracking sets of potentially entangled qubits and updating the set when uncomputation occurs. Uncomputation happens when a gate occurs for a second time, assuming that the control(s) of the gate have not been the target of any operation since the last occurrence; the `ApplyAnd3` function above satisfies this criterion. Silq's type system [Bic+20a] similarly enforces that qubits are uncomputable (i.e., unentangled) before they go out of scope, restricting to the same (non-universal) gate set as ScaffCC. Twist's type system [YMC22] tracks purity in a program using a combination of static and dynamic checks. The static checks enforce a property similar to the one enforced by the rules in Table 5.3; the dynamic checks take the place of our manual entailment proofs. Q# uses dynamic simulator-based checks to enforce discard safety.

[TODO: Relationship with *ReQWIRE* [Ran+19]]

5.4.3 Custom Properties

Once we have a Q# program represented in F^* , we can perform arbitrary reasoning about the program's behavior. For example, we might prove *functional correctness*, e.g., that an operation prepares a certain state. We presented several examples of functional correctness in Chapter 3. We also might prove that manual implementations of control or adjoint, Q#-specific features, match their specifications.

Due to our choice of semantics, which ignores the probability of different measurement outcomes, there are some properties of interest that we will not be able to verify. For example, we cannot prove that two arbitrary F^* programs are equivalent, only that they produce the same set of outcomes; we cannot prove that a program returns a result with a particular probability, only that it may return the result. This nondeterministic view is sufficient for proving properties like discard safety and correct qubit usage, as well as other correctness properties that do not depend on measurement outcome probabilities. For example, the quantum teleportation protocol teleports a qubit regardless of intermediate measurement outcomes.

We support two approaches to proving custom correctness properties. The first option is to use the pre and postconditions from Section 5.4.1 to ensure basic program

```

let lemma_entangle_correct (qA qB:qbit)
    (s0:state{live s0 qA ∧ live s0 qB ∧ qA ≠ qB})
: Lemma (requires (in_1q_classical_state s0 qA false s0 ∧
                    in_1q_classical_state s0 qB false s0))
(ensures (let (| v, s1 |) = eval_inst_tree (entangle qA qB) s0 in
            in_2q_state qA qB bell00 s1))
= ...

```

Listing 5.10: The statement of functional correctness for `entangle` says that if qubits qA and qB are both initially in the $|0\rangle$ state, then after execution of `entangle`, they will be in the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

well-formedness, and then prove custom lemmas in F^* about the program's semantics (`eval_inst_tree` from Listing 5.8). This is the same as what we did in Chapter 3, where we manually stated and proved properties about the result of running `uc_eval` on a SQIR program. We show an example of this style of reasoning for the `entangle` function in Listing 5.10.

Alternatively, we can use the rules from Section 5.4.2 to reason about correctness, following the example of existing quantum separation logics [Le+22; Zho+21]. Figure 5.2 shows how we can use the pre and postconditions for discard safety to verify correctness of quantum teleportation. In our proof sketch, we show all intermediate reasoning steps, aside from application of the frame rule. We make use of the properties of \star and \mapsto discussed in Section 5.4.2.

```

{  $q_B \mapsto |0\rangle \star q_M \mapsto |\psi\rangle$  }
use qAlice = Qubit();
{  $q_B \mapsto |0\rangle \star q_M \mapsto |\psi\rangle \star q_A \mapsto |0\rangle$  }
{  $q_A, q_B \mapsto |00\rangle \star q_M \mapsto |\psi\rangle$  }
Entangle(qAlice, qBob);
{  $q_A, q_B \mapsto (CNOT_{q_A, q_B} \times H_{q_A} \times |00\rangle) \star q_M \mapsto |\psi\rangle$  }
{  $q_A, q_B \mapsto \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \star q_M \mapsto |\psi\rangle$  }
{  $q_M, q_A, q_B \mapsto |\psi\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  }

let classicalBits = SendMsg(qAlice, qMsg);
{  $q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star$ 
 $q_B \mapsto \text{disc}(q_A, b_2, \text{disc}(q_M, b_1, H_{q_M} \times CNOT_{q_M, q_A} \times (|\psi\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))))$  }
{  $q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star q_B \mapsto Z^{b_1}X^{b_2}|\psi\rangle$  }

DecodeMsg(qBob, classicalBits);
{  $q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star$ 
 $(b_1 \wedge b_2 \implies q_B \mapsto X \times Z \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$ 
 $(b_1 \wedge \neg b_2 \implies q_B \mapsto Z \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$ 
 $(\neg b_1 \wedge b_2 \implies q_B \mapsto X \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$ 
 $(\neg b_1 \wedge \neg b_2 \implies q_B \mapsto Z^{b_1}X^{b_2}|\psi\rangle)$  }
{  $q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star q_B \mapsto |\psi\rangle$  }

// deallocate qAlice (done implicitly)
{  $q_M \mapsto |b_1\rangle \star q_B \mapsto |\psi\rangle$  }

