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 a classical problem: How can they be sure that their code does what they intend it to do? I aim 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. In support of this thesis, I present several 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 that contains state-of-the-art SQIR program optimizations with performance on par with unverified tools. I additionally discuss VQO, a framework for specifying and verifying oracle programs, which can then be optimized with VOQC. Finally, I present exploratory work on providing high assurance for a high-level industry quantum programming language, Q#, in the F* proof assistant.

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 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 work on the $\mathcal{Q}$WIRE 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 am also grateful to the programming languages communities at the University of Minnesota, the University of Maryland, and Microsoft Research for their support. In particular, I benefitted 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 thank my husband and my father for their unconditional support.

Table of contents

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 outputs over an exponentially large domain, so characterizing the output distribution may require exponentially many runs of the program. 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 used 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] or 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 a program's vector or matrix state. Using variables directly (e.g., with higher-order abstract syntax [PE88], as in the $\mathcal{Q}$WIRE [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 [Sel04a] and $\mathcal{Q}$WIRE), 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* [Sho97]. 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 $\mathcal{Q}$WIRE 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 $\mathcal{Q}$WIRE's Coq development, and takes inspiration and lessons from its design. Concurrently with our work, Chareton et al. [Cha+21] introduced $\mathcal{Q}$BRICKS, 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 $\mathcal{Q}$BRICKS have been used to verify similar algorithms (in particular, Grover's search and QPE), but $\mathcal{Q}$BRICKS 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 representation in a compiler. Compilers play a key role in the near-term quantum software toolchain because current machines are resource-limited. They have few qubits, restrictions on how qubits can be used together in quantum operations, limitations on the types of operations allowed, and high error rates, requiring that programs be short to prevent decoherence. It is the job of the compiler to perform the optimizations and transformations necessary to run a program on a quantum machine.

Such sophisticated optimizations are hard to get right: Kissinger and van de Wetering [Kv19a] 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. These bugs were found via *translation validation* (i.e., comparing the semantics of the compiler's target to its source) 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 in the previous section, 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

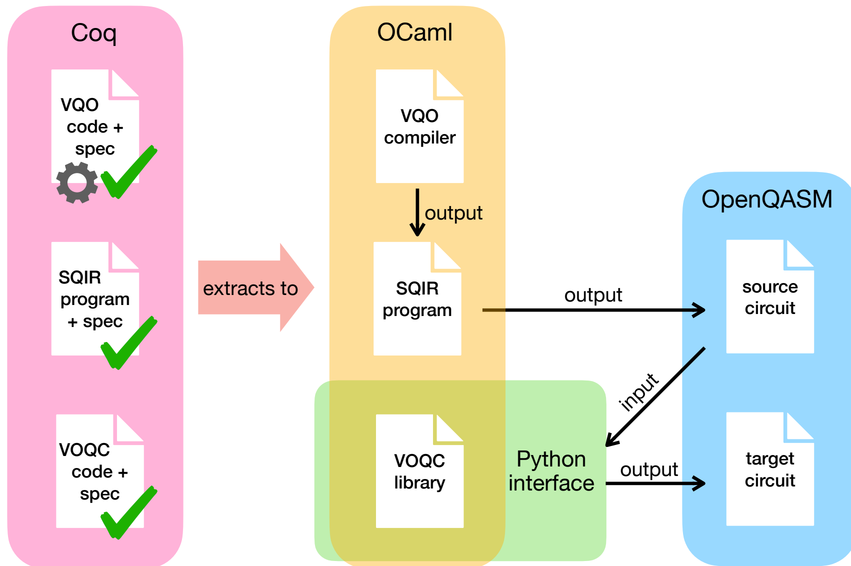


Figure 1.1: Overview of VOQC toolchain. Checks mark verified components and gears marks tested components.

developed by Nam et al. [Nam+18], but we also take inspiration from the Qiskit compiler [Qis17]. Along with optimizations, we also provide verified utilities for circuit mapping, which transform a SQIR program to satisfy constraints on how qubits may interact on a target architecture. We also look at the problem of compiling programs from a high-level language into SQIR: VQO is the first high-assurance compiler for *quantum oracles*, which are classical components leveraged by many quantum algorithms. VQO efficiently compiles programs written in a high-level classical language into SQIR, guaranteeing that the semantics of the compiled code matches the source.

The VOQC toolchain in summarized in Figure 1.1. VOQC optimizations, the VQO compiler, 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 and VQO into standalone OCaml libraries 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 [Qis17], t$|$ket$\rangle$ [Siv+20], Quil [Rig19b], and Cirq [Cir18].

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$\rangle$. We also find that the oracles produced by VQO have performance on par with those generated by Quipper [Gre+13], a popular unverified quantum programming framework.

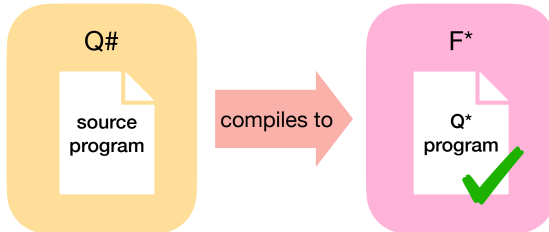


Figure 1.2: Overview of Q⋆ toolchain

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 (classical) 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, Tao et al. [Tao+22] developed Giallar, which uses symbolic execution and SMT solving to verify circuit transformations in the Qiskit compiler. Giallar 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 and means that it cannot be used as a general tool for verifying quantum programs.

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 directly encode the high-level constructs used in many 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 well-formedness properties not currently enforced by the Q# compiler. Q⋆ is the first attempt to support formal verification of programs written in an industry quantum programming language. Prior work on verifying quantum source programs, like SQIR, $\mathcal{Q}$WIRE, and $\mathcal{Q}$BRICKS, 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 optimizer for quantum circuits that contains implementations of state-of-the-art circuit transformations, and VQO, a framework for developing correct and efficient oracle programs. Evaluating on a large set of benchmark programs, we find that VOQC and VQO achieve performance comparable with popular unverified tools. (Chapter 4.)

3. We present $Q^\star$, the first effort to add the benefits of source-level formal verification to a widely-used industry 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 *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, especially in the development of *proof assistants* [Rin+19], which are general-purpose tools for defining mathematical structures and mechanizing proofs about those structures. This dissertation primarily uses the Coq proof assistant [Coq19], which is an established tool that has been used both to verify complex programs and to prove hard mathematical theorems. A prime example of a certified program is the CompCert compiler [Ler09], implemented and verified in Coq. CompCert compiles code written in the widely used C programming language to instruction sets for ARM, x86, and other computer architectures. CompCert's design precisely reflects the intended program behavior—the *semantics*—given in the C99 specification, and all of its optimizations are guaranteed to preserve that behavior. For this project, 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]. Coq has also been used to verify proofs of the notoriously hard-to-check Four Color Theorem [Gon08], as well as the Feit–Thompson (or odd order) theorem [Gon+13]. Coq's dual uses for both programming and mathematics make it an ideal tool for verifying quantum algorithms.

Many other proof assistants have similar success stories. The F* language (discussed in Chapter 5) is being used to certify a number of key internet security protocols, including Transport Layer Security (TLS) [Bha+17], and the High Assurance

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

(a) H matrix

$$\begin{pmatrix}1 & 0\\ 0 & e^{i\theta}\end{pmatrix}$$

(b) $Rz(\theta)$ matrix

$$\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$$

(c) $CNOT$ matrix

Figure 2.1: Matrix representation of common quantum gates

Cryptographic Library, HACL* [Zin+17], which has been integrated into the Firefox web browser. Isabelle/HOL was used to verify the seL4 operating system kernel. The Lean proof assistant has been used to verify large portions of the undergraduate and graduate mathematics curricula [Com20]. Indeed, Lean has reached the point where it can verify cutting-edge proofs: It was recently used to prove a core theorem in Peter Scholze's theory of condensed mathematics, first proven in 2019 [Cas21; Har21]. The approach taken in this dissertation to verify quantum software could be implemented using these other tools as well.

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 non-zero. Just as Schrodinger's cat is both dead and alive until the box is opened, a qubit is 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*—their conjugate transpose (or adjoint) 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 squared corresponds to 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.

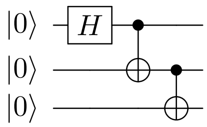
An important non-unitary quantum operator is *projection* onto a subspace. For example, $|0\rangle\langle 0|$ (in matrix notation $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$) projects a qubit onto the subspace where that qubit is in the $|0\rangle$ state. Projections are useful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

Most quantum programming languages in popular use today are expressed at a low-level of abstraction, essentially as a thin layer over circuits. Examples include Quil and IBM's quantum assembly language OpenQASM 2.0 [Cro+17]. In these languages, users are restricted to programming directly with gate applications and measurement, which often obscures the high-level goal of the program. Higher-level languages are now beginning to emerge to fill this gap. Some of these languages (e.g. Quipper [Gre+13], PyQuil [Rig19a]) focus on providing methods to easily manipulate circuits while others (e.g. Scaffold [Jav+12], Q# [Svo+18], and Silq [Bic+20]) focus on providing a programming environment familiar from classical programming.

Figure 2.2(c) shows a program in PyQuil, a quantum programming framework from Rigetti embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the Greenberger-Horne-Zeilinger state [GHZ89],



(a) Circuit

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

(b) Quil

```
def ghz_state(qubits):
  program = Program()
  program += H(qubits[0])
  for q1,q2 in zip(qubits, qubits[1:]):
    program += CNOT(q1, q2)
  return program
```

(c) PyQuil

```
operation PrepareGHZ (qs : Qubit[])
    : Unit is Adj + Ctl {
  H(qs[0]);
  ApplyCNOTChain(qs);
}
```

(d) Q#

Figure 2.2: Example quantum program: GHZ state preparation

which is an n-qubit entangled quantum state of the form

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

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.

Figure 2.2(d) shows the same program in Q#, a standalone quantum programming language from Microsoft. Unlike the PyQuil program, which explicitly constructs a circuit (called `program`), the Q# program applies `H` using the syntax of a function call, with no reference to a circuit object. It also provides a variety of high-level primitives, like `ApplyCNOTChain`, which applies a sequence of *CNOT* gates. The *characteristics* on the return type on the Q# program (`Adj + Ctl`) say that the compiler can automatically generate controlled or adjoint variants of the operation. We say more about Q# in Chapter 5.

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 quilc 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. quilc uses the following translation rules.

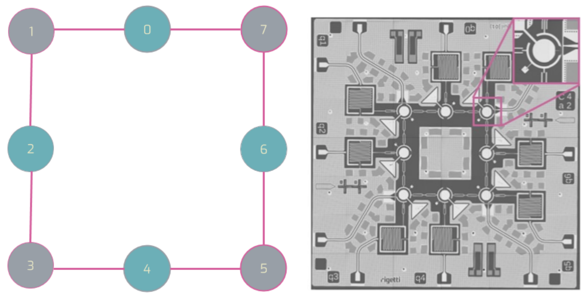
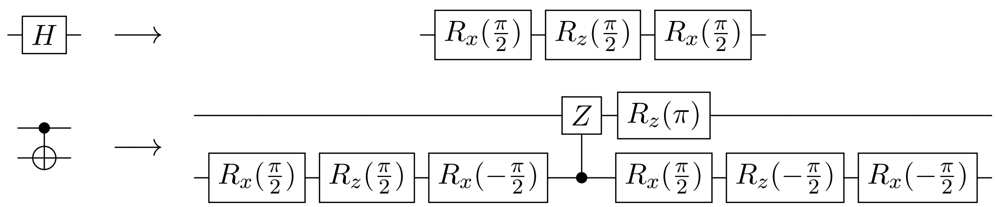
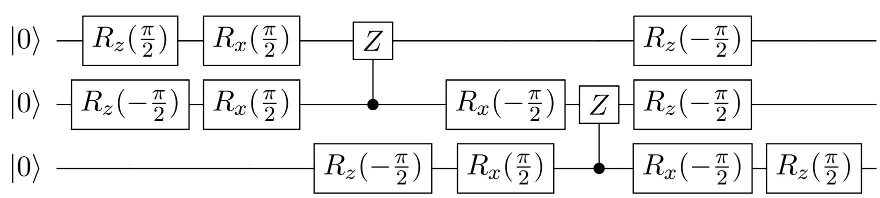


Figure 2.3: Rigetti's 8-qubit Agave machine [Com18] (deployed June 4, 2017). The processor consists of 8 superconducting transmon qubits, which can interact according to the graph on the left. The right shows an optical image of an 8-qubit chip representative of the Agave machine.



Along with performing gate set translation, quilc may also need to account for connectivity constraints, which restrict which qubits can be used together ("interact") in a two-qubit gate. For example, consider the diagram of Rigetti's Agave machine shown in Figure 2.3; On this machine, qubits are arranged in a ring, and only qubits adjacent on the ring can interact. In the program in Figure 2.2, qubits 0 and 1 interact and qubits 1 and 2 interact, which is allowed by the Agave machine, so no additional instructions need be inserted. Finally, quilc will perform optimizations that rewrite multiple R_x or R_z instructions acting on the same qubit into fewer instructions, producing the following program[1], which is sent to the hardware for execution.



Current high-level languages like Q# [Svo+18] and Silq [Bic+20] are intended to run on simulators rather than quantum hardware, so there has been limited work on compiling these languages to circuits. One notable exception is the compiler for the Scaffold programming language, ScaffCC [Jav+15].

[1] https://pyquil-docs.rigetti.com/en/v3.1.0/compiler.html

Chapter 3

A Quantum Intermediate Representation for Verification

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 [Sho97]. 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 3.1 presents SQIR's syntax and semantics. Section 3.2 discusses key elements of SQIR's design and compares and contrasts them to design decisions made in the related tools $\mathcal{Q}$WIRE [PRZ17], $\mathcal{Q}$BRICKS [Cha+21], 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 3.3 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`.

Acknowledgements This chapter is primarily based on two papers [Hie+21a; Hie+21b], which were joint work with Robert Rand, Shih-Han Hung, Liyi Li, Xiaodi Wu, and Michael Hicks. Section 3.3.3 describes joint work with Yuxiang Peng, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu on verifying

Shor's algorithm [Pen+22]. Robert initially conceived the idea of a simplified form of the $\mathcal{Q}$WIRE 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.

3.1 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.

3.1.1 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:

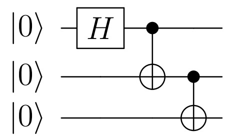
```
Inductive ucom (U: ℕ → Set) (d : ℕ) : Set :=
| useq  : ucom U d → ucom U d → ucom U d
| uapp1 : U 1 → ℕ → ucom U d
| uapp2 : U 2 → ℕ → ℕ → ucom U d
```

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]

```
Inductive base : ℕ → Set :=
  | U_R (θ ϕ λ : 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

[1] It 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.



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

```
Fixpoint ghz (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ I 0
  | 1 ⇒ H 0
  | S n' ⇒ ghz n'; CNOT (n'−1) n'
end.
```

(a) Quantum circuit (b) SQIR assembly (c) SQIR meta-program

Figure 3.1: GHZ state preparation in SQIR

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 3.1.2). 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 `uapp2 U_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.

3.1.2 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:

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

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 $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ 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 $\frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$, 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 3.2.6. 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 $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. 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 $[\![\texttt{c}]\!]_d$ for `uc_eval d c`. The denotation of composition is simple matrix multiplication: $[\![\texttt{U1; U2}]\!]_d = [\![\texttt{U2}]\!]_d \times [\![\texttt{U1}]\!]_d$. 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:

$$[\![\texttt{uapp1 U q}]\!]_d = \begin{cases} I_{2^q} \otimes [\![U]\!] \otimes I_{2^{d-q-1}} & q < d \\ 0_{2^d} & \text{otherwise} \end{cases}$$

where I_n is the $n \times n$ identity matrix. In the case of our `base` gate set, $[\![U]\!]$ 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 $[\![\texttt{CNOT q1 q2}]\!]_d$, we first decompose the $CNOT$ matrix into $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \otimes I_2 + \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right) \otimes X$. We then pad the expression appropriately, obtaining the following when $q_1 < q_2 < d$:

$$I_{2^{q_1}} \otimes \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes I_2 \otimes I_{2^{d-q_2-1}} \;+\; I_{2^{q_1}} \otimes \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes X \otimes I_{2^{d-q_2-1}}.$$

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 →
  ⟦SWAP 2 0 1⟧₂ × (φ ⊗ ψ) = ψ ⊗ φ.
```

`WF_Matrix` says that ϕ and ψ are well-formed matrices [Ran18, Section 2] (`Vector` is notation for a matrix with one column). 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 $\mathtt{n} > 0$ the circuit generated by `ghz n` produces the corresponding $GHZ(n)$ vector when applied to $|0\rangle^{\otimes n}$:

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

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'` $|0\rangle$ is the state $(\frac{1}{\sqrt{2}} * |0\rangle^{\otimes n'} + \frac{1}{\sqrt{2}} * |1\rangle^{\otimes n'}) \otimes |0\rangle$. By applying $\mathtt{CNOT}\ (\mathtt{n'} - 1)\ \mathtt{n'}$ to this state, we show that $\mathtt{ghz}\ (\mathtt{n'} + 1) = GHZ(n' + 1)$.

3.1.3 Full SQIR: Adding Measurement

The full SQIR language adds a branching measurement construct inspired by Selinger's QPL [Sel04a]. 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: ℕ → Set) (d : ℕ) : Set :=
| uc   : ucom U d → com U d
| skip : com U d
| meas : ℕ → com U d → com U d → com U d
| seq  : com U d → com U d → com U d.
```

The command `meas q` P_1 P_2 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 [Sel04b; 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 $\{\!|\texttt{P}|\!\}_d$ of a full SQIR program `P` is a function from density matrices to density matrices. Naturally, $\{\!|\texttt{skip}|\!\}_d\ \rho = \rho$ and $\{\!|\texttt{P1 ; P2}|\!\}_d = \{\!|\texttt{P2}|\!\}_d \circ \{\!|\texttt{P1}|\!\}_d$. For unitary subroutines, we have $\{\!|\texttt{uc U}|\!\}_d\ \rho\ = [\![\texttt{U}]\!]_d \rho [\![\texttt{U}]\!]_d{}^\dagger$: 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 $\{\!|\texttt{meas q P1 P2}|\!\}_d\ \rho$ is

$$\{\!|P_1|\!\}_d(|1\rangle_q\langle 1|\,\rho\,|1\rangle_q\langle 1|) + \{\!|P_2|\!\}_d(|0\rangle_q\langle 0|\,\rho\,|0\rangle_q\langle 0|)$$

which corresponds to probabilistically applying `P1` to ρ with the specified qubit projected to $|1\rangle\langle 1|$ or applying `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` : $\{\!|\texttt{coin}|\!\}_1\ |0\rangle\langle 0| = \frac{1}{2}|1\rangle\langle 1| + \frac{1}{2}|0\rangle\langle 0|$.

The proof proceeds by simple matrix arithmetic. $\{\!|\texttt{H}|\!\}\ |0\rangle\langle 0|$ is $H\,|0\rangle\langle 0|\,H^\dagger = \frac{1}{2}\left(\begin{smallmatrix}1&1\\1&1\end{smallmatrix}\right)$. 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 that $\langle 1|\,\rho_{12}\,|1\rangle = \langle 0|\,\rho_{12}\,|0\rangle = (\,\frac{1}{2}\,)$, 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 `u` will (deterministically) evaluate to $[\![\texttt{u}]\!]_d \times |\psi\rangle$. However, `meas q p1 p2` may evaluate to either `p1` applied to $|1\rangle\langle 1| \times |\psi\rangle$ or `p2` applied to $|0\rangle\langle 0| \times |\psi\rangle$ (we ignore scaling for simplicity). We use notation `p` / $\psi \Downarrow \psi'$ to say that on input ψ program `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 3.2.3). 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 `q` to the $|0\rangle$ state.

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

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.

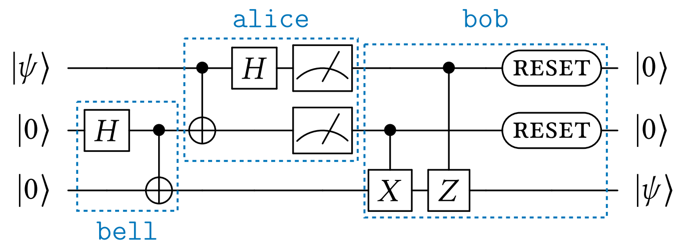


Figure 3.2: Circuit for quantum teleportation. In the standard presentation Bob only acts on the last qubit, given two classical bits as input. In our presentation, Bob (equivalently) performs operations controlled by the first two qubits, which are in a post-measurement classical state. We include the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

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

The proof is straightforward:

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

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`: $\forall\ (\psi\ \psi' : \texttt{Vector}\ 2),\ \texttt{WF\_Matrix}\ \psi \rightarrow \texttt{reset}\ /\ \psi \Downarrow \psi' \rightarrow \psi' \propto |0\rangle$.

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 $|\psi\rangle \otimes |00\rangle$, `teleport` will produce a state that is proportional to $|00\rangle \otimes |\psi\rangle$.

```
Lemma teleport_correct : ∀ (ψ : Vector (2^1)) (ψ' : Vector (2^3)),
  WF_Matrix ψ → teleport / (ψ  ⊗ |00⟩) ⇓ ψ' → ψ' ∝ |00⟩ ⊗ 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.

3.1.4 Running SQIR Programs

Coq is a language designed for formal proof, not program execution. In order to produce efficient, executable verified code, a common workflow is to define a function and prove properties about it in Coq, and then *extract* the function to OCaml (or Haskell), where it can be compiled with a standard optimizing compiler and run on a machine. We follow a similar approach: we extract Coq programs that generate SQIR into OCaml using Coq's standard extraction mechanism [Inr21b] and we provide a simple translation function to write a SQIR program to a file in the OpenQASM 2.0 format [Cro+17]. The generated OpenQASM can then be sent to a quantum computer (or a classical simulation of a quantum computer) to be executed.

As an example, the `ghz` program from Figure 3.1(c) will be extracted to the OCaml program shown in Figure 3.3(a), which, given input 3, produces a SQIR program that translates to the OpenQASM program in Figure 3.3(b). In Figure 3.3(a), `coq_SKIP`, `coq_H`, and `coq_CNOT` are the extracted forms of `SKIP`, `H`, and `CNOT` used in Figure 3.1 and `Coq_useq` is the extracted form of the `useq` constructor. Our proof of correctness for the `ghz` function in Coq guarantees that the output OpenQASM program produces the mathematical GHZ state.[2]

[2]This assumes that extraction produces OCaml code consistent with our Coq definitions and that we do not introduce errors in our conversion from SQIR to OpenQASM. One potential issue is

```
let rec ghz n =
  if n=0 then coq_SKIP else
  if n=1 then coq_H 0 else
  Coq_useq (ghz (n−1), coq_CNOT (n−2) (n−1))
```

(a) Extracted OCaml code (prettified)

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
h q[0];
cx q[0], q[1];
cx q[1], q[2];
```

(b) Generated OpenQASM circuit

Figure 3.3: Result of extracting the example from Figure 3.1

Choice of Gate Set In Section 3.1.1 we presented the base gate set consisting of `U_R` and `U_CNOT`. This set is conveniently simple for proof, but inconvenient for extraction because quantum hardware (and simulators) usually support a different set of gates, and naïvely forcing output to be in this gate set will significantly increase the size of the output circuits, making them prohibitively expensive to run.

The culprit is the `control` function shown in Figure 3.4, which is used in all the algorithm examples discussed in Section 3.3. This function applies the textbook decompositions of controlled applications of `U_R` (`CU`) and `U_CNOT` (`CCX`) [NC10, Chapter 4]. We have verified that this function is correct; it will always produce a controlled application of its input (assuming some well-typedness constraints). However, it can also generate highly inefficient code. In our evaluation of VQO (Appendix B) we found that decomposing the CCU_1 gate as `control _ (control _ (U1 _ _))` could lead to a 4.4× increase in the number of output gates, even after applying VOQC optimizations.

To avoid this unnecessary blowup, when extracting to OpenQASM we define programs over the gate set X, H, U_1, U_2, U_3, CX, CH, CU_1, $SWAP$, CCX, CCU_1, $CSWAP$, $C3X$, and $C4X$. X, H are the Pauli X and Hadamard gates. U_1, U_2, U_3 are the single-qubit rotation gates from the OpenQASM standard library [Cro+17]. CU_1 is the controlled version of the U_1 gate and CCU_1 is the controlled version of CU_1. $SWAP$ and $CSWAP$ are the swap gate and its controlled version. $CX, CCX, C3X$, and $C4X$ are controlled versions of the X gate, with different numbers of control qubits. In particular, CX is the $CNOT$ gate. We chose these gates because they were the ones we observed in our applications that required extraction, namely Shor's algorithm (Section 3.3.3) and VQO (Section 4.5). The (verified) decompositions of these gates are shown in Figure 3.5.

In order to take a property proved about a program in the base gate set and apply it to a program in the new gate set, we simply prove that the new definition is equivalent to the old. Thus, this larger gate set does not complicate proof; it is not used in involved correctness proofs, but only in simpler (largely automated) equivalence proofs.

---

that we extract Coq's axiomatized Reals to OCaml floats—see Section 4.1.4 for details. We have tested our extraction process by generating order-finding circuits (Section 3.3.3) for varying sizes and confirming that they produce the expected results in a simulator.

```
Definition CCX {dim} a b c : ucom base dim :=
  H c ; CNOT b c ; TDAG c ; CNOT a c ;
  T c ; CNOT b c ; TDAG c ; CNOT a c ;
  CNOT a b ; TDAG b ; CNOT a b ;
  T a ; T b ; T c ; H c.

Definition CU {dim} θ φ λ c t : ucom base dim :=
  Rz ((λ + φ)/2) c ; Rz ((λ − φ)/2) t ;
  CNOT c t ; uapp1 (U_R (−θ/2) 0 (−(φ + λ)/2)) t ;
  CNOT c t ; uapp1 (U_R (θ/2) φ 0) t.

Fixpoint control {dim} q (c : ucom base dim) : ucom base dim :=
  match c with
  | c1; c2 ⇒ control q c1; control q c2
  | uapp1 (U_R θ φ λ) n ⇒ CU θ φ λ q n
  | uapp2 U_CNOT m n ⇒ CCX q m n
  | _ ⇒ SKIP
  end.

Lemma control_correct : ∀ d q (c : ucom base d),
  is_fresh q c → uc_well_typed c →
  ⟦control q c⟧_d = |0⟩_q ⟨0| + |1⟩_q ⟨1| × ⟦c⟧_d.
Proof.
...
Qed.
```

Figure 3.4: `control` function for SQIR programs using the base gate set. Braces mark an argument as implicit, meaning that it will be inferred by the compiler. `Rz` λ is shorthand for `uapp1` (`U_R` 0 0 λ), and `T` and `TDAG` are `Rz` (`PI`/4) and `Rz` (−`PI`/4).

3.2 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).

3.2.1 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

```
Definition P8 {dim} n : ucom base dim := Rz (PI / 8) n.

Definition P8DAG {dim} n : ucom base dim := Rz (- (PI / 8)) n.

Definition CH {dim} (a b : ℕ) : ucom base dim :=
  U3 (PI/4) 0 0 b ; CNOT a b ; U3 (- (PI/4)) 0 0 b.

Definition CU1 {dim} r1 (a b : ℕ) : ucom base dim :=
  U1 (r1/2) a ; U1 (r1/2) b ; CNOT a b ; U1 (- (r1/2)) b ; CNOT a b.

Definition CCU1 {dim} r1 (a b c : ℕ) : ucom base dim :=
  CU1 (r1/2) a b ; CNOT b c ; CU1 (-r1/2) a c ; CNOT b c ; CU1 (r1/2) a c.

Definition CSWAP {dim} (a b c : ℕ) : ucom base dim := CNOT c b ; CCX a b c ; CNOT c b.

Definition C3X {dim} (a b c d : ℕ) : ucom base dim :=
  H d ; P8 a ; P8 b ; P8 c ; P8 d ; CNOT a b ; P8DAG b ; CNOT a b ; CNOT b c ;
  P8DAG c ; CNOT a c ; P8 c ; CNOT b c ; P8DAG c ; CNOT a c ; CNOT c d ;
  P8DAG d ; CNOT b d ; P8 d ; CNOT c d ; P8DAG d ; CNOT a d ; P8 d ; CNOT c d ;
  P8DAG d ; CNOT b d ; P8 d ; CNOT c d ; P8DAG d ; CNOT a d ; H d.

Definition RC3X {dim} (a b c d : ℕ) : ucom base dim :=
  H d ; T d ; CNOT c d ; TDAG d ; H d ; CNOT a d ; T d ; CNOT b d ; TDAG d ;
  CNOT a d ; T d ; CNOT b d ; TDAG d ; H d ; T d ; CNOT c d ; TDAG d ; H d.

Definition RTX {dim} r q : ucom base dim := H q ; U1 r q ; H q.

Definition CRTX {dim} (r : R) (a b : ℕ) : ucom base dim := H b ; CU1 r a b ; H b.

Definition C3SQRTX {dim} (a b c d : ℕ) : ucom base dim :=
  CRTX (PI/8) a d ; CNOT a b ; CRTX (-PI/8) b d ; CNOT a b ; CRTX (PI/8) b d ;
  CNOT b c ; CRTX (-PI/8) c d ; CNOT a c ; CRTX (PI/8) c d ;
  CNOT b c ; CRTX (-PI/8) c d ; CNOT a c ; CRTX (PI/8) c d.

Definition C4X {dim} (a b c d e : ℕ) : ucom base dim :=
  CRTX (PI/2) d e ; RC3X a b c d ; CRTX (-PI/2) d e ;
  invert (RC3X a b c d) ; C3SQRTX a b c e.
```

Figure 3.5: Decompositions for CH, CU_1, CCU_1, $CSWAP$, $C3X$, and $C4X$ based on those available in Qiskit [Qis21].

```
box (x,y,z) ⇒
  gate x     ← H x;
  gate (x,y) ← CNOT (x,y);
  gate (y,z) ← CNOT (y,z);
  output (x,y,z).
```

(a) $\mathcal{Q}$WIRE

```
SEQ(SEQ(PAR(H, PAR(I, I)),
        PAR(CNOT, I)),
    PAR(I, CNOT))
```

(b) $\mathcal{Q}$BRICKS-DSL

```
q1 := H q1;
q1,q2 := CNOT q1,q2;
q2,q3 := CNOT q2,q3
```

(c) QWhile

$$(I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

(d) Matrix expression

Figure 3.6: Alternate descriptions of the GHZ program in Figure 3.1(a–b)

reasoning about quantum "programs" directly via their matrix semantics (e.g. Figure 3.6(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 quantum programs expressed as matrix products (Figure 3.6(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: $\mathcal{Q}$WIRE [Ran18] (implemented in Coq [Coq19]); quantum Hoare logic [Liu+19b] (in Isabelle/HOL [NWP02]); and $\mathcal{Q}$BRICKS [Cha+21] (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 $\mathcal{Q}$WIRE and the quantum Hoare logic. All matrix formalisms provided by Bordg et al. are available in the QuantumLib library [The22], which we developed for SQIR based on content in the $\mathcal{Q}$WIRE project.

QWIRE The $\mathcal{Q}$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 3.6(a) shows the $\mathcal{Q}$WIRE equivalent of the SQIR program in Figure 3.1(b). $\mathcal{Q}$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 $\mathcal{Q}$WIRE program uses variables x, y, and z, while the SQIR program uses indices 0, 1, and 2 to refer to the first, second, and third qubits in the global register. $\mathcal{Q}$WIRE does not distinguish between unitary and non-unitary programs, and thus uses density matrices for its semantics. $\mathcal{Q}$WIRE has been used to verify simple randomness generation circuits and a few textbook examples [Ran18].

QBRICKS $\mathcal{Q}$BRICKS [Cha+21] is a quantum proof framework implemented in Why3 [FP13], developed concurrently with SQIR. $\mathcal{Q}$BRICKS provides a domain-specific language (DSL) for constructing quantum circuits using combinators for parallel and sequential composition (among others). Figure 3.6(b) presents the GHZ example written in $\mathcal{Q}$BRICKS' DSL. The semantics of $\mathcal{Q}$BRICKS are based on the *path-sums* formalism by Amy [Amy19a; Amy19a], 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 3.6(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 3.6(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 repository[3] and believe that this approach was abandoned in favor of Liu et al. [Liu+19a].

3.2.2 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 3.1.1), `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 subprogram composition. Unfortunately, abstract qubits significantly complicate formal verification. To translate circuits to operations on density matrices, variables must be

[3] https://github.com/ijcar2016/propitious-barnacle

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 application can be computed by its nested position in the program. However, this syntax is even less convenient than SQIR's for programming: Although $\mathcal{Q}$BRICKS 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 3.6(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.

3.2.3 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 $\mathcal{Q}$WIRE 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 3.2.7).

On the other hand, $\mathcal{Q}$BRICKS' 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 the "best of both worlds," utilizing a unitary semantics when possible, but supporting non-unitary semantics when needed. Furthermore, as we show in section 3.2.6, abstractions like path-sums can be easily defined on top of SQIR's unitary semantics.

3.2.4 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 $a \neq b$.

$\mathcal{Q}$WIRE addresses this issue through its linear type system, which also guarantees that qubits are never reused. However, well-typedness is a (nontrivial) extrinsic proposition in $\mathcal{Q}$WIRE, meaning that many proofs require an assumption that the input program is well-typed and must manipulate this typing judgment within the proof. $\mathcal{Q}$BRICKS 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. `SWAP d a b` $\equiv$ `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. `SWAP d a c` $\equiv$ `SWAP d a b` ; `SWAP d b c` ; `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 $\texttt{b} < \texttt{d} \wedge \texttt{b} \neq \texttt{a} \wedge \texttt{b} \neq \texttt{c}$. We capture these in our `uc_well_typed` predicate, which resembles the `WF_Matrix` predicate (used in the `SWAP` example in Section 3.1.2) that guarantees that a matrix's non-zero entries are all within its bounds. Both conditions are easily checked via automation.

3.2.5 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. $\forall\ a\ b\ c$, `CNOT a c` ; `CNOT b c` $\equiv$ `CNOT b c` ; `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]. $\mathcal{Q}$BRICKS 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 $\mathcal{Q}$WIRE [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.

3.2.6 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` : ℕ and `f` : ℕ → 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⟩ for `f_to_vec n f` and |i⟩ 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 lemmas succinctly describe the behavior of the $CNOT$ and $Rz(\theta)$ gates, where $Rz(\theta) = U_R(0, 0, \theta)$:

```
Lemma f_to_vec_CNOT : ∀ (d i j : ℕ) (f : ℕ → 𝔹),
  i < d → j < d → i ≠ j →
  let f' := update f j (f j ⊕ f i) in
  ⟦CNOT i j⟧_d × |f⟩ = |f'⟩.