```

Figure 5.2: Body of the `teleport` function from Listing 5.1 annotated with pre and postconditions for proving functional correctness. q_A , q_B , and q_M are shorthand for `qAlice`, `qBob`, and `qMsg`, and b_1 and b_2 are shorthand for `Fst(classicalBits)` and `Snd(classicalBits)`. Expression M^b is equal to M if b is true and I (the identity matrix) if b is false.

Chapter 6

Conclusion

This dissertation has presented SQIR, a small quantum intermediate representation that can also be used to implement and verify quantum source programs; VOQC, a verified optimizer for quantum circuits that has performance on par with unverified tools; and Q^{*}, an initial attempt to support formal verification in the high-level language Q#. But there is still plenty of work to be done. We highlight some interesting directions below.

Verifying Near-term Algorithms So far, work on formally verified quantum computation has been limited to textbook quantum algorithms like QPE and Grover's. Although these algorithms are a useful stress-test for tools, they do not accurately reflect the types of quantum programs that are expected to run on near-term machines. Near-term algorithms are usually *approximate*. They do not implement the desired operation exactly, but rather perform an operation “close” to what was intended. For example, the version of QFT considered in Section 3.4 is the textbook presentation. In practice, it is more popular to consider an approximate QFT that removes gates that perform rotations by small angles [NC10].

Another issue is that many near-term algorithms run in a loop where: (i) the quantum program executes a circuit, (ii) the classical program observes the output of execution and performs post-processing (e.g. a classical non-linear optimizer), and (iii) the classical computer generates a continuation circuit for the quantum computer to run. As an example, the variational quantum eigensolver (VQE) uses this approach to approximate the smallest eigenvalue of a Hamiltonian [Per+14]. Verifying these types of programs requires reasoning properties about (simple) quantum programs, (non-trivial) classical programs, and their interplay. It also requires considering issue like convergence.

Additionally, near-term algorithms often need to account for hardware errors. Thus, verifying these algorithms may require considering their behavior in the presence of errors. So far, most of our work in SQIR has revolved around the unitary semantics and vector-based state abstractions because we find these simpler to work with. However, it is more natural to describe states subject to error using density matrices, since noisy states are mixtures of pure states [NC10, Chapter 8].

Higher-Level Abstractions for Verification On another front, there is important work to be done on describing quantum algorithms and correctness properties at a higher level of abstraction. The proofs and definitions in this paper follow the standard textbook presentation, which is in terms of circuits and are thus lower-level than similar proofs about classical programs. Rather than working from the circuit model, used in verification tools like *QWIRE*, *SQIR*, *QBRICKS*, and (to some extent) *QWhile*, it would be interesting to verify programs written in higher-level languages like Silq [Bic+20b] or Q# [Svo+18], as we aim to do in our work on Q^* .

Proving Complexity Bounds On-paper proofs of correctness of quantum algorithms argue not only that the algorithm manipulates the state in the desired way, but also that it does so using a number of gates (say) polynomial in the input size. In this age of exploration where we are searching for problems where quantum computers can outperform classical computers, arguments about complexity are almost as important as standard correctness properties. *QBRICKS* includes a size predicate, and such a predicate can easily be added to *SQIR*, but so far this predicate has only been used to count the number of gates in a (parameterized) circuit—it has not been used to argue about optimality or lower/upper bounds.

Extending the Verified Software Toolchain Along with verifying quantum programs, it is equally important to verify the infrastructure used to reason about those programs and turn them into executable code (along the lines of *VST* for classical software [App11]). Our work on *SQIR* and *VOQC* are a key part of this toolchain, but there is still much to be done. For example, a proper compiler needs a high-level source language, perhaps along the lines of recent languages like *Q#* [Svo+18] or *Silq* [Bic+20b]. Conversely, there is currently no support for verifying programs below the gate level. The lowest level in the quantum compiler stack is analog *pulse* instructions for the classical control hardware [Ale+20]. It may also be fruitful to verify other components of the quantum software toolchain, such as resource estimators and simulators.