Lemma f_to_vec_Rz: ∀ (d j : ℕ) (θ : R) (f : ℕ → 𝔹),
  j < d →
  ⟦Rz θ j⟧_d × |f⟩ = e^{iθ(f j)} ∗ |f⟩.
```

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 $[\![\texttt{CNOT i j}]\!]_d$ and $[\![\texttt{Rz}\ \theta\ \texttt{j}]\!]_d$. 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 ($CNOT \mapsto \_ \otimes \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \otimes \_ \otimes I_2 \otimes \_ \ + \ \_ \otimes \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right) \otimes \_ \otimes \sigma_x \otimes \_$).

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 2^n := ...
```

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

$$\texttt{vkron n (fun i} \Rightarrow \tfrac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(i)}\,|1\rangle)) \quad \text{or}$$
$$\tfrac{1}{\sqrt{2^n}} * (\texttt{vsum}\ 2^n\ (\texttt{fun i} \Rightarrow (-1)^{\texttt{to\_int}(f)\bullet i} * |i\rangle)),$$

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; Amy19a] semantics used by $\mathcal{Q}$BRICKS. 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 $|x\rangle \rightarrow \frac{1}{\sqrt{2}} \sum_{y=0}^{1} e^{2\pi i x y/2} |y\rangle$. 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 : ℕ) P f x :=
  vsum 2^m (fun y ⇒ e^{2πiP(x,y)/2^m} ∗ (vkron m (fun i ⇒ 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.

3.2.7 Measurement Predicates

The proofs in Section 3.3 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 φ given input ψ. *)
Definition probability_of_outcome {n} (φ ψ : Vector n) : R := |φ · ψ|^2.

(* Prob. of measuring φ on the first n qubits given (n+m) qubit input ψ. *)
Definition prob_partial_meas {n m} (φ : Vector 2^n) (ψ : Vector 2^(n+m)) :=
  ‖ (φ† ⊗ I_{2^m}) × ψ ‖^2.
```

Above, $\varphi \cdot \psi$ is the dot product of φ and ψ, $\|v\|$ 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 $\{\varphi \times \varphi^\dagger, \mathtt{I}_{2^n} - \varphi \times \varphi^\dagger\}$.

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]. $\mathcal{Q}$BRICKS also uses measurement predicates, but unlike SQIR does not support a general measurement construct.

```
(* 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 3.7: 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.

3.3 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 [Sho97]. The proofs and specifications follow the standard textbook arguments.

3.3.1 Grover's Algorithm

Overview Given a circuit implementing Boolean oracle $f : \{0,1\}^n \to \{0,1\}$, 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 3.7.

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 Sketch 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 ($\psi\mathtt{g}$) that satisfy f and "bad" states ($\psi\mathtt{b}$) that do not.

Definition $\psi := \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle$.
Definition $\theta :=$ asin $(\sqrt{k/2^n})$.
Lemma decompose_ψ : $\psi =$ (sin θ) ψg + (cos θ) ψb.

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

Lemma loop_body_action_on_unif_superpos : ∀ i,
$[\![\mathtt{body}]\!]_{n+1}^{i}\ (\psi \otimes |-\rangle) =$
$(-1)^i$ (sin ((2 ∗ i + 1) ∗ θ) ψg + cos ((2 ∗ i + 1) ∗ θ) ψb) $\otimes |-\rangle$.

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 : ∀ i,
$\sum_{z=0}^{2^n}$ (if f z then prob_partial_meas $|z\rangle$ ($[\![\mathtt{grover\ i}]\!]_{n+1} \times |0\rangle^{n+1}$) else 0) =
(sin ((2 ∗ i + 1) ∗ θ))^2.

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

3.3.2 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 [Sho97].

The sqir program for QPE is shown in Figure 3.8. 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 3.9. 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 nontrivial 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].

```
(* 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 / 2^n) 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. *)
Fixpoint reverse_qubits' dim n : ucom base dim :=
  match n with
  | 0    ⇒ SKIP
  | S n' ⇒ reverse_qubits' dim n' ; SWAP n' (dim − n' − 1)
  end.
Definition reverse_qubits n := reverse_qubits' n (n/2).
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 2^k' (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 3.8: 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. $\forall\ u.\ [\![\text{invert } u]\!]_n = [\![u]\!]_n^\dagger$).

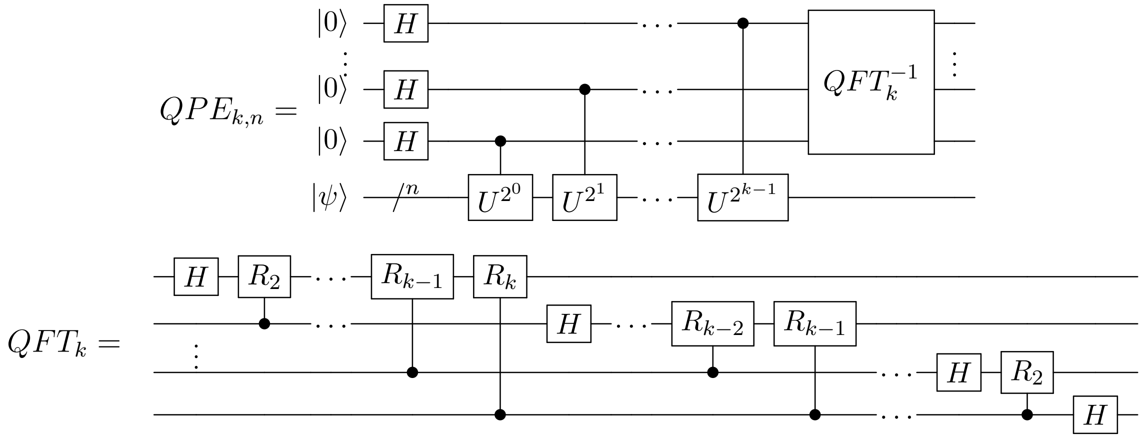


Figure 3.9: Circuit for quantum phase estimation (QPE) with k bits of precision and an n-qubit input state (top) and quantum Fourier transform (QFT) on k qubits (bottom). $|\psi\rangle$ and U are inputs to QPE. R_m is a z-axis rotation by $2\pi/2^m$.

Proof Sketch 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.

```
Lemma QPE_correct_simplified: ∀ k n (u : ucom base n) z (ψ : Vector 2^n),
  n > 0 → k > 1 → uc_well_typed u → WF_Matrix ψ →
  let θ := z / 2^k in
  ⟦u⟧_n × ψ = e^{2πiθ} ∗ ψ →
  ⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ 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 2^n) (δ : R),
  n > 0 → k > 1 → uc_well_typed u → Pure_State_Vector ψ →
  −1 / 2^{k+1} ≤ δ < 1 / 2^{k+1} → δ ≠ 0 →
  let θ := z / 2^k + δ in
  ⟦u⟧_n × ψ = e^{2πiθ} ∗ ψ →
  prob_partial_meas |z⟩ (⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ ψ)) ≥ 4 / π^2.
```

`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 $e^{2\pi i(x_0 \cdot x_1x_2 \ldots x_{n-1})/2^n}|x\rangle$ where x_0 is the highest-order bit of x represented as a binary string and $x_1x_2\ldots x_{n-1}$ are the lower-order $n-1$ bits.

$$
\begin{aligned}
&[\![\texttt{controlled\_rotations (n+1)}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control}\ x_n\ (\texttt{Rz}\ (2\pi/2^{n+1})\ 0]\!]_{n+1} \times [\![\texttt{controlled\_rotations n}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control}\ x_n\ (\texttt{Rz}\ (2\pi/2^{n+1})\ 0]\!]_{n+1} \times e^{2\pi i(x_0\ \cdot\ x_1 x_2 \ldots x_{n-1})/2^n}\ |x_1 x_2 \ldots x_{n-1} x_n\rangle \\
&\quad = e^{2\pi i(x_0\ \cdot\ x_n)/2^{n+1}} e^{2\pi i(x_0\ \cdot\ x_1 x_2 \ldots x_{n-1})/2^n}\ |x_1 x_2 \ldots x_{n-1} x_n\rangle \\
&\quad = e^{2\pi i(x_0\ \cdot\ x_1 x_2 \ldots x_n)/2^{n+1}}\ |x_1 x_2 \ldots x_{n-1} x_n\rangle
\end{aligned}
$$

Figure 3.10: 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.

```
Lemma controlled_rotations_correct : ∀ n x,
  n > 1 → ⟦controlled_rotations n⟧_n × |x⟩ = e^{2πi(x_0 · x_1x_2...x_{n-1})/2^n}|x⟩.
```

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:

$$
[\![\texttt{controlled\_rotations n}]\!]_n \times |x_1 x_2 \ldots x_{n-1}\rangle = e^{2\pi i(x_0\ \cdot\ x_1 x_2 \ldots x_{n-1})/2^n} |x_1 x_2 \ldots x_{n-1}\rangle.
$$

We then perform the simplifications shown in Figure 3.10, which complete the proof.

3.3.3 Shor's Prime Factorization Algorithm

Overview Shor's prime factorization algorithm [Sho97] has been a fundamental motivation for the development of quantum computers as it provides a method to break widely-used cryptographic systems, including RSA. A recent study [GE21] suggests that with 20 million noisy qubits, it would take a few hours for Shor's algorithm to factor a 2048-bit key instead of trillions of years as would be required by modern classical computers using the best-known methods. We have produced the first fully-certified implementation of Shor's prime factorization algorithm, which is the most sophisticated quantum algorithm verified to date.

Shor's quantum-classical hybrid algorithm for factoring a number N is summarized in Figure 3.11: the key quantum part (order finding) is preceded and followed by classical computation (primality testing before, and conversion of found orders to prime factors after). Our definition of Shor's algorithm in Coq is shown in Figure 3.12. It uses the `QPE` function presented in the previous section, applied to an oracle `IMM` that performs *in-place modular multiplication*. Modular multiplication is fundamentally a classical operation—it transforms a classical state into a classical state. When attempting to implement and verify `IMM` directly in SQIR, we found that SQIR's general semantics of matrices of complex numbers unecessarily complicated proof. This led us to develop a new language in Coq for expressing classical circuits—called *reversible*

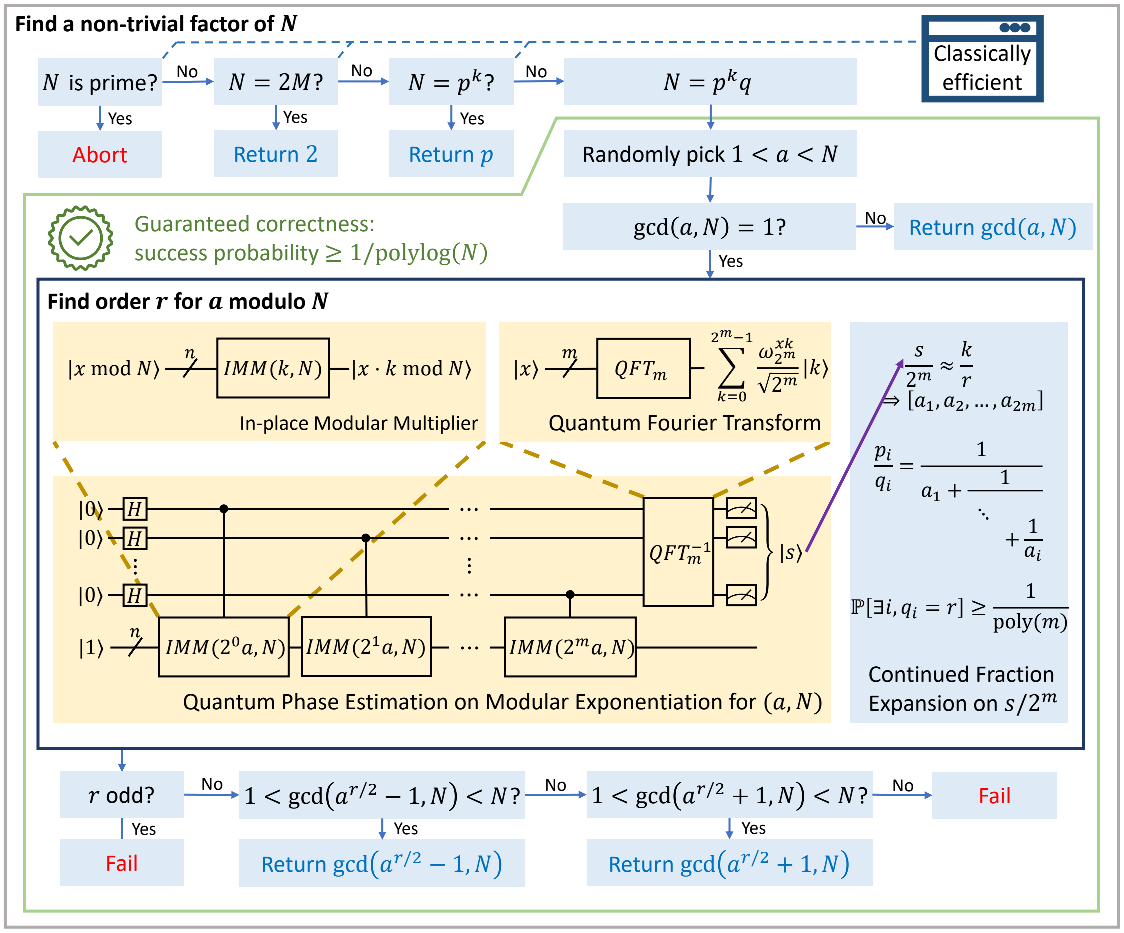


Figure 3.11: Overview of Shor's factoring algorithm from Peng et al. [Pen+22]. The goal is to find a nontrivial factor of integer N. The algorithm begins by performing classical preprocessing to identify cases where N is prime, even, or a prime power, in which case no nontrivial factor exists or factoring can be done efficiently classically. Next, it chooses a uniformly at random from the numbers 1 through N. If a is not coprime to N, then $\gcd(a, N)$ is a nontrivial factor of N, so the algorithm terminates. Otherwise, it invokes the hybrid quantum-classical order finding procedure bounded by the dark box. The quantum part of the order finding subroutine applies quantum phase estimation (QPE) to an oracle implementing in-place modular multiplication (IMM). Roughly speaking, the output of QPE over IMM is a close approximation of k/r for a k uniformly sampled from $\{0, 1, \cdots, r-1\}$. The classical part then uses the continued fraction expansion (CFE) $[a_1, a_2, \cdots, a_{2m}]$ to recover the order r of a modulo N (i.e., the smallest positive integer r such that $a^r \equiv 1 \mod N$). Further classical postprocessing rules out cases where r is odd before outputting the nontrivial factor. We have verified implementations of all routines inside the green outline.

circuit intermediate representation (RCIR)—that has a verified compiler to SQIR. We implement `IMM` using RCIR.[4]

In Figure 3.12, `shor_body` performs the algorithm enclosed in the green outline in Figure 3.11. `shor_circuit` generates the QPE circuit applied to `IMM`, `OF_post` applies continued fraction expansion, and `factor` performs classical postprocessing given an order candidate r. `end_to_end_shors` iterates `shor_body`, returning a factor if any iteration succeeds.

```
Definition shor_circuit (a N : ℕ) :=
 let m := log2 (2*N^2) in       (* number of qubits storing QPE output *)
 let n := log2 (2*N) in         (* number of data qubits used in IMM *)
 let f i := IMM (modexp a (2^i) N) N n in
 X (m + n − 1); QPE m f.

Definition factor (a N r : ℕ) : option ℕ :=
 let cand1 := gcd (a ^ (r / 2) − 1) N in
 let cand2 := gcd (a ^ (r / 2) + 1) N in
 if (1 <? cand1) && (cand1 <? N) then Some cand1
 else if (1 <? cand2) && (cand2 <? N) then Some cand2
 else None.

Definition shor_body (N : ℕ) (rnd : R) : option ℕ :=
 let m := log2 (2*N^2) in              (* number of qubits storing QPE output *)
 let k := 4*log2 (2*N)+11 in           (* total number of qubits used in IMM *)
 let adist := uniform 1 N in           (* create a uniform distribution *)
 let a := sample adist rnd in          (* choose an a *)
 if gcd a N =? 1                       (* do a and N share a factor? *)
 then
  let c := shor_circuit a N in         (* use a to construct a circuit *)
  let rnd' := get_new_rnd rnd in
  let x := run c rnd' in               (* sample from the circuit's output *)
  factor a N (OF_post a N (fst k x) m) (* postprocessing *)
 else Some (gcd a N).

Definition end_to_end_shor (N : ℕ) (rnds : list R) :  option ℕ :=
 iterate rnds (shor_body N).
```

Figure 3.12: SQIR definition of Shor's algorithm (simplified for presentation). `uniform` creates a uniform distribution, `sample` samples from a distribution using a real number as a source of randomness, `run` samples from the distribution created by running a circuit, and `iterate` repeats a function until it runs out of randomness (i.e., `rnds` is empty) or some iteration succeeds (i.e., `shor_body` returns `Some`).

The implementation and proof comprise approximately 14k lines of code and took us roughly a year to complete.

[4]Work on `IMM` and RCIR was primarily completed by Yuxiang Peng.

Proof Sketch We prove three key properties of our implementation:

1. For any $N > 1$, if `end_to_end_shors` (or `shor_body`) returns `Some x` then x is a nontrivial factor of N.

2. For any N that is not prime, not even, and not a prime power, the probability that `shor_body` returns `Some x` is at least $\frac{\kappa}{\lfloor \log_2(N) \rfloor^4}$ where $\kappa = \frac{2e^{-2}}{\pi^2}$. Therefore, the probability that `end_to_end_shors` returns `None` (meaning that it failed to find a factor) is no greater than $(1 - \frac{\kappa}{\lfloor \log_2(N) \rfloor^4})^t$ where t is the number of iterations.

3. For any N, `shor_circuit` uses at most $(212n^2+975n+1031)m+4m+m^2$ gates, where n is to the number of bits representing N and m is the number of bits in QPE's output. m and n are $O(\log N)$, which leads to an overall $O(\log^3 N)$ asymptotic complexity that matches the original paper [Sho97].

Property (1) requires a proof that $\gcd(x, N)$ is a nontrivial factor of N when $1 < \gcd(x, N) < N$ and $x < N$. Property (3) is proved by analyzing the structure of `shor_circuit`. The only complexity is that `IMM` is constructed using RCIR and then compiled to SQIR, so we need to prove how gate counts in RCIR translate to gate counts in SQIR. Property (2) requires the most work.

First, we prove that the hybrid order finding procedure (which takes inputs a and N) returns $\mathrm{ord}(a, N)$ with probability at least $\frac{2\kappa}{\lfloor \log_2(N) \rfloor^4}$. This requires the general correctness property for QPE (presented in the last section), a proof of correctness for `IMM` (proved using RCIR's semantics), and a proof that continued fraction expansion produces the expected result. Second, we prove that for half of the possible choices of a, $\mathrm{ord}(a, N)$ can be used to find a nontrivial factor of N in postprocessing. Together, this means that the probability of successfully finding the order r of a modulo N, and using r to find a factor, is at least $\frac{2\kappa}{\lfloor \log_2(N) \rfloor^4} * \frac{1}{2}$. For more details about the proof see Peng et al. [Pen+22].

Running Certified Code Having completed our certified-in-Coq implementation of Shor's algorithm, we *extract* the program—both classical and quantum parts—to executable code. For the quantum part of Shor's (the order finding circuit), we use the extraction process described in Section 3.1.4. We extract the classical pre- and postprocessing directly to OCaml.

In principle, the generated order finding circuits could be executed on any quantum machine. However, even for small instances (e.g., factoring $N = 15$), the generated quantum circuits require 35 qubits and over 20k gates, which is well out of reach of current hardware. As an alternative, we ran the circuits using the DDSIM simulator [Cha21] on a laptop with an Intel Core i7-8705G CPU.[5]

Figure 3.13 shows the success probability and gate counts for order finding and factorization instances with input size ($n = \log(N)$) from 2 to 10 bits (i.e., $N \leq 1024$). Red circles mark instances (i.e., values of N) that can be simulated by DDSIM in under an hour. Blue intervals mark the empirical success probability, computed

[5]Experiments performed by Yuxiang Peng.

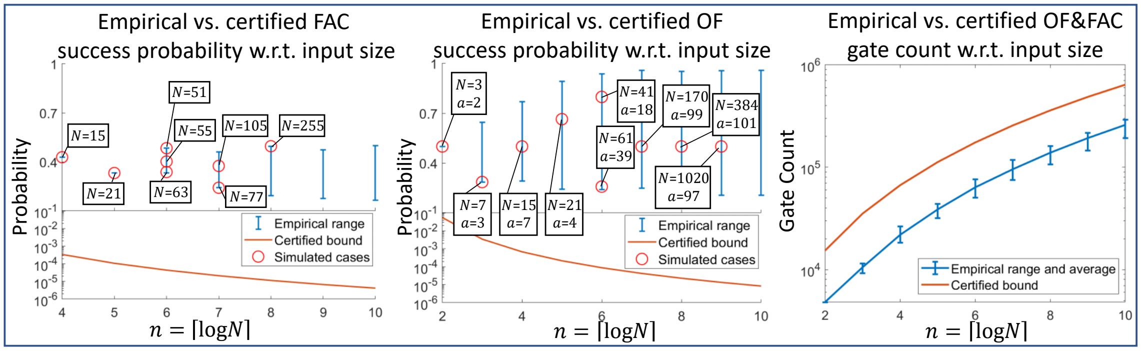


Figure 3.13: Success probability and gate counts for order finding (OF) and factorization (FAC) for every input N of size $n = 2$ to 10 bits. We draw the bounds certified in Coq as red curves. Whenever simulation is possible with DDSIM, we draw the observed results as red circles. Otherwise, we compute the corresponding bounds using analytical formulas; these are drawn as blue intervals.

according to Shor's original analysis [Sho97] for particular values of N. Red curves mark our certified bounds, which are a function of n (the size of N). We observed that (1) the certified bounds hold for all instances (2) the empirical bounds are considerably better than certified ones for the studied instances. The latter is likely due to the non-optimality of our proofs in Coq and the fact that we only investigated small instances.

We note that experimental demonstrations of Shor's algorithm already exist for small instances like $N = 15$ [Van+01; Lu+07; Lan+07; Luc+12; Mon+16], which use around 5 qubits and 20 gates. These experimental demonstrations are possible because they leverage quantum circuits that are specially designed for fixed inputs, but cannot extend to work in the general case.

Chapter 4

A Verified Optimizer for Quantum Circuits

Compilers are a key component 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*, which applies a series of optimizations to SQIR programs, ultimately producing a result that is compatible with a specified quantum architecture. We additionally present VQO, a *verified quantum oracle framework* that compiles oracle programs written in a high-level classical language into SQIR, allowing them to be optimized using VOQC.

At the core of VOQC is a framework for writing transformations of SQIR programs and verifying their correctness (Section 4.1). To ensure that the framework is suitably expressive, we have used it to develop verified versions of a variety of optimizations (Section 4.2). 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 4.3).

We evaluated the quality of the optimizations we verified in VOQC by measuring how well it optimizes a set of benchmark programs, compared to several other optimizing compilers (Section 4.4). The results are encouraging. On a benchmark of 35 circuit programs developed by Amy and Gheorghiu [AG20] we find that VOQC reduces total gate count on average by 28.5% compared to 14.4% for IBM's Qiskit compiler [Qis17], and 18.5% for CQC's t|ket⟩ [Siv+20]. On the same benchmarks, VOQC reduces T-gate count (an important measure when considering fault tolerance) on average by 43.1% compared to 45.4% by Amy and Gheorghiu [AG20] and 47.1% by the PyZX optimizer [Kv19a], although VOQC outperforms both in terms of total gate count reduction. In sum, VOQC is expressive enough to verify a range of useful optimizations, yielding performance competitive with leading unverified compilers.

In addition to VOQC, this chapter presents VQO, our framework to ease the construction of efficient, correct *oracle functions*, which are the classical subroutines used by many quantum algorithms (Section 4.5). The core of VQO is $\mathcal{O}$QASM, the *oracle quantum assembly language*. $\mathcal{O}$QASM operations move qubits between two different bases via the quantum Fourier transform, thus admitting important optimizations, but without inducing *entanglement* and the exponential blowup that comes with it. $\mathcal{O}$QASM's design enabled us to prove correct VQO's compilers—from a simple imperative language called $\mathcal{O}$QIMP to $\mathcal{O}$QASM, and from $\mathcal{O}$QASM to SQIR—and allowed us to efficiently test properties of $\mathcal{O}$QASM programs using the QuickChick property-based testing framework [Par+15]. We have used VQO to implement a variety of arithmetic and geometric operators that are building blocks for important oracles, including those used in Shor's and Grover's algorithms. We found that VQO's QFT-based arithmetic oracles require fewer qubits than those constructed using "classical" gates; VQO's versions of the latter were nevertheless on par with or better than (in terms of both qubit and gate counts) oracles produced by Quipper [Gre+13], a state-of-the-art unverified quantum programming platform.

VOQC is the first fully verified optimizer for general quantum programs and VQO is the first verified compiler for a high-level oracle language (Section 4.6). Amy, Roetteler, and Svore [ARS17] developed a verified optimizing compiler from classical Boolean expressions to reversible circuits and Fagan and Duncan [FD18] verified an optimizer for ZX-diagrams representing Clifford circuits; however, neither of these tools support general quantum programs. Tao et al. [Tao+22] developed Giallar, which uses symbolic execution and SMT solving to automatically verify circuit transformations in the Qiskit compiler; however, Giallar 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 van de Wetering [Kv19a], and Smith and Thornton [ST19] use equivalence checking to compare the semantics of a compiled circuit against its source. However, like any other translation validation technique, these approaches add compile-time overhead and may fail to identify latent bugs in the optimizer.

All code is freely available online:

- VOQC Coq definitions and proofs: `https://github.com/inQWIRE/SQIR`
- VOQC OCaml library: `https://github.com/inQWIRE/mlvoqc`
- VOQC Python bindings and tutorials: `https://github.com/inQWIRE/pyvoqc`

- VOQC benchmarking scripts: https://github.com/inQWIRE/VOQC-benchmarks
- VQO development: https://github.com/inQWIRE/VQO

Acknowledgements This chapter is primarily based on joint work with Robert Rand, Liyi Li, Shih-Han Hung, Xiaodi Wu, and Michael Hicks [Hie+21a]. 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 4.2.2). Section 4.5 describes joint work with Liyi Li, Finn Voichick, Yuxiang Peng, Xiaodi Wu, and Michael Hicks on VQO [Li+21]; for this project, I assisted with overall design and presentation and implemented extraction to OCaml for benchmarking.

4.1 A Verified Framework for Optimizing Quantum Programs

This section introduces general features of VOQC's design. We discuss specific optimizations in Section 4.2 and circuit mapping routines in Section 4.3.

4.1.1 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 4.2.1).
- Replacing a sub-program with an equivalent program (see Section 4.2.2).
- 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).

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:	$CNOT,\ SWAP$

4.1.2 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., $[\![U_1]\!]_d = [\![U_2]\!]_d$. We can then write our soundness condition for optimization function `optimize` as follows.

```
Definition sound {G} (optimize : ∀ {d : ℕ}, ucom G d → ucom G d) :=
  ∀ (d : ℕ) (u : ucom G d), ⟦optimize u⟧_d ≡ ⟦u⟧_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 $[\![U_1]\!]_d = e^{i\theta}[\![U_2]\!]_d$; We say that two circuits are *equivalent up to permutation* if there exist permutation matrices P_1, P_2 such that $[\![U_1]\!]_d = P_1 \times [\![U_2]\!]_d \times P_2$.[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 4.3) where inserted $SWAP$ gates may change the positions of qubits in the system.

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

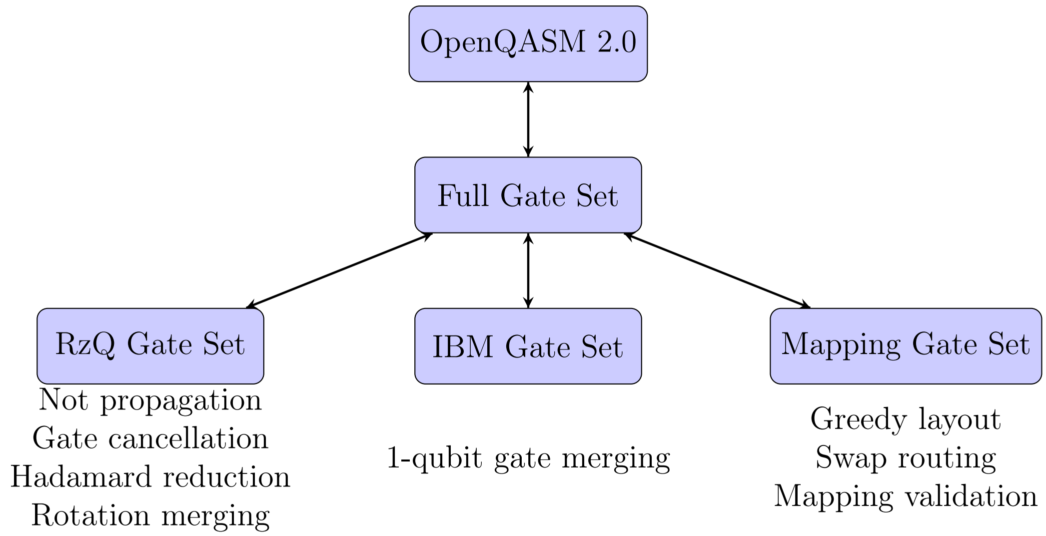


Figure 4.1: Summary of features available in VOQC

4.1.3 Supported Gate Sets

The VOQC framework supports arbitrary gate sets; the utility functions and properties described above are all parameterized by choice of gate set. However, the program transformations in Sections 4.2 and 4.3 are defined over the particular gate sets listed in Table 4.1. Using a custom gate set for a 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* is used for parsing and consists of a variety of standard quantum gates. It 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 optimizations are defined over different gate sets internally, 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 $\{H,\ X,\ R_zQ,\ CNOT\}$ where $R_zQ(q)$ describes rotation about the z-axis by $q\pi$ for $q \in \mathbb{Q}$. We use a rational parameter for the R_zQ 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

Table 4.2: 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 4.3. Note that *CNOT* is primitive in every gate set we support.

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

Table 4.3: Decompositions of single-qubit gates in the full gate set into gates in the RzQ and IBM gate sets. When needed, we perform implict coercion from real expressions (e.g., $\frac{r}{\pi}$) to rational numbers.

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

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 general[2], 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 4.2.2). 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 4.3). It is parameterized by a set 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 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.

4.1.4 Extraction to Executable Code

We use Coq's standard code extraction mechanism [Inr21b] 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. They may even enforce precision limitations internally: For example, Qiskit's `CommutativeCancellation` optimization pass [Qis21] uses a cutoff precision of 10^{-5}, below which rotations are treated as identities.

[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 $U_2(\phi, \lambda) = U_3(\frac{\pi}{2}, \phi, \lambda)$.