Teaching Quantum Computing An early version of *SQIR* is the basis for Verified Quantum Computing (*VQC*) [Ran19], an online textbook in the style of Software Foundations [Pie+18] introducing readers to quantum computing through the *Coq* proof assistant. *VQC* has been successfully taught in a formal verification course at the University of Maryland and a tutorial at the Principles of Programming Languages conference, but it remains work-in-progress. The techniques and algorithms in this paper (particularly in Chapter 3) should allow us to cover the material in a standard quantum computing textbook while providing instant feedback to students. This will further strengthen the connection between quantum computing and formal proof, which we expect to prove valuable to programmers in the near future.

Bibliography

- [AC09] Samson Abramsky and Bob Coecke. “Categorical quantum mechanics”. In: *Handbook of quantum logic and quantum structures* 2 (2009), pp. 261–325.
- [Ale+19] Gadi Aleksandrowicz et al. *Qiskit: An Open-Source Framework for Quantum Computing*. Version 0.7.2. Zenodo, Jan. 2019. DOI: [10.5281/zenodo.2562111](https://doi.org/10.5281/zenodo.2562111).
- [Ale+20] Thomas Alexander et al. “Qiskit Pulse: Programming Quantum Computers Through the Cloud with Pulses”. In: *arXiv e-prints* (Apr. 2020). arXiv: [2004.06755 \[quant-ph\]](https://arxiv.org/abs/2004.06755).
- [Amy19a] Matt Amy. “Formal Methods in Quantum Circuit Design”. PhD thesis. University of Waterloo, 2019.
- [Amy18] Matthew Amy. *Towards large-scale functional verification of universal quantum circuits*. https://www.mathstat.dal.ca/qpl2018/papers/QPL_2018_paper_30.pdf. Presented at QPL 2018. 2018.
- [Amy19b] Matthew Amy. “Towards Large-Scale Functional Verification of Universal Quantum Circuits”. In: *Proceedings of the 15th International Conference on Quantum Physics and Logic, Halifax, Canada, 3-7th June 2018*. Ed. by Peter Selinger and Giulio Chiribella. Vol. 287. Electronic Proceedings in Theoretical Computer Science. Comment: In Proceedings QPL 2018, arXiv:1901.09476. Open Publishing Association, 2019, pp. 1–21. DOI: [10.4204/EPTCS.287.1](https://doi.org/10.4204/EPTCS.287.1).
- [AAM18] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. “On the controlled-NOT complexity of controlled-NOT-phase circuits”. In: *Quantum Science and Technology* 4.1 (2018).
- [AG20] Matthew Amy and Vlad Gheorghiu. “staq—A full-stack quantum processing toolkit”. In: *Quantum Science and Technology* 5.3 (June 2020), p. 034016. DOI: [10.1088/2058-9565/ab9359](https://doi.org/10.1088/2058-9565/ab9359). URL: <https://doi.org/10.1088/2058-9565/ab9359>.
- [AMM13] Matthew Amy, Dmitri Maslov, and Michele Mosca. “Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33 (Mar. 2013). DOI: [10.1109/TCAD.2014.2341953](https://doi.org/10.1109/TCAD.2014.2341953).