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

Figure 4.2: Equivalences used in not propagation

In order to make VOQC compatible with existing Python-based frameworks for compiling quantum programs (e.g., Qiskit [Qis17], Sivarajah2020 [Siv+20], Quilc [Rig19b], Cirq [Cir18]), 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 [Inr21a]) 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). We use the VOQC Qiskit pass in our evaluation in Section 4.4.2.

4.2 Optimizations

VOQC primarily implements optimizations inspired by the state-of-the-art circuit optimizer by 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.

4.2.1 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

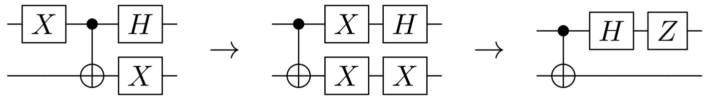


Figure 4.3: An example of not propagation. In the first step the leftmost X gate propagates through the $CNOT$ gate to become two X gates. In the second step the upper X gate propagates through the H gate and the lower X gates cancel.

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 supports the three-qubit $CCNOT$ gate; this gate can be decomposed into a $\{H, R_zQ, CNOT\}$ program per Table 4.2. 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 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 $R_zQ(k + k')$.

4.2.2 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

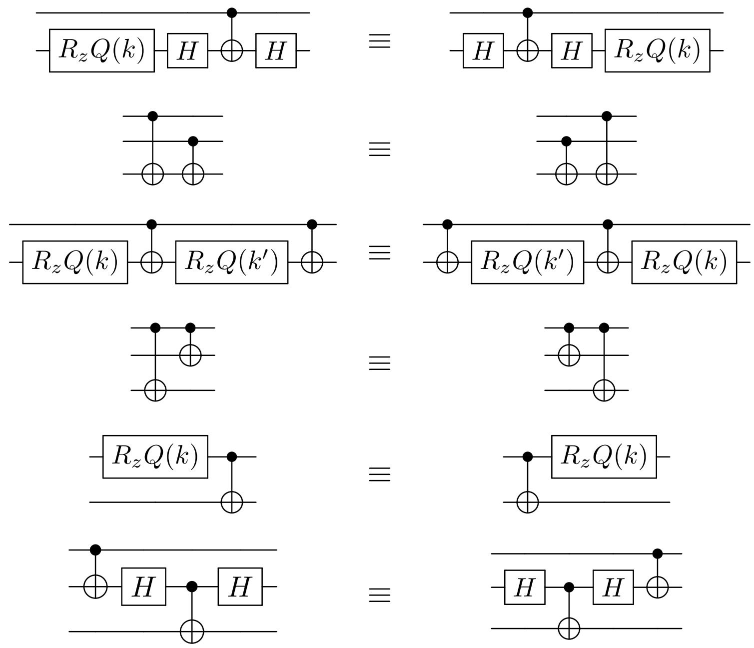


Figure 4.4: Commutation equivalences for single- and two-qubit gates adapted from Nam et al. [Nam+18, Figure 5]. We use the second and third rules for propagating both single- and two-qubit gates.

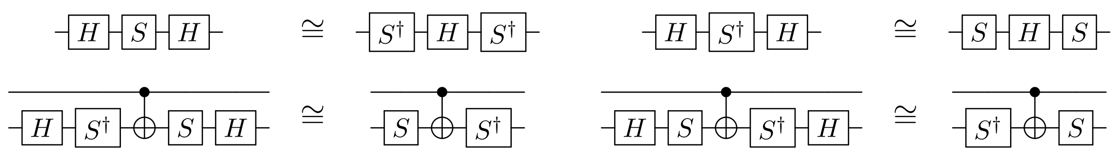


Figure 4.5: Equivalences for removing Hadamard gates adapted from Nam et al. [Nam+18, Figure 4]. S is the phase gate $R_zQ(\frac{1}{2})$ and $S^\dagger$ is its inverse $R_zQ(\frac{3}{2})$.

reduce the gate count directly, or to set the stage for additional optimizations.

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.

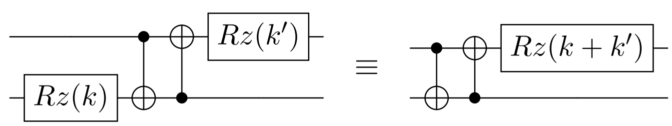
The basic idea behind rotation merging is to (i) identify subcircuits consisting of only *CNOT* and R_zQ gates and (ii) merge R_zQ gates within those subcircuits that are applied to qubits in the same logical state. The argument for the correctness of this optimization relies on the *phase polynomial* representation of a circuit. Let C be a circuit consisting of *CNOT* gates and rotations about the z-axis. Then on basis state $|x_1, ..., x_n\rangle$ for $x_i \in \{0, 1\}$, C will produce the state

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

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

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

is a linear combination of affine boolean functions. $p(x_1, ..., x_n)$ 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 $f_i(x_1, ..., x_n) = f_j(x_1, ..., x_n)$ 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 $e^{ik\pi x} |x\rangle$ 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:

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

Whereas evaluation of the right-hand circuit produces

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

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 $\{R_zQ$, $CNOT\}$ 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_zQ rotations when they are applied to qubits associated with the same Boolean function. To prove equivalence over $\{R_zQ, CNOT\}$ circuits, we show that the original and optimized circuits produce the same output on every basis state. We

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

Figure 4.6: Rules for single-qubit gate merging

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 $\{R_zQ,\ CNOT\}$ subcircuits differs. We construct a $\{R_zQ, CNOT\}$ subcircuit beginning from a R_zQ gate whereas Nam et al. begin from a $CNOT$ gate. The result of this simplification is that our subcircuits may be smaller than Nam et al.'s, causing us to 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: $U_3(\theta, \phi, \lambda) \longrightarrow R_z(\phi) \cdot R_y(\theta) \cdot R_z(\lambda)$. 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 $\alpha,\ \beta,\ \gamma$ 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 $\alpha,\ \beta,\ \gamma$ can be generated using arithmetic over trignometric 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.

```
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 ξ θ2 ξ1 θ ξ2,
   yzy_to_zyz θ1 ξ 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

4.2.3 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 4.4), we observed that single-qubit gate cancellation and rotation merging were the most effective at reducing gate count.

We (optionally) conclude by converting 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.

4.2.4 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 [Qis17]: 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` $q\ P_1\ P_2$ 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, $\{\!|P_1|\!\}_d(\rho) = \{\!|P_2|\!\}_d(\rho)$.

z-rotations Before Measurement

z-axis rotations (or, more generally, diagonal unitary operations) before a measurement have no effect on the measurement outcome, so they can safely be removed from the program. This optimization locates and removes Rz gates before measurement operations. 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 and if $i = 1$ then $CNOT\ q\ q'$ becomes $X\ q'$.
- `meas` $q\ P_1\ P_0$ becomes P_i.
- $H\ q$ and $CNOT\ q'\ q$, where q' is not known to be classical, 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.

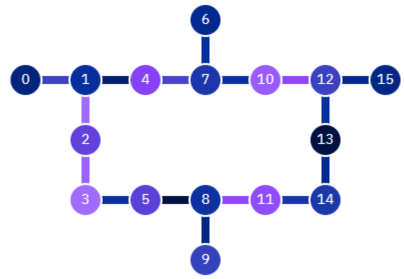


Figure 4.8: IBM's 16-qubit Guadalupe machine [IBM22]. The different colors on nodes and connections reflect different error rates (darker means lower error and lighter means higher error).

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.

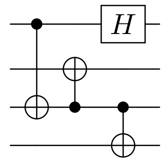
4.3 Circuit Mapping

While optimization aims to reduce qubit and gate usage to make programs more feasible to run on near-term machines, *circuit mapping* addresses the connectivity constraints of near-term machines, transforming a program so that it is able to run on a machine [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 IBMs'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 [Qis17] and t$|$ket$\rangle$ [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.

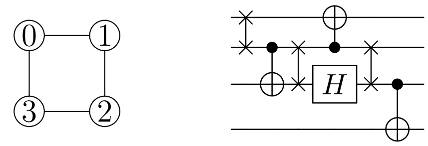
4.3.1 Verified Layout

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. Greedy layout scans through the program and allocates



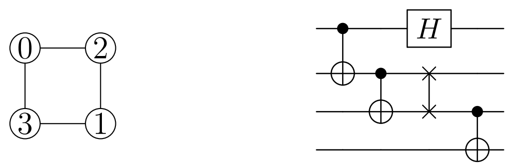
P1 = CNOT 0 2; CNOT 2 1; H 0;
CNOT 2 3

(a) Input program



P2 = SWAP 0 2; CNOT 1 2; SWAP 1 2; CNOT 1 0;
H 2; SWAP 1 2; CNOT 2 3

(b) Trivial layout



P3 = CNOT 0 1; CNOT 1 2; H 0; SWAP 1 2; CNOT 2 3

(c) Greedy layout

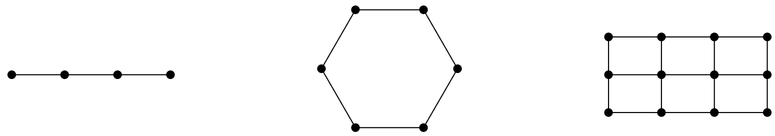
Figure 4.9: Circuit mapping example

logical qubits to adjacent physical qubits when they are used together in a *CNOT* gate, until all logical qubits are allocated. 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 l to a layout where physical qubit i maps to logial qubit $l[i]$; 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 4.3.3.

4.3.2 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



(a) (b) (c)

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 vertices, but in our implementation the number of vertices is a parameter.

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 *SWAP*s 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 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.

4.3.3 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* [PSS98] function to check the correctness of circuit mapping on particular inputs. Unlike our verified optimizations and mapping routines, the translation validator may fail at runtime, indicating a potential bug in the circuit mapper. However, our proofs guarantee that if translation validation succeeds, then 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⟩).[3] It will not be able to validate

[3] t|ket⟩ may choose to compile a distance-2 *CNOT* gate to a distributed *CNOT*, rather than a

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. For example, 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 4.2.2.

Translation validation is popular for providing assurance for quantum compilers: Amy [Amy19b] checks for equivalence of optimized and unoptimized programs using the path-sums semantics; PyZX [Kv19a] 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] present 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 circuits that use tens of thousands of operations. However, none of these other tools has been formally verified, and all aside from Burgholzer, Raymond, and Wille [BRW20] are significantly more expensive 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.

4.3.4 Mapping with Optimization

Circuit mapping increases the size of the program, typically adding many *CNOT* gates to perform *SWAP*s 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 4.2 preserve connectivity guarantees, allowing us to apply optimization before and 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:

$$SWAP\ a\ b\ =\ CNOT\ a\ b;\ CNOT\ b\ a;\ CNOT\ a\ b$$

and

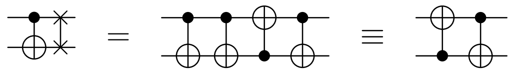
$$SWAP\ a\ b\ =\ CNOT\ b\ a;\ CNOT\ a\ b;\ CNOT\ b\ a.$$

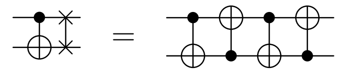
We choose the decomposition that will allow *CNOT* gates to be removed during gate cancellation (Section 4.2.1). For example, the subcircuit in Figure 4.9(b) and Figure 4.9(c) that consists of a *CNOT* followed by *SWAP* will be decomposed as

---

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 the *CNOT* gates corresponding to swaps from the mapped circuits.

shown on the left below, rather than the right, since this will save two gates in the optimized form.





4.4 Experimental Evaluation

The value of VOQC (and SQIR) is determined by the quality of the verified optimizations we can write with it. We can judge optimization quality empirically. In particular, we can run VOQC on a benchmark of circuit programs and see how well it optimizes those programs, compared to (non-verified) state-of-the-art compilers.

To this end, in Section 4.4.1, we compare the performance of VOQC's verified optimizations against IBM's Qiskit compiler [Qis17], CQC's t|ket⟩ [Siv+20], PyZX [Kv19a], and Staq [AG20] on a set of benchmarks developed for Staq. We find that VOQC has comparable performance to all of these: it generally beats these tools in terms of total gate count reduction, and often matches reduction of T gate count. In Section 4.4.2, we evaluate VOQC's optimization and mapping routines by running them via a pass in the Qiskit transpiler on a set of benchmarks used to evaluate prior work on mapping algorithms [ZPW17; WBZ19]. Compared to Qiskit's default settings, we find that VOQC's optimizations provide an advantage, even for mapped circuits, and our verified translation validation does not add undue overhead. Finally, in Section 4.4.3, we provide a detailed comparison of the performance of VOQC and Nam et al., showing that our verified implementation is mostly faithful to its inspiration.

Note that the aim of this section is not to claim superiority over existing tools (after all, we have implemented a subset of the optimizations available in Nam et al. [Nam+18] and Qiskit), but to demonstrate that the optimizations we have implemented in VOQC are on par with existing *unverified* tools.

4.4.1 Evaluation on Staq Benchmarks

We begin by comparing VOQC, Qiskit, t|ket⟩, PyZX, and Staq on a benchmark developed for Staq [AG20]. This benchmark consists of 35 programs written in the "Clifford+T" gate set ($CNOT$, H, S and T, where S and T are z-axis rotations by $\pi/2$ and $\pi/4$, respectively). The benchmark programs contain arithmetic circuits, implementations of multiple-control X gates, Galois field multiplier circuits, and some small quantum algorithm components. We exclude programs with more than 10^4 gates, following the precedent of prior work [Siv+20, Section 9.1.1].

We measure reduction in total gate count, two-qubit gate count, and T-gate count. Total gate count and two-qubit gate count are useful metrics for near-term quantum computing, where the length of the computation must be minimized to reduce error and two-qubit gates have higher error rates than single-qubit gates. T-gate count is relevant in the *fault-tolerant regime* where qubits are encoded using quantum error correcting codes and operations are performed fault-tolerantly. In this regime, the standard method for making Clifford+T circuits fault tolerant produces particularly

Table 4.4: Summary of optimizations used in Staq evaluation

`qiskit-terra` 0.19.1	
Optimize1qGatesDecomposition	✓
CommutativeCancellation	✓*
ConsolidateBlocks w/ UnitarySynthesis	
`pytket` 0.19.2	
RemoveRedundancies	✓
FullPeepholeOptimise	
`pystaq` 2.1	
simplify	✓
rotation_fold	✓*
cnot_resynth	
`pyzx` 0.7.0	
full_optimize	✓*
full_reduce	

expensive translations for T gates, so reducing T-count is a common optimization goal. The Clifford+T set is a subset of VOQC's RzQ gate set where each z-axis rotation is restricted to be a multiple of $\pi/4$.

Baselines We compare VOQC's performance with that of Qiskit Terra version 0.19.1 (release date December 10, 2021), t|ket⟩ version 0.19.2 (February 18, 2022), Staq version 2.1 (January 17, 2022), and PyZX version 0.7.0 (February 19, 2022). When comparing against Qiskit and t|ket⟩, we use VOQC's IBM gate set. When comparing against Staq and PyZX, we use the RzQ gate set.

Table 4.4 lists the optimizations we include in our evaluation. For every tool except t|ket⟩, we evaluate all available (unitary) optimizations; we exclude t|ket⟩'s OptimisePhaseGadgets and PauliSimp as they hurt improve performance on our benchmarks. VOQC provides the complete and verified functionality of the routines marked with ✓; we write ✓* to indicate that VOQC contains a verified optimization with similar, although not identical, behavior. VOQC's gate cancellation routines generalize Qiskit's Optimize1qGatesDecomposition, t|ket⟩'s RemoveRedundancies, and Staq's simplify; VOQC's gate cancellation is also similar to Qiskit's CommutativeCancellation, but Qiskit uses matrix multiplication to determine whether gates commute while we use a rule-based approach. VOQC's rotation merging is similar to Staq's rotation_fold and (when combined with gate cancellation) PyZX's full_optimize.

As far as the optimizations VOQC does not support: Qiskit's UnitarySynthesis and t|ket⟩'s FullPeepholeOptimize resynthesize two-qubit gate sequences (e.g., using KAK decomposition [VW04]), Staq's cnot_synthesis resynthesizes arbitrary $\{CNOT, X, Rz\}$ subcircuits (as discussed in Section 4.3), and PyZX's full_reduce applies the ZX-calculus rewrite rules described in Kissinger and van de Wetering [Kv19b].

Table 4.5: Geometric mean gate count reduction on the Staq benchmarks using the IBM gate set. The full results are presented in Appendix A.

	Qiskit	t|ket⟩	VOQC
Total gate count	14.4%	18.5%	28.5%
Two-qubit gate count	2.3%	3.8%	9.8%

Table 4.6: Geometric mean gate count reduction on the Staq benchmarks using the RzQ gate set. The negative values for PyZX indicate that it *increases* the gate count. The full results are presented in Appendix A.

	Staq	PyZX	VOQC
Total gate count	15.4%	-27.8%	23.2%
Two-qubit gate count	0.6%	-91.7%	9.8%
T-gate count	45.4%	47.1%	43.1%

Results The results are summarized in Tables 4.5 and 4.6; the full results are given in Appendix A. Overall, VOQC is the most effective at reducing total and two-qubit gate count for both the IBM and RzQ gate sets, while it is slightly less effective than Staq and PyZX at eliminating T gates.

On all 35/35 programs, VOQC is more effective at reducing total gate count than Qiskit. On 33/35 programs it is more effective at reducing total gate count than t|ket⟩. This gap in performance is primarily due to VOQC's rotation merging optimization, which has no analogue in Qiskit or t|ket⟩. On 9/35 programs, Qiskit, t|ket⟩, and VOQC provide no reduction in two-qubit gate count, suggesting that these tools are not particularly effective for this type of gate. Of the remaining 23/35 programs, VOQC provides a higher reduction in two-qubit gate count, despite the fact that Qiskit and t|ket⟩ both support two-qubit circuit resynthesis, which should give them an advantage. This suggests that small circuit resynthesis optimizations are not particularly effective for this class of circuits.

VOQC is more effective than (or equally effective as) Staq and PyZX at reducing total gate count on 29/35 programs. On 4/35 programs, Staq is the most effective, and on the remaining 2/35 PyZX performs best. However, on 26/35 programs PyZX actually *increases* the total gate count.[4] The reason for this is that PyZX converts a circuit to a ZX-diagram, performs optimization, and then converts back; this conversion process can introduce many additional gates. However, PyZX is the best overall at reducing T-gate count. On 28/35 programs, PyZX is more effective than (or equally effective as) Staq and VOQC at reducing T-gate count. On 7/35 programs, Staq performs best. Surprisingly, on 13/35 benchmarks, all optimizers produce the same T-count. This is unexpected since, although all these optimizers rely on some form of rotation merging, their implementations differ substantially. Kissinger and

[4]To account for this, when computing the "geometric mean reduction" for each type of gate, we actually compute the geometric mean *proportion* of the final to original gate count, which is a strictly positive number. We then subtract this number from 1, which produces a negative value in cases where gate count increased on average.

Table 4.7: Geometric mean running times

Qiskit	t|ket⟩	Staq	PyZX	VOQC
0.60s	1.53s	0.02s	14.58s	0.01s

van de Wetering [Kv19b] posit that this indicates a local optimum in the ancilla-free case for some of the benchmarks (in particular the tof benchmarks, whose T-count is not reduced by applying additional techniques [HT18]). As with the IBM gate set experiments, VOQC is not particularly effective at reducing two-qubit gate count. However, unlike PyZX and Staq, it never increases the number of two-qubit gates, which leads to significantly better performance overall.

To compare the running times of the different tools, we ran 11 trials of VOQC, Qiskit, t|ket⟩, Staq, and PyZX (taking the median time for each benchmark) on a standard laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Monterey. We show the geometric mean running times over all 35 benchmarks in Table 4.7. Qiskit, Staq, and VOQC all have mean running times of under one second. t|ket⟩ is slightly slower, but the mean is still under two seconds. PyZX was consistently the slowest; the full_optimize routine in particular scaled poorly with increasing qubit and gate counts. We set a time limit of 10 minutes for each run of PyZX, defaulting to applying full_reduce (but not full_optimize) in case of a timeout. In the case of a timeout, we consider only the time required to run full_reduce. In addition, full_optimize is not deterministic, and may produce different output circuits for different runs. The gate counts we report are from the trial that produced the lowest T-gate count.

Overall, these results are encouraging evidence that VOQC supports useful and interesting verified optimizations. Furthermore, despite having been written with verification in mind, VOQC's running times are not significantly worse than (and often better than) that of current tools.

4.4.2 Evaluation of Mapping Validation

To evaluate VOQC's support for circuit mapping, we used the test suite from the MQT quantum circuit mapping tool [Cha22], subsets of which have been used in the evaluations of Zulehner, Paler, and Wille [ZPW17] and Wille, Burgholzer, and Zulehner [WBZ19]. The benchmark programs all use at most 16 qubits, and, like before, we include only those circuits with less than 10^4 gates. The circuits are primarily taken from the RevLib [Wil+08] suite of classical reversible circuits, but the benchmark also includes some truly quantum circuits including quantum Fourier transforms and circuits for quantum chemistry. In total, we consider 126 circuit programs.

Baselines We developed a custom pass for the Qiskit compiler that applies the following sequence of transformations: VOQC optimization (using the RzQ gate set), Qiskit circuit mapping, VOQC mapping validation, and VOQC optimization (using

Table 4.8: Geometric mean gate count increases and running times for the mapping experiment over all 126 benchmark circuits.

	Qiskit	**VOQC**	**Qiskit+VOQC**
Total gate count increase	80.9%	73.9%	38.9%
Two-qubit gate count increase	113.5%	204.1%	126.0%
Running time	1.07s	<0.01s	1.01s

the IBM gate set). Our verified translation validation allows us to use the sophisticated mapping routines available in Qiskit without sacrificing soundness. We can run VOQC optimizations both before and after mapping because we have proved that our optimizations preserve mapping guarantees.

We compare our pass against the default Qiskit pass with optimization level 3, which applies mapping followed by optimization. For both passes, we use Qiskit's default layout and routing routines which are based on Li, Ding, and Xie [LDX19]. This routing algorithm is non-deterministic, so we run multiple (11) trials, storing the result of the run that produced the circuit with the lowest total gate count. We additionally compare against an OCaml program that applies VOQC optimization (using the RzQ gate set), greedy layout and swap routing (Section 4.3), and optimization (using the IBM gate set). We record change in total gate count and two-qubit gate count. Note that unlike Section 4.4.1, where we expected gate count reduction, for this experiment we expect gate count to *increase* since mapping introduces many additional CX gates, some of which will be removed by optimization. All circuits are mapped to a 16-qubit ring architecture (Figure 4.10(b)). Like before, we use Qiskit Terra version 0.19.1 and run on a standard laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Monterey. When recording timing data, we take the median running time over 11 trials.

Results The results are summarized in Table 4.8. The first column shows the result of the Qiskit default pass, the second column shows the result of the VOQC OCaml program, and the last column shows the result of the VOQC pass within Qiskit. Compared to the Python-based Qiskit passes, the OCaml program is significantly faster; however, it also introduces the most two-qubit gates due to its simple $SWAP$-insertion strategy. Despite this, the OCaml program results in a lower total gate count overhead than Qiskit—this can be explained by VOQC's superior optimizations, as per Section 4.4.1. Overall, Qiskit's default pass provides the lowest overhead in two-qubit gate count. This is due to Qiskit's two-qubit circuit resynthesis optimization, which is especially effective at reducing two-qubit gate count post-mapping. The Qiskit+VOQC pass has the best performance overall: it has running time and two-qubit gate count overhead comparable with Qiskit, but provides a much lower total gate count overhead and *proven guarantees* that the output circuit is semantically equivalent to the input. During our trials, we found that translation validation accounted for <1% of the Qiskit+VOQC pass running time and we never encountered a validation failure, which means that Qiskit's mapping routines assuredly preserved the semantics of the programs we tested (although this says nothing about semantics-

Table 4.9: Summary of optimizations available in Nam et al. [Nam+18]

Nam et al.	
Not propagation (P)	✓*
Hadamard gate reduction (L, H)	✓
Single-qubit gate cancellation (L, H)	✓
Two-qubit gate cancellation (L, H)	✓
Rotation merging using phase polynomials (L)	✓*
Floating R_z gates (H)	
Special-purpose optimizations (L, H)	
• LCR optimizer	✓
• Toffoli decomposition	

preservation of Qiskit's optimizations).

4.4.3 Evaluation on Nam Benchmarks

In this section, we evaluate VOQC's performance on all 99 benchmark programs considered by Nam et al. [Nam+18], confirming that VOQC is a faithful implementation of a subset of the optimizations present in Nam et al. (along with being proved correct!). The benchmarks are divided into three categories, as described below. Our versions of the benchmarks are available online.[5]

We summarize the optimizations available in Nam et al. in Table 4.9. P stands for "preprocessing" and L and H indicate whether the routine is in the "light" or "heavy" versions of the optimizer. VOQC provides the complete and verified functionality of the routines marked with ✓; we write ✓* to indicate that VOQC contains a verified optimization with similar, although not identical, behavior. We have not yet implemented "Toffoli decomposition" and "Floating R_z gates" and compared to Nam et al.'s rotation merging, VOQC performs a slightly less powerful optimization (as discussed in Section 4.2.2).

In cases where Toffoli decomposition and heavy optimization are not used (the QFT, QFT-based adder, and product formula circuits), VOQC's results are identical to Nam et al.'s. In the other cases, VOQC is slightly less effective. In the worst case, VOQC's running time is four orders of magnitude worse than Nam et al.'s. However, VOQC's running time is often less than a second. We view this performance as acceptable, given that benchmarks with more than 1000 two-qubit gates (the only programs for which VOQC optimization takes longer than one second) are well out of reach of current quantum hardware [Pre18]. We are confident that VOQC's performance can be improved through more careful engineering.

Arithmetic and Toffoli These benchmarks overlap with the Staq benchmarks—both originate from an earlier paper by Amy, Maslov, and Mosca [AMM13]. The programs range from 45 to 346,533 gates and 5 to 489 qubits. The total gate count reduction and timing results for all 32 benchmarks are given in Appendix A. In

[5] https://github.com/inQWIRE/VOQC-benchmarks

sum: in 12 out of 32 cases VOQC outperforms Nam et al., but VOQC has a lower average reduction. This is primarily due to Nam et al.'s "special-purpose Toffoli decomposition", which affects how CCX gates are decomposed. Their decomposition enables rotation merging and single-qubit gate cancellation to cancel two gates (e.g. cancel T and $T^\dagger$) where we instead combine two gates into one (e.g. T and T becomes S). Interestingly, the cases where VOQC outperforms Nam et al. can also be attributed to their Toffoli decomposition heuristic, which sometimes result in fewer cancellations than the naïve decomposition that we use. We do not expect adding and verifying this form of Toffoli decomposition to pose a challenge in VOQC. Reduction in T-gate count is not shown, but VOQC matches Nam et al. (both L and H) on all benchmarks but two. The first case (qcla_adder_10) is due to our simplification in rotation merging. In the second case (qcla_mod_7), Nam et al.'s optimized circuit was later found to be inequivalent to the original circuit [Kv19b, Section 2], so the lower T-count is spurious.

QFT and Adders These benchmarks consist of components of Shor's integer factoring algorithm, in particular the quantum Fourier transform (QFT) and integer adders. Two types of adders are considered: an in-place modulo $2q$ adder implemented in the Quipper library and an in-place adder based on the QFT. These benchmarks range from 148 to 381,806 gates and 8 to 4096 qubits. Results on all 27 benchmarks are given in Appendix A. The Quipper adder programs use similar gates to the arithmetic and Toffoli circuits, so the results are similar—VOQC is close to Nam et al., but under-performs due to our simplified Toffoli decomposition. The QFT circuits use rotations parameterized by $\pi/2^n$ for varying $n \in \mathbb{N}$ (and no Toffoli gates) so VOQC's results are identical to Nam et al.'s. For consistency with Nam et al., on the QFT and QFT-based adder circuits we run a simplified version of our optimizer that does not include rotation merging.