- [ARS17] Matthew Amy, Martin Roetteler, and Krysta M. Svore. “Verified compilation of space-efficient reversible circuits”. In: *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer, July 2017. URL: <https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/>.
- [App11] Andrew W. Appel. “Verified Software Toolchain”. In: ESOP’11/ETAPS’11. 2011.
- [Bar+95] Adriano Barenco et al. “Elementary gates for quantum computation”. In: *Phys. Rev. A* 52 (5 Nov. 1995), pp. 3457–3467. DOI: [10.1103/PhysRevA.52.3457](https://doi.org/10.1103/PhysRevA.52.3457). URL: <https://link.aps.org/doi/10.1103/PhysRevA.52.3457>.
- [Bic+20a] Benjamin Bichsel et al. “Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 286–300. ISBN: 978-1-4503-7613-6. DOI: [10.1145/3385412.3386007](https://doi.org/10.1145/3385412.3386007).
- [Bic+20b] Benjamin Bichsel et al. “Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 286–300. ISBN: 9781450376136. DOI: [10.1145/3385412.3386007](https://doi.org/10.1145/3385412.3386007). URL: <https://doi.org/10.1145/3385412.3386007>.
- [BKN15] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. “Formalization of Quantum Protocols using Coq”. In: *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015*. Ed. by Chris Heunen, Peter Selinger, and Jamie Vicary. Vol. 195. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2015, pp. 71–83. DOI: [10.4204/EPTCS.195.6](https://doi.org/10.4204/EPTCS.195.6).
- [BLH20] Anthony Bordg, Hanna Lachnitt, and Yijun He. “Certified Quantum Computation in Isabelle/HOL”. In: *Journal of Automated Reasoning* (2020). DOI: [10.1007/s10817-020-09584-7](https://doi.org/10.1007/s10817-020-09584-7). URL: <https://doi.org/10.1007/s10817-020-09584-7>.
- [BRW20] Lukas Burgholzer, Rudy Raymond, and Robert Wille. “Verifying results of the IBM Qiskit quantum circuit compilation flow”. In: *International Conference on Quantum Computing and Engineering*. Oct. 2020.
- [BW20] Lukas Burgholzer and Robert Wille. “Advanced Equivalence Checking for Quantum Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.9 (Oct. 2020). DOI: [10.1109/TCAD.2020.3032630](https://doi.org/10.1109/TCAD.2020.3032630).

- [Cam19] Cambridge Quantum Computing Ltd. *pytket*. 2019. URL: <https://cqcl.github.io/pytket/build/html/index.html>.
- [Cas21] Davide Castelvecchi. “Mathematicians welcome computer-assisted proof in ‘grand unification’ theory”. In: *Nature* 595.7865 (July 2021), pp. 18–19. DOI: [10.1038/d41586-021-01627-](https://doi.org/10.1038/d41586-021-01627-).
- [Cha+20] Christophe Chareton et al. “Toward Certified Quantum Programming”. In: *arXiv e-prints* (2020). arXiv: [2003.05841 \[cs.PL\]](https://arxiv.org/abs/2003.05841).
- [Cha+21] Christophe Chareton et al. “An Automated Deductive Verification Framework for Circuit-Building Quantum Programs”. In: *Programming Languages and Systems*. Ed. by Nobuko Yoshida. Cham: Springer International Publishing, 2021, pp. 148–177. ISBN: 978-3-030-72019-3.
- [CSU19] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. “Circuit Transformations for Quantum Architectures”. In: *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*. Ed. by Wim van Dam and Laura Mancinska. Vol. 135. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 3:1–3:24. ISBN: 978-3-95977-112-2. DOI: [10.4230/LIPIcs.TQC.2019.3](https://doi.org/10.4230/LIPIcs.TQC.2019.3). URL: <http://drops.dagstuhl.de/opus/volltexte/2019/10395>.
- [CD11] Bob Coecke and Ross Duncan. “Interacting Quantum Observables: Categorical Algebra and Diagrammatics”. In: *New Journal of Physics* 13.4 (Apr. 2011), p. 043016. ISSN: 1367-2630. DOI: [10.1088/1367-2630/13/4/043016](https://doi.org/10.1088/1367-2630/13/4/043016). (Visited on 11/07/2019).
- [Coq19] The Coq Development Team. *The Coq Proof Assistant, version 8.10.0*. Version 8.10.0. Oct. 2019. DOI: [10.5281/zenodo.3476303](https://doi.org/10.5281/zenodo.3476303). URL: <https://doi.org/10.5281/zenodo.3476303>.
- [Cro+17] Andrew W. Cross et al. “Open Quantum Assembly Language”. Comment: 24 pages, for additional examples and updates, see <https://github.com/IBM/qiskit-openqasm>. July 11, 2017. arXiv: [1707.03429](https://arxiv.org/abs/1707.03429).
- [DLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perles. “Social Processes and Proofs of Theorems and Programs”. In: *Commun. ACM* 22.5 (May 1979), pp. 271–280. ISSN: 0001-0782. DOI: [10.1145/359104.359106](https://doi.org/10.1145/359104.359106). URL: <https://doi.org/10.1145/359104.359106>.
- [DJ92] David Deutsch and Richard Jozsa. “Rapid solution of problems by quantum computation”. In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558.
- [Dev22] F* Developers. *F*: A Proof-Oriented Programming Language*. 2022. URL: <https://www.fstar-lang.org/>.
- [Dev21] Cirq Developers. *Cirq*. See full list of authors on Github: <https://github.com/quantumlib/Cirq>. 2021. URL: <https://doi.org/10.5281/zenodo.4586899>.