Product Formula These benchmarks implement product formula algorithms for simulating Hamiltonian dynamics. The benchmarks range from 260 to 127,500 gates and 10 to 100 qubits; they use rotations parameterized by floating point numbers, which we convert to OCaml rationals at parse time. The product formula circuits are intended to be repeated for a fixed number of iterations, and our resource estimates account for this. VOQC applies Nam et al.'s "LCR" optimization routine to optimize programs across loop iterations. On all 40 product formula benchmarks, our results are the same as those reported by Nam et al. [Nam+18, Table 3]. H gate reductions range from 62.5% to 75%. Reductions in Clifford z-axis rotations (i.e. rotations by multiples of $\pi/2$) range from 75% to 87.5% while reductions in non-Clifford z-axis rotations range from 0% to 28.6%. $CNOT$ gate reductions range from 0% to 33%. Runtimes range from 0.01s for parsing and optimizing to 610.46s for parsing and 406.93s for optimizing. By comparison, Nam et al.'s running times range from 0.004s to 0.137s.

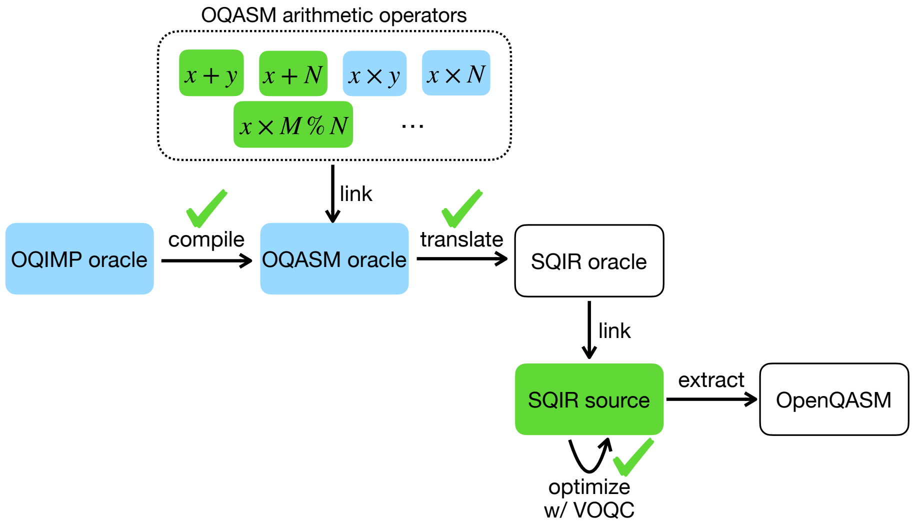


Figure 4.11: Overview of VQO toolchain

4.5 Compiling Oracle Programs

VOQC contains verified implementations of state-of-the-art optimizations and circuit mapping techniques, but it is a *transpiler*, rather than a *compiler*, since it converts SQIR to SQIR—it does not compile from a high-level source to a low-level target. A natural next step towards our goal of providing a high-assurance software toolchain for quantum programming is to develop a compiler from higher-level languages to SQIR, the output of which could then be optimized using VOQC. Our first step in this direction is VQO: a *verified quantum oracle* framework that helps programmers write *correct* and *efficient* quantum oracles programs.[6]

As discussed in Section 3.3, Grover's search algorithm [Gro96] can query unstructured data in sub-linear time (compared to linear time on a classical computer), and Shor's algorithm [Sho97] can factorize a number in polynomial time (compared to the sub-exponential time for the best known classical algorithm). An important source of speedups in these algorithms are the quantum computer's ability to apply an *oracle function* coherently, i.e., to a *superposition* of classical queries, thus carrying out in one step a function that would potentially take many steps on a classical computer. For Grover's, the oracle is a predicate function that determines when the searched-for data is found. For Shor's, it is a classical modular exponentiation function; the algorithm finds the period of this function where the modulus is the number being factored.

While the classical oracle function is perhaps the least interesting part of a quantum algorithm, it contributes a significant fraction of the final program's compiled quantum circuit. For example, Gidney and Ekerå [GE21] estimated that Shor's modular exponentiation function constitutes 90% of the final code. In our own experiments with Grover's, our oracle makes up over 99% of the total gate count (the oracle has 3.3 million gates).

[6]The VQO project was spearheaded by Liyi Li.

To aid programmers in writing correct and efficient quantum oracles, we deveoped VQO. Figure 4.11 summarizes the VQO toolchain; it marks verified components in green and tested components in blue.

- Using VQO, an oracle can be specified in a simple, high-level programming language we call $\mathcal{O}$QIMP, which has standard imperative features and can express arbitrary classical programs. It distinguishes quantum variables from classical parameters, allowing the latter to be *partially evaluated* [JGS93].

- The resulting $\mathcal{O}$QIMP program is compiled to $\mathcal{O}$QASM (pronounced "O-chasm"), the *oracle quantum assembly language*. $\mathcal{O}$QASM was designed to be efficiently simulatable while nevertheless admitting important optimizations. The generated $\mathcal{O}$QASM code links against implementations of standard operators (addition, multiplication, sine, cosine, etc.) also written in $\mathcal{O}$QASM.

- The $\mathcal{O}$QASM oracle is then translated to SQIR. After linking the oracle with the quantum program that uses it, the complete SQIR program can be optimized with VOQC and extracted to OpenQASM 2.0 [Cro+17] to run on a real quantum machine. Both VQO's compilation from $\mathcal{O}$QIMP to $\mathcal{O}$QASM and translation from $\mathcal{O}$QASM to SQIR have been proved correct in Coq, ensuring that properties proved of the $\mathcal{O}$QIMP source hold of the optimized SQIR output as well.

In this section, we summarize the design of $\mathcal{O}$QASM and $\mathcal{O}$QIMP and highlight key evaluation results. See Li et al. [Li+21] for more details.

4.5.1 $\mathcal{O}$QASM: An Assembly Language for Quantum Oracles

Because oracles are classical functions, a reasonable approach would have been to design $\mathcal{O}$QASM to be a circuit language comprised of "classical" gates; e.g., prior work has targeted gates X ("not"), $CNOT$ ("controlled not"), and $CCNOT$ ("controlled controlled not," aka *Toffoli*). Doing so would simplify proofs of correctness and support efficient testing by simulation because an oracle's behavior could be completely characterized by its behavior on computational basis states (essentially, classical bitstrings). ReverC [ARS17] and *Re$\mathcal{Q}$*WIRE [Ran+19] take this approach; we used a similar approach when developing RCIR in Section 3.3.3.

However, this approach cannot support optimized oracle implementations that use fundamentally quantum functionality, e.g., as in *quantum Fourier transform* (QFT)-based arithmetic circuits [Bea03; Dra00]. These circuits employ quantum-native operations (e.g., controlled-phase operations) in the *QFT basis*. Our key insight is that expressing such optimizations does not require expressing all quantum programs, as is possible in a language like SQIR. Instead, $\mathcal{O}$QASM's type system restricts programs to those that admit important optimizations while keeping simulation tractable.

$\mathcal{O}$QASM States An $\mathcal{O}$QASM program state φ of d qubits is a length-d tuple of qubit values q; the state models the tensor product of those values. This means that the size of φ is $O(d)$ where d is the number of qubits. A d-qubit state in SQIR is

$$
\begin{array}{llll}
\text{Position} & p & ::= & (x,n) \quad \text{Nat. } n \quad \text{Variable } x \\
\text{Instruction} & \iota & ::= & \texttt{ID}\ p \mid \texttt{X}\ p \mid \iota \,;\, \iota \\
 & & \mid & \texttt{SR}^{[-1]}\ n\ x \mid \texttt{QFT}^{[-1]}\ n\ x \mid \texttt{CU}\ p\ \iota \\
 & & \mid & \texttt{Lshift}\ x \mid \texttt{Rshift}\ x \mid \texttt{Rev}\ x
\end{array}
$$

Figure 4.12: $\mathcal{O}$QASM syntax

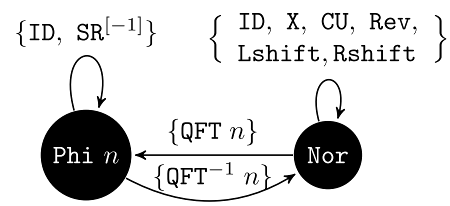


Figure 4.13: State machine

represented as a length 2^d vector of complex numbers, which is $O(2^d)$ in the number of qubits. $\mathcal{O}$QASM's linear state representation is possible because applying any well-typed $\mathcal{O}$QASM program on any well-formed state never causes qubits to be entangled.

A qubit value q has one of two forms, scaled by a global phase $\alpha(r)$. The two forms depend on the *basis* τ that the qubit is in—it could be either `Nor` or `Phi`. A `Nor` qubit has form $|b\rangle$ (where $b \in \{0,1\}$), which is a computational basis value. A `Phi` qubit has form $|\Phi(r)\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \alpha(r)\,|1\rangle)$, which is a value of the QFT basis.

Syntax Figure 4.12 presents $\mathcal{O}$QASM's syntax. An $\mathcal{O}$QASM program consists of a sequence of instructions ι. Each instruction applies an operator to either a variable x, which represents a group of qubits, or a *position* p, which identifies a particular offset into a variable x.

The instructions in the first row correspond to simple single-qubit quantum gates—`ID` p and `X` p—and instruction sequencing. The instructions in the next row apply to whole variables: `QFT` n x applies the approximate QFT (AQFT) to variable x with n-bit precision and $\texttt{QFT}^{-1}$ n x applies its inverse. If n is equal to the size of x ($|x|$), then the AQFT operation is the standard QFT from Section 3.3.2; otherwise, it drops the `RZ` gates applied to the lowest $|x| - n$ qubits. $\texttt{SR}^{[-1]}$ n x applies a series of `RZ` gates. Operation `CU` p ι applies instruction ι *controlled* on qubit position p. All of the operations in this row—`SR`, `QFT`, and `CU`—will be translated to multiple SQIR gates.

In the last row, instructions `Lshift` x, `Rshift` x, and `Rev` x are *position shifting operations*. Assuming that x has d qubits and x_k represents the k-th qubit state in x, `Lshift` x changes the k-th qubit state to $x_{(k+1)\%d}$, `Rshift` x changes it to $x_{(k+d-1)\%d}$, and `Rev` changes it to x_{d-1-k}. In our implementation, shifting is *virtual* not physical. The $\mathcal{O}$QASM to SQIR translator maintains a logical map of variables/positions to concrete qubits, ensuring that shifting operations introduce no extra gates.

Typing The type system (presented in Li et al. [Li+21, Section 3]) enforces two key invariants. First, it enforces that instructions are well-formed, meaning that gates are applied to valid qubit positions and that any control qubit is distinct from the target(s). These requirements are similar to SQIR's notion of well-typedness.

Second, the type system enforces that instructions leave affected qubits in a proper basis (thereby avoiding entanglement). The rules implement the state machine shown in Figure 4.13. For example, `QFT` n transforms a variable from `Nor` to `Phi` n, while $\texttt{QFT}^{-1}$ n transforms it from `Phi` n back to `Nor`. Position shifting operations are disallowed on variables x in the `Phi` basis because the qubits that make up x are internally related and cannot be rearranged. Indeed, applying a `Lshift` and then a $\texttt{QFT}^{-1}$ on x in `Phi` would entangle x's qubits.

Translation from $\mathcal{O}$QASM to SQIR VQO translates $\mathcal{O}$QASM to SQIR by mapping $\mathcal{O}$QASM positions to SQIR concrete qubit indices and expanding $\mathcal{O}$QASM instructions to sequences of SQIR gates. We have proved $\mathcal{O}$QASM-to-SQIR translation correct. To extract SQIR to OpenQASM 2.0, we use the approach described in Section 3.1.4.

Testing Leveraging $\mathcal{O}$QASM's efficient simulatability, we implemented a *property-based random testing* (PBT) framework for $\mathcal{O}$QASM programs in QuickChick [Par+15], a variant of Haskell's QuickCheck [CH00] for Coq programs. This framework provides two benefits. First, we can test that an $\mathcal{O}$QASM program is correct according to its specification. Formal proof in Coq can be labor-intensive, so PBT provides an easy-to-use confidence boost, especially prior to attempting formal proof. Second, we can use testing to assess the effect of *approximation* when developing oracles. For example, we might like to use approximate QFT, rather than full-precision QFT, in an arithmetic oracle in order to save gates. PBT can be used to test the effect of this approximation within the overall oracle by measuring the *distance* between the fully-precise result and the approximate one.

4.5.2 $\mathcal{O}$QIMP: A High-level Oracle Language

It is not uncommon for programmers to write oracles as metaprograms in a quantum assembly's host language (like how we wrote quantum algorithms as a combination of SQIR and Coq in Chapter 3), but this process can be tedious and error-prone, especially when trying to write optimized code. To make writing efficient arithmetic-based quantum oracles easier, we developed $\mathcal{O}$QIMP, a high-level imperative language that compiles to $\mathcal{O}$QASM.

Language Features An $\mathcal{O}$QIMP program is a sequence of function definitions, with the last acting as the "main" function. Each function definition is a series of statements that concludes by returning a value v. $\mathcal{O}$QIMP statements contain variable declarations, assignments (e.g., $x_r \texttt{ = } x_{/8}$), arithmetic computations ($n_1 \texttt{ = } i+1$), loops, conditionals, and function calls. Variables x have types τ, which are either primitive types ω^m or arrays thereof, of size n. A primitive type pairs a base type ω with a *quantum mode* m. There are three base types: type `nat` indicates non-negative (natural) numbers; type `fixedp` indicates fixed-precision real numbers in the range $(-1, 1)$; and type `bool` represents booleans. The programmer specifies the number of qubits to use to represent `nat` and `fixedp` numbers when invoking the $\mathcal{O}$QIMP compiler. The mode $m \in \{C, Q\}$ on a primitive type indicates when a type's value is expected to be known: C indicates that the value is based on a classical parameter of the oracle, and should be known at compile time; Q indicates that the value is a quantum input to the oracle, computed at run-time.

Compilation from $\mathcal{O}$QIMP to $\mathcal{O}$QASM The $\mathcal{O}$QIMP compiler performs *partial evaluation* [JGS93] on the input program given classical parameters; the residual program is compiled to a quantum circuit. In particular, we compile an $\mathcal{O}$QIMP program by evaluating its C-mode components, storing the results in a store, and then using these results while translating its Q-mode components into $\mathcal{O}$QASM code. We have verified that compilation from $\mathcal{O}$QIMP to $\mathcal{O}$QASM is correct in Coq, with a caveat: Proofs for assignment statements are parameterized by correctness statements about the involved operators. For example, when compiling $x = y + z$ we require that the $\mathcal{O}$QASM implementation of addition matches the $\mathcal{O}$QIMP specification.

4.5.3 Evaluation Highlights

To assess VQO's effectiveness we have used it to build several efficient oracles and oracle components, and have tested or proved their correctness. We highlight key results here; details are provided in Appendix B.

- We have implemented a variety of arithmetic operators in $\mathcal{O}$QASM, including QFT-, approximate QFT- and Toffoli-based multiplication, addition, modular multiplication, and modular division, as summarized in Figure 4.14. Overall, circuit sizes are competitive with, and oftentimes better than, those produced by Quipper [Gre+13], a state-of-the-art quantum programming framework. Qubit counts for the final QFT-based circuits are always lower, sometimes significantly so (up to 53%), compared to the Toffoli-based circuits. The QFT circuits also typically use fewer gates.

- Using $\mathcal{O}$QIMP, we implemented sine, cosine, and other geometric functions used in Hamiltonian simulation [Fey82], leveraging the arithmetic circuits described above. Compared to a sine function implemented in Quipper, VQO's uses far fewer qubits thanks to $\mathcal{O}$QIMP's partial evaluation.

- We used PBT to analyze the precision difference between QFT and approximate QFT (AQFT) circuits, and the suitability of AQFT in different algorithms. We found that the AQFT adder (which uses AQFT in place of QFT) is not an accurate implementation of addition, but that it can be used as a subcomponent of division/modulo with no loss of precision, reducing gate count by 4.5–79.3%.

- Finally, to put all of the pieces together, we implemented the ChaCha20 stream cipher [Ber08] in $\mathcal{O}$QIMP and used it as an oracle for Grover's search, previously implemented and proved correct in SQIR (Section 3.3.1). We used PBT to test the oracle's correctness. Combining its tested property with Grover's correctness property, we demonstrate that Grover's is able to invert the ChaCha20 function and find collisions.

type	Verified	Randomly Tested
Nat / Bool	$[x+N]_q$ $[x+y]_{q,t}$ $[(x \times N)\%M]_{q,t}$ $[x-N]_q$ $[N-x]_q$ $[x-y]_q$ $[x=N]_{q,t}$ $[x<N]_{q,t}$ $[x=y]_{q,t}$ $[x<y]_{q,t}$	$[x \times N]_{q,t}$ $[x \times y]_{q,t}$ $[x-N]_t$ $[N-x]_t$ $[x-y]_t$ $[x\%N]_{a,q,t}$ $[x/N]_{a,q,t}$
FixedP	$[x \times N]_q$ $[x+y]_t$ $[x-N]_q$ $[N-x]_q$ $[x-y]_t$ $[x=N]_{q,t}$ $[x<N]_{q,t}$ $[x=y]_{q,t}$ $[x<y]_{q,t}$	$[x+N]_t$ $[x+y]_q$ $[x-N]_t$ $[N-x]_t$ $[x-y]_q$ $[x \times N]_{q,t}$ $[x \times y]_{q,t}$ $[x/N]_{q,t}$

x, y = variables, M, N = constants,

$[]_{a,q,t}$ = AQFT-based (a), QFT-based (q), or Toffoli-based (t)

Figure 4.14: Summary of $\mathcal{O}$QASM arithmetic operations

4.6 Related Work

Quantum Compilers Quantum compilation is an active area. In addition to the circuit transpilers Qiskit, t|ket⟩, Staq, PyZX, and Nam et al. (discussed in Section 4.4), other recent efforts include quilc [Rig19b] and Cirq [Cir18]. Due to resource limits on near-term quantum machines, most of these tools 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. There has been more limited work on true *compilers*, which translate from a high-level source language to circuits. Some examples include ScaffCC [Jav+15], Project Q [SHT18], and Microsoft's under-development compiler from Q# to QIR [Gel20].

Verified Quantum Compilers Previously, formal verification has been applied to parts of the quantum compiler stack, but has not supported general quantum programs. Amy, Roetteler, and Svore [ARS17] and Rand, Paykin, and Zdancewic [RPZ18] developed certified compilers from source Boolean expressions to reversible circuits. Fagan and Duncan [FD18] verified an optimizer for ZX diagrams representing Clifford circuits (which use the non-universal gate set $\{CX, H, S\}$). Work on translation validation [ST19; BRW20; Kv19a] (discussed in Section 4.3.3) supports general quantum programs, but adds compile-time overhead and allows for the possibility of compile-time failure due to input/output inequivalence.

Concurrently with our work, Tao et al. [Tao+22] developed Giallar, a verification toolkit used to verify transformations in the Qiskit compiler. Their approach has two steps. First, like VOQC, they use Coq to prove that a circuit equivalence is valid. Second, they use symbolic execution to generate verification conditions for parts of the program 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. Compared to VOQC, Giallar is easier for non-experts to use to develop verified software, and simpler to integrate into existing workflows. However, more complex optimizations like rotation merging (which provides a significant benefit over the optimizations in Qiskit, per Section 4.4) cannot be implemented by applying small local rewrites. Giallar may also fail to prove an optimization correct, e.g., because of complicated control code. We have tried to alleviate proof burden in VOQC by providing a library of verified functions and optimization "templates" (Section 4.1.1). In particular, the optimizations implemented in Giallar could be implemented using our propagate-cancel template.

After publication of our initial work, Xu et al. [Xu+22] presented Quartz, a *super-optimizer* for quantum circuits with SMT-based verification of circuit equivalences. Quartz begins by applying (unverified implementations of) Toffoli decomposition and rotation merging, as described in Nam et al. [Nam+18]. Similar to our results, they find that these passes are a significant source of gate count reduction. After that, having generated a complete set of small (verified) circuit rewrite rules, they use (unverified) cost-based backtracking search to find the optimal sequence of rewrites.This is in contrast to VOQC, which applies only the fixed set of rules described in Section 4.2. Excitingly, this leads to higher gate count reduction than VOQC (see [Xu+22, Section 7]), but it does so at a higher cost: their experiments are run on a 128-core CPU with 512GB RAM, with a search timeout of 24 hours (compared to VOQC, which typically has a running time of under a second on a standard 2015 laptop). It would be interesting to analyze the results of Quartz's optimization to see which rewrite rules are most often triggered and implement and verify these inside of VOQC, improving VOQC's gate count reduction while avoiding the expense of backtracking search.

The problem of verified compilation from a high-level language to quantum circuits has received less attention. The only examples of verified compilers for quantum circuits are ReVerC [ARS17] and *ReQ*WIRE [RPZ18], both of which support verified translation from a low-level Boolean expression language to circuits consisting of X, $CNOT$, and $CCNOT$ gates. Compared to these tools, VQO supports both a higher-level classical source language ($\mathcal{O}$QIMP) and a more interesting quantum target language ($\mathcal{O}$QASM).

Oracles in Quantum Languages Many recent quantum programming languages (e.g., Quil [Rig19b], OpenQASM 2.0 [Cro+17], SQIR) describe low-level circuit programs and provide no abstractions for describing quantum oracles. Higher-level languages may provide library functions for performing common oracle operations (e.g., Q# [Svo+18], Scaffold [Jav+12; Jav+15]) or support compiling from classical (sub)programs to quantum circuits (e.g., Quipper [Gre+13]), but still leave important details (like uncomputation of ancilla qubits) to the programmer.

There has been some work on type systems to enforce that uncomputation happens correctly (e.g. Silq [Bic+20]), and on automated insertion of uncomputation circuits (e.g. Quipper, Unqomp [Par+21]), but while these approaches provide useful automation, they also lead to inefficiencies in compiled circuits. For example, all of these tools force compilation into the classical gate set X, $CNOT$, and $CCNOT$, which precludes the use of QFT-based arithmetic, which uses fewer qubits than Toffoli-based approaches. Of course, programmers are not obligated to use automation for constructing oracles—they can do it by hand for greater efficiency—but this risks mistakes. VQO allows programmers to produce oracles automatically from $\mathcal{O}$QIMP, or to manually implement oracle functions in $\mathcal{O}$QASM, in both cases supporting formal verification and testing.

Chapter 5

Applying Formal Verification to Q#

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 directly encode the high-level constructs used in many quantum algorithms. Meta-programming in Coq can recover 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 it has no support for formal verification. In this chapter, we discuss our efforts to adapt our work on formal verification to the Q# programming framework.

Figure 5.1 presents our vision of a verified software toolchain [App11] 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 a proof assistant. Various properties are proved about the Q* program, and then the original Q# program is compiled to Microsoft's quantum intermediate representation, QIR [Gel20] (or refined, if the proofs do not succeed). Next, the quantum/classical QIR is compiled to (potentially multiple) SQIR circuits, which are optimized with VOQC. 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, ensuring that properties proved of the Q# source also hold for code executed on the machine.

To achieve this goal requires developing a semantics for Q#, QIR, and "executable code", as well as verified compilers between them and SQIR—a substantial task. As an initial step toward this goal, in this chapter we present Q*, a quantum programming language that closely resembles Q#, shallowly embedded in the F* proof assistant [Dev22]. 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.

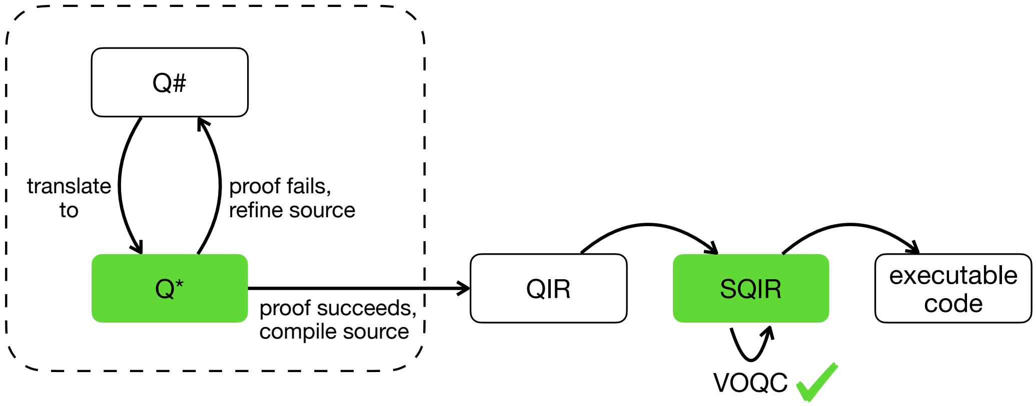


Figure 5.1: Overview of (proposed) toolchain for Q#. The dashed box encloses the contents of this chapter. Green marks verified components.

Under the hood, Q* is a 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 arguments in multi-qubit gates being distinct (which we call *linear 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 $\mathcal{Q}$WIRE [RPZ18] in Coq and $\mathcal{Q}$BRICKS [Cha+21] in Why3) that describe circuits, rather than high-level programs.

- *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 *nondeterministic* rather than *probabilistic* in its handling of measurement. While our semantics is less expressive than prior semantics for quantum programs, this choice simplifies implementation and proof, and is still powerful enough to verify useful properties.

- *Application.* We focus on automated proofs of properties that are useful to Q# developers, like linear 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 5.1), presents our formalism for quantum instruction trees and the Q* language (Section 5.2), and demonstrates applications of our formalism to verify useful properties of quantum programs (Section 5.3).

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 benefited from conversations with Bettina Heim, Andres Paz, Cassandra 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].

5.1 Background

5.1.1 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. 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. 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 checks made by the simulator. Figure 5.2 show Q# programs that pass Q#'s type checker, but fail during simulation. In Figure 5.2(a), the same qubit is used as both inputs to a multi-qubit gate, violating the quantum no cloning theorem. In Figure 5.2(b), the `Qubit` value returned from `InitQubit` is actually discarded, and unavailable for reuse.

Figure 5.2(c) contains a more subtle bug. In `PrepareBell`, qubit `q2` may be *entangled*