- [Ebe99] David Eberly. *Euler Angle Formulas*. Included in documentation for Geometric Tools. 1999. URL: <https://www.geometrictools.com/Documentation/EulerAngles.pdf>.
- [FD18] Andrew Fagan and Ross Duncan. “Optimising Clifford Circuits with Quantomatic”. In: *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*. 2018.
- [FP13] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 — Where Programs Meet Provers”. In: *Proceedings of the 22nd European Symposium on Programming*. Lecture Notes in Computer Science. 2013.
- [Fro+21] Aymeric Fromherz et al. “Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic”. In: *Proc. ACM Program. Lang.* 5.ICFP (2021). DOI: [10.1145/3473590](https://doi.org/10.1145/3473590).
- [Gel20] Alan Geller. *Introducing QIR*. Q# Blog. Sept. 2020. URL: <https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/>.
- [Gon+08] Georges Gonthier et al. “Formal proof—the four-color theorem”. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.
- [Got10] Daniel Gottesman. “An introduction to quantum error correction and fault-tolerant quantum computation”. In: *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*. Vol. 68. 2010, pp. 13–58.
- [Gre10] Alexander S. Green. “Towards a Formally Verified Functional Quantum Programming Language”. University of Nottingham, July 20, 2010. URL: <http://eprints.nottingham.ac.uk/11457/> (visited on 04/19/2021).
- [Gre+13] Alexander S. Green et al. “Quipper: A Scalable Quantum Programming Language”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA). Vol. 48. PLDI ’13. New York, NY, USA: ACM, June 2013, pp. 333–342. ISBN: 978-1-4503-2014-6. DOI: [10/gf8t6q](https://doi.org/10/gf8t6q). (Visited on 09/24/2019).
- [GHZ89] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. “Going Beyond Bell’s Theorem”. In: *Bell’s Theorem, Quantum Theory and Conceptions of the Universe*. Ed. by Menas Kafatos. Dordrecht: Springer Netherlands, 1989, pp. 69–72. ISBN: 978-94-017-0849-4. DOI: [10.1007/978-94-017-0849-4_10](https://doi.org/10.1007/978-94-017-0849-4_10). URL: https://doi.org/10.1007/978-94-017-0849-4_10.
- [Gro96] Lov K Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*. 1996, pp. 212–219.

- [Har21] Kevin Hartnett. *Proof Assistant Makes Jump to Big-League Math*. <https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/>. July 2021.
- [Hei20] Bettina Heim. “Development of Quantum Applications”. PhD thesis. ETH Zurich, 2020. Chap. 8: "Domain-Specific Language Q#". DOI: [10.3929/ethz-b-000468201](https://doi.org/10.3929/ethz-b-000468201).
- [Hie+20] Kesha Hietala et al. “Proving Quantum Programs Correct”. Oct. 3, 2020. arXiv: [2010.01240v2](https://arxiv.org/abs/2010.01240v2).
- [Hie+21] Kesha Hietala et al. “A Verified Optimizer for Quantum Circuits”. In: *Proceedings of the ACM on Programming Languages* 5 (POPL Jan. 4, 2021), 37:1–37:29. DOI: [10.1145/3434318](https://doi.org/10.1145/3434318). (Visited on 04/19/2021).
- [Hoa03] Tony Hoare. “The Verifying Compiler: A Grand Challenge for Computing Research”. In: *J. ACM* 50.1 (Jan. 2003), pp. 63–69. ISSN: 0004-5411. DOI: [10.1145/602382.602403](https://doi.org/10.1145/602382.602403). URL: <https://doi.org/10.1145/602382.602403>.
- [IBM22] IBM. *IBM Quantum Processor Types*. 2022. URL: <https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/processors>.
- [INR21] INRIA. *Interfacing C with OCaml*. <https://ocaml.org/manual/intfc.html>. Accessed: 2021-04-09. 2021.
- [INR22] INRIA. *Library Coq.Reals.Reals*. Accessed: 2022-03-23. 2022. URL: <https://coq.inria.fr/library/Coq.Reals.Reals.html>.
- [Jav+15] Ali Javadi-Abhari et al. “ScaffCC: Scalable Compilation and Analysis of Quantum Programs”. In: *Parallel Computing* 45 (2015). Computing Frontiers 2014: Best Papers, pp. 2–17. ISSN: 0167-8191. DOI: [10.1016/j.parco.2014.12.001](https://doi.org/10.1016/j.parco.2014.12.001). URL: <https://www.sciencedirect.com/science/article/pii/S0167819114001422>.
- [JPV18] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. “A complete axiomatisation of the ZX-calculus for Clifford+ T quantum mechanics”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM. 2018, pp. 559–568.
- [Kv19] Aleks Kissinger and John van de Wetering. “Reducing T-count with the ZX-calculus”. In: *arXiv e-prints* (2019). arXiv: [1903.10477 \[quant-ph\]](https://arxiv.org/abs/1903.10477).
- [KW19] Aleks Kissinger and John van de Wetering. “PyZX: Large Scale Automated Diagrammatic Reasoning”. In: *Proceedings of the 16th International Conference on Quantum Physics and Logic, QPL 2019*. June 2019.
- [KZ15] Aleks Kissinger and Vladimir Zamdzhev. “Quantomatic: A Proof Assistant for Diagrammatic Reasoning”. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 326–336.

- [Kle+09] Gerwin Klein et al. “SeL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: [10.1145/1629575.1629596](https://doi.org/10.1145/1629575.1629596). URL: <https://doi.org/10.1145/1629575.1629596>.
- [Kni96] E. Knill. *Conventions for Quantum Pseudocode*. LA-UR-96-2724. Los Alamos National Lab., NM (United States), June 1, 1996. DOI: [10.2172/366453](https://doi.org/10.2172/366453). URL: <https://www.osti.gov/biblio/366453-conventions-quantum-pseudocode> (visited on 04/19/2021).
- [Le+22] Xuan-Bach Le et al. “A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: [10.1145/3498697](https://doi.org/10.1145/3498697). URL: <https://doi.org/10.1145/3498697>.
- [Ler09] Xavier Leroy. “Formal verification of a realistic compiler”. en. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: [10/c9sb7q](https://doi.org/10/c9sb7q). URL: <http://doi.acm.org/10.1145/1538788.1538814> (visited on 09/30/2019).
- [LDX19] Gushu Li, Yufei Ding, and Yuan Xie. “Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 1001–1014. ISBN: 9781450362405. DOI: [10.1145/3297858.3304023](https://doi.org/10.1145/3297858.3304023). URL: <https://doi.org/10.1145/3297858.3304023>.
- [Liu+19a] Junyi Liu et al. “Formal Verification of Quantum Algorithms Using Quantum Hoare Logic”. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 187–207. ISBN: 978-3-030-25543-5. DOI: [10.1007/978-3-030-25543-5_12](https://doi.org/10.1007/978-3-030-25543-5_12).
- [Liu+19b] Junyi Liu et al. “Quantum Hoare Logic”. In: *Archive of Formal Proofs* (Mar. 2019). <http://isa-afp.org/entries/QHLP prover.html>, Formal proof development. ISSN: 2150-914x.
- [Liu+16] Tao Liu et al. “A theorem prover for quantum Hoare logic and its applications”. In: *arXiv preprint arXiv:1601.03835* (2016).
- [Mel20] Guillaume Melquiond. *Interval Package for Coq*. Mar. 2020. URL: <https://gitlab.inria.fr/coqinterval/interval>.
- [MT06] D. M. Miller and M. A. Thornton. “QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits”. In: *36th International Symposium on Multiple-Valued Logic (ISMVL’06)*. May 2006.