```
namespace QStar.Teleport {
    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])

with qubit `q1` when it is discarded, resulting in unexpected runtime behavior. For example, say that the logical qubit allocated to `q2` in `PrepareBell` is reallocated in another operation, say, to qubit `q3`. Now future updates to `q3`, like measurement, can impact the seemingly unrelated `q1`. To avoid this uncertainty, the Q# simulator checks at runtime that any discarded qubit is in a classical state. We can fix the bug in Figure 5.2(c) by inserting an explicit measurement of `q2` before the end of `PrepareBell`, or by taking `q2` as an additional input (i.e., not allocating it locally).

5.1.2 F⋆: A Proof Oriented Programming Language

F⋆ is a general-purpose functional programming language with effects and a dependent type system enabling program verification. Its type-checker proves that programs meet their specifications (i.e., satisfy their types) using a combination of SMT solving and interactive proof. After verification, F⋆ programs can be extracted to efficient OCaml, F#, or C. The F⋆ language has been used to certify key internet security protocols, including Transport Layer Security (TLS) [Bha+17], and produce high-performance cryptographic libraries [Zin+17].

Compared to Coq, F⋆ provides better automation (due to its use of an SMT solver) and a more natural programming interface. While Coq is primarily a proof assistant,

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

(a) Violates no cloning

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

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

(b) Uses a discarded qubit

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

(c) Discards an entangled qubit

Figure 5.2: Buggy Q# programs

designed to enable proofs about both programs and mathematical objects, F* is a *language* designed for producing verified software. F* is also developed and maintained by Microsoft Research, making it more feasible to integrate into Microsoft's Q# development environment.

5.2 Quantum Instruction Trees

We describe Q# operations using *instruction trees*, which are a sequence of `Action`, `Weaken`, and `Return` nodes, annotated with explicit preconditions, which must be true before program execution, and postconditions, which will be true after execution. An instruction tree has type `inst_tree a p q` where a is the return type, p is the precondition, and q is the postcondition. This type says that given a state that satisfies `p`, the instruction tree will return a value of type a and modify the state so that it satisfies q. An `Action` node applies an instruction (defined below), a `Return` node returns a value, and a `Weaken` node updates a tree's pre- and postconditions to allow for composition with other nodes. This 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 5.2. We leave the pre- and postcondition unspecified for brevity; we discuss their form in Section 5.3.

Semantics Instruction trees act on a program state consisting of:

- Σ: A set of defined qubits.

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

Listing 5.2: Example instruction tree. `PRE` and `POST` are placeholders for the pre- and postconditions we present in Figure 5.4.

$$
\begin{aligned}
\textbf{init} &: \text{unit} \to \text{qbit} \\
\textbf{disc} &: \text{qbit} \to \text{unit} \\
\textbf{meas} &: \text{qbit} \to \text{bool} \\
\textbf{had} &: \text{qbit} \to \text{unit} \\
\textbf{cnot} &: \text{qbit} \to \text{qbit} \to \text{unit}
\end{aligned}
$$

(a) Types

input $\xmapsto{\textbf{instruction}}$		**output**
(Σ, Ψ)	$\xmapsto{q \leftarrow \textbf{init}}$	$(\Sigma \cup q,\ \lvert 0\rangle_q \otimes \Psi)$
(Σ, Ψ)	$\xmapsto{\textbf{disc}\ q}$	$(\Sigma \setminus q,\ \text{disc}(q, \Psi))$
(Σ, Ψ)	$\xmapsto{b \leftarrow \textbf{meas}\ q}$	$(\Sigma,\ \text{meas}(q, \Psi))$
(Σ, Ψ)	$\xmapsto{\textbf{had}\ q}$	$(\Sigma,\ H_q \times \Psi)$
(Σ, Ψ)	$\xmapsto{\textbf{cnot}\ q_1\ q_2}$	$(\Sigma,\ CNOT_{q_1,q_2} \times \Psi)$

(b) Semantics

Figure 5.3: Example quantum instructions

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

Figure 5.3 lists example instructions with their input/output types and semantics. **init** allocates a qubit and returns a reference to it (type `qbit`), **disc** deallocates a qubit, **meas** measures a qubit and returns the result, and **had** and **cnot** apply a quantum gate. In the rule for **meas**, we do not associate output b with a particular probability. Instead, $\text{meas}(q, \Psi)$ chooses a value b (using an external source of randomness) such that $\left\lVert \lvert b\rangle_q \langle b\rvert \times \Psi \right\rVert \neq 0$ and updates Ψ to be $\frac{\lvert b\rangle_q \langle b\rvert \times \Psi}{\left\lVert \lvert b\rangle_q \langle b\rvert \times \Psi \right\rVert}$. $\text{disc}(q, \Psi)$ chooses b using the same constraint and updates Ψ to be $\frac{\langle b\rvert_q \times \Psi}{\left\lVert \lvert b\rangle_q \langle b\rvert \times \Psi \right\rVert}$. Subscripts on matrix terms (e.g., q in $\lvert b\rangle_q \langle q\rvert$ 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 sketched in Listing 5.3. The type of the interpretation function is shown in Listing 5.4. 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. In the type for `eval_inst_tree`, `a` is the return type, and `p` and `q` are pre- and postconditions. All three are annotated with a hash (#) to mark them as implicit. Given an instruction tree of type `inst_tree a p q` and a state satisfying the precondition `p`, `eval_inst_tree` returns a pair of a return value and a state that satisfies the postcondition `q`.

```
let rec eval_inst_tree it s0 : a & state
  = match it with
  | Return v → (v, s0)
  | Action i k → let (v, s1) = i.sem s0 in
                 eval_inst_tree (k v) s1
  | Weaken f → eval_inst_tree f s0
```

Listing 5.3: Simplified definition of instruction tree evaluation

```
// state = (Σ, Ψ)
let pre = state → prop
let post a = a → state → state → prop

val eval_inst_tree : (#a:Type) → (#p:pre) → (#q:post) →
      inst_tree a p q → (s0:state{p s0}) → (v:a & s1:state{q v s0 s1})
```

Listing 5.4: Types for instruction tree evaluation

Instructions In our implementation, instructions are F* records that define a quantum operation's semantics, useful pre- and postconditions, and optional implementations of adjoint and controlled variants, which are used to implement equivalents of Q#'s `Adjoint` and `Controlled` combinators. Our instructions are roughly equivalent to Q# *intrinsics*, which are predefined primitive operations. The instruction type `inst` is shown in Listing 5.5. 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.

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 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.

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 qubits parameters for an operation are

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

type inst (a:Type) = {
   // pre— and postconditions
   pre: pre;
   post: post a;

   // semantic 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 5.5: `inst` type

live (i.e., defined in Σ) and distinct, and a postcondition 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 currently not enforced by the Q# compiler (see Section 5.3.1). We are interested in exploring the possibility of specifying pre- and postconditions in the Q# source as annotations on operation definitions.

Listing 5.6 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.

5.3 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 instructions' pre- and postconditions, or manual; we discuss examples of both below.

5.3.1 Linear Qubit Usage

There are several basic requirements on how qubits can be used in order for the semantics in Figure 5.3 to make sense. In particular,

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

let sendMsg (qA : qbit) (qM : qbit)
  : inst_tree (𝔹 & 𝔹) (fun s0 → qA ≠ qM ∧ live s0 qA ∧ live s0 qM) eqpost
  = bind (adjoint (entangle qM qA)) (fun _ →
     Action (meas qM) (fun m1 →
      Action (meas qA) (fun m2 →
       Return eqpost (m1, m2))))

let decodeMsg (qB : qbit) (b1 : 𝔹) (b2 : 𝔹)
  : inst_tree unit (fun s0 → live s0 qB) eqpost
  = bind (cond b1
          (Action (pauli_x qB) (fun _ → Return eqpost ()))
          (Return eqpost ())) (fun _ →
     cond b2
     (Action (pauli_z qB) (fun _ → Return eqpost ()))
     (Return eqpost ()))

let teleport (qM : qbit) (qB : qbit)
  : inst_tree unit (fun s0 → qM ≠ qB ∧ live s0 qM ∧ live s0 qB) eqpost
  = using (fun s0 → qM ≠ qB ∧ live s0 qM ∧ live s0 qB) eqpost (fun qA →
     bind (entangle qA qB) (fun _ →
      bind (sendMsg qA qM) (fun classicalBits →
       bind (decodeMsg qB classicalBits) (fun _ →
        Return eqpost ()))))
```

Listing 5.6: Q* translation of `teleport` from Listing 5.1. `Weaken` nodes omitted for brevity. `qA`, `qB`, and `qM` are shorthand for `qAlice`, `qBob`, and `qMsg`. `live s q` says that qubit `q` is live in the state `s`. `eqpost` is the postcondition that says that the set of live qubits does not change.

$$\{\ \top\ \}\ q \leftarrow \textbf{init}\ \{\ \Sigma_1 = \Sigma_0 \cup q\ \}$$
$$\{\ q \in \Sigma_0\ \}\ \textbf{disc}\ q\ \{\ \Sigma_1 = \Sigma_0 \setminus q\ \}$$
$$\{\ q \in \Sigma_0\ \}\ b \leftarrow \textbf{meas}\ q\ \{\ \Sigma_1 = \Sigma_0\ \}$$
$$\{\ q \in \Sigma_0\ \}\ \textbf{had}\ q\ \{\ \Sigma_1 = \Sigma_0\ \}$$
$$\{\ q_1, q_2 \in \Sigma_0 \wedge q_1 \neq q_2\ \}\ \textbf{cnot}\ q_1\ q_2\ \{\ \Sigma_1 = \Sigma_0\ \}$$

(a) Linear qubit usage

$$\{\ \textbf{emp}\ \}\ q \leftarrow \textbf{init}\ \{\ q \mapsto_{s_1} |0\rangle\ \}$$
$$\{\ q \mapsto_{s_0} |\psi\rangle\ \}\ \textbf{disc}\ q\ \{\ \textbf{emp}\ \}$$
$$\{\ q, \overline{q} \mapsto_{s_0} |\psi\rangle\ \}\ b \leftarrow \textbf{meas}\ q\ \{\ q \mapsto_{s_1} |b\rangle \star \overline{q} \mapsto_{s_1} \mathrm{disc}(q, b, |\psi\rangle)\ \}$$
$$\{\ q, \overline{q} \mapsto_{s_0} |\psi\rangle\ \}\ \textbf{had}\ q\ \{\ q, \overline{q} \mapsto_{s_1} H_q\, |\psi\rangle\ \}$$
$$\{\ q_1, q_2, \overline{q} \mapsto_{s_0} |\psi\rangle\ \}\ \textbf{cnot}\ q_1\ q_2\ \{\ q_1, q_2, \overline{q} \mapsto_{s_1} CNOT_{q_1,q_2}\, |\psi\rangle\ \}$$

(b) Discard safety

Figure 5.4: Pre- and postconditions for enforcing linear qubit usage and discard safety. Preconditions may refer to the input state $s_0 = (\Sigma_0, \Psi_0)$ and postconditions may refer to both the input state and output state $s_1 = (\Sigma_1, \Psi_1)$.

- 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 *linear qubit usage*. Other quantum languages like Silq [Bic+20] and $\mathcal{Q}$WIRE [PRZ17] enforce this using linear type systems, which treat qubits as a resource.

To check linear qubit usage, we use the pre- and postconditions shown in Figure 5.4(a), which reflect the effect of the semantics in Figure 5.3 on the set of defined qubits Σ. As discussed above, 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 5.6 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. No Q# operation can return a live `Qubit` value, and, since Q# does not expose a primitive for discarding qubits, users cannot discard qubits that are not locally allocated.

5.3.2 Discard Safety

It is "safe" to discard a qubit when it is not entangled with any other program qubits. Discarding an entangled qubit may change the rest of the program state in unintended ways (see Section 5.1.1). 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 the pre- and postconditions shown in Figure 5.4(b), which are inspired by recent quantum separation logics [Le+22; Zho+21]. In Figure