- [Myk20] Mariia Mykhailova. “The Quantum Katas: Learning Quantum Computing Using Programming Exercises”. In: *Proc. SIGCSE 2020*. ACM, 2020, p. 1417. DOI: [10.1145/3328778.3372543](https://doi.org/10.1145/3328778.3372543). URL: <https://github.com/microsoft/QuantumKatas>.
- [Nam+18] Yunseong Nam et al. “Automated optimization of large quantum circuits with continuous parameters”. In: *npj Quantum Information* 4.1 (2018), p. 23. DOI: [10.1038/s41534-018-0072-4](https://doi.org/10.1038/s41534-018-0072-4). URL: <https://doi.org/10.1038/s41534-018-0072-4>.
- [NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge: Cambridge University Press, 2010. DOI: [10.1017/CBO9780511976667](https://doi.org/10.1017/CBO9780511976667).
- [NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7.
- [PS14] Adam Paetznick and Krysta M Svore. “Repeat-until-success: non-deterministic decomposition of single-qubit unitaries”. In: *Quantum Information & Computation* 14.15-16 (2014), pp. 1277–1301.
- [PRZ17] Jennifer Paykin, Robert Rand, and Steve Zdancewic. “QWIRE: A Core Language for Quantum Circuits”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Vol. 52. New York, NY, USA: ACM, Jan. 2017, pp. 846–858. ISBN: 978-1-4503-4660-3. DOI: [10/gf8t6s](https://doi.acm.org/citation.cfm?doid=3009837.3009894). URL: <http://dl.acm.org/citation.cfm?doid=3009837.3009894> (visited on 06/12/2018).
- [Pen+22] Yuxiang Peng et al. “A Formally Certified End-to-End Implementation of Shor’s Factorization Algorithm”. In: *arXiv e-prints*, arXiv:2204.07112 (Apr. 2022), arXiv:2204.07112. arXiv: [2204.07112 \[cs.PL\]](https://arxiv.org/abs/2204.07112).
- [Per+14] Alberto Peruzzo et al. “A variational eigenvalue solver on a photonic quantum processor”. In: *Nature Communications* 5.1 (2014), p. 4213.
- [PE88] Frank Pfenning and Conal Elliott. “Higher-order Abstract Syntax”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208. DOI: [10.1145/53990.54010](https://doi.org/10.1145/53990.54010).
- [Pie+18] Benjamin C. Pierce et al. *Software Foundations*. Version 5.6. <https://softwarefoundations.cis.upenn.edu/>. Electronic textbook, 2018.
- [Pro+17] Jonathan Protzenko et al. “Verified Low-Level Programming Embedded in F^{*}”. In: *Proc. ACM Program. Lang.* 1.ICFP (2017). DOI: [10.1145/3110261](https://doi.org/10.1145/3110261).
- [Pty21] Python Software Foundation. *ctypes – A foreign function library for Python*. <https://docs.python.org/3/library/ctypes.html>. Accessed: 2021-04-09. 2021.

- [Qis21] Qiskit Development Team. *Transpiler Passes* (`qiskit.transpiler.passes`). https://qiskit.org/documentation/apidoc/transpiler_passes.html. Accessed: 2021-04-05. 2021.
- [Ran18] Robert Rand. “Formally Verified Quantum Programming”. Publicly Accessible Penn Dissertations. 3175. PhD Thesis. Philadelphia, PA, USA: University of Pennsylvania, 2018. 212 pp. URL: <https://repository.upenn.edu/edissertations/3175>.
- [Ran19] Robert Rand. *Verified Quantum Computing*. online. May 2019. URL: <http://www.cs.umd.edu/~rrand/vqc/index.html>.
- [RPZ18] Robert Rand, Jennifer Paykin, and Steve Zdancewic. “QWIRE Practice: Formal Verification of Quantum Circuits in Coq”. In: *Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, the Netherlands, 3-7 July 2017*. Ed. by Bob Coecke and Aleks Kissinger. Vol. 266. Electronic Proceedings in Theoretical Computer Science. Comment: In Proceedings QPL 2017, arXiv:1802.09737. Open Publishing Association, 2018, pp. 119–132. DOI: <10.4204/EPTCS.266.8>. arXiv: <1803.00699>.
- [Ran+19] Robert Rand et al. “ReQWIRE: Reasoning about Reversible Quantum Circuits”. In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 31, 2019). Comment: In Proceedings QPL 2018, arXiv:1901.09476, pp. 299–312. ISSN: 2075-2180. DOI: <10/ggb9c2>. arXiv: <1901.10118>.
- [Rey02] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: <10.1109/LICS.2002.1029817>.
- [Rig19a] Rigetti Computing. *Pyquil Documentation*. 2019. URL: <http://pyquil.readthedocs.io/en/latest/>.
- [Rig19b] Rigetti Computing. *The @rigetti optimizing Quil compiler*. 2019. URL: <https://github.com/rigetti/quilc>.
- [SWD11] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. “Synthesis of quantum circuits for linear nearest neighbor architectures”. In: *Quantum Information Processing* 10.3 (June 2011), pp. 355–377. ISSN: 1573-1332. DOI: <10.1007/s11128-010-0201-2>. URL: <https://doi.org/10.1007/s11128-010-0201-2>.
- [Sel04] Peter Selinger. “Towards a Quantum Programming Language”. In: *Mathematical Structures in Computer Science* 14.4 (Aug. 2004), pp. 527–586. DOI: <10/fgpw8k>. URL: <https://www.cambridge.org/core/journals/mathematical-structures-in-computer-science/article/towards-a-quantum-programming-language/54D5BCF28724CA6BE38F98DC4B6803DF> (visited on 09/25/2019).
- [Shi+19] Yunong Shi et al. “Contract-based verification of a realistic quantum compiler”. In: *arXiv e-prints* (Aug. 2019). arXiv: [1908.08963 \[quant-ph\]](1908.08963).

- [Shi+20] Yunong Shi et al. *CertiQ: A Mostly-automated Verification of a Realistic Quantum Compiler*. 2020. arXiv: [1908.08963 \[quant-ph\]](https://arxiv.org/abs/1908.08963).
- [Sho94] P. W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. FOCS ’94. 1994.
- [Sim94] DR Simon. “On the power of quantum computation”. In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 116–123.
- [Sin+22] Kartik Singhal et al. *Q# as a Quantum Algorithmic Language*. 2022. URL: http://rand.cs.uchicago.edu/files/q_sharp_algol.pdf.
- [Siv+20] Seyon Sivarajah et al. “ $t|ket\rangle$: a retargetable compiler for NISQ devices”. In: *Quantum Science and Technology* 6.1 (Nov. 2020), p. 014003. DOI: [10.1088/2058-9565/ab8e92](https://doi.org/10.1088/2058-9565/ab8e92). URL: <https://doi.org/10.1088/2058-9565/ab8e92>.
- [ST19] Kaitlin N. Smith and Mitchell A. Thornton. “A Quantum Computational Compiler and Design Tool for Technology-specific Targets”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA ’19. 2019.
- [SCZ16] Robert S. Smith, Michael J. Curtis, and William J. Zeng. “A Practical Quantum Instruction Set Architecture”. Aug. 11, 2016. arXiv: [1608 . 03355v2](https://arxiv.org/abs/1608.03355v2).
- [SHT18] Damian S. Steiger, Thomas Häner, and Matthias Troyer. “ProjectQ: An Open Source Software Framework for Quantum Computing”. In: *Quantum* 2 (Jan. 31, 2018), p. 49. DOI: [10 . 22331/q-2018-01-31-49](https://doi.org/10.22331/q-2018-01-31-49). URL: <https://quantum-journal.org/papers/q-2018-01-31-49/> (visited on 04/19/2021).
- [Svo+18] Krysta Svore et al. “Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL”. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018* (Vienna, Austria). RWDSL2018. Comment: 11 pages, no figures, REVTeX. New York, NY, USA: ACM, Feb. 24, 2018, 7:1–7:10. ISBN: 978-1-4503-6355-6. DOI: [10/gdcc72](https://doi.org/10/gdcc72). (Visited on 09/26/2019).
- [Swa+20] Nikhil Swamy et al. “SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs”. In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: [10.1145/3409003](https://doi.org/10.1145/3409003).
- [TQ19] Swamit S. Tannu and Moinuddin K. Qureshi. “Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. 2019. DOI: [10.1145/3297858.3304007](https://doi.org/10.1145/3297858.3304007).

- [Yan+11] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 283–294. DOI: [10.1145/1993316.1993532](https://doi.org/10.1145/1993316.1993532).
- [Yin12] Mingsheng Ying. “Floyd–Hoare Logic for Quantum Programs”. In: *ACM Trans. Program. Lang. Syst.* 33.6 (Jan. 2012). ISSN: 0164-0925. DOI: [10.1145/2049706.2049708](https://doi.org/10.1145/2049706.2049708). URL: <https://doi.org/10.1145/2049706.2049708>.
- [YMC22] Charles Yuan, Christopher McNally, and Michael Carbin. “Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: [10.1145/3498691](https://doi.org/10.1145/3498691). URL: <https://doi.org/10.1145/3498691>.
- [Zho+21] Li Zhou et al. “A Quantum Interpretation of Bunched Logic amp; Quantum Separation Logic”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2021, pp. 1–14. DOI: [10.1109/LICS52264.2021.9470673](https://doi.org/10.1109/LICS52264.2021.9470673).
- [ZPW17] Alwin Zulehner, Alexandru Paler, and Robert Wille. “An efficient methodology for mapping quantum circuits to the IBM QX architectures”. In: *arXiv e-prints* (Dec. 2017). arXiv: [1712.04722 \[quant-ph\]](https://arxiv.org/abs/1712.04722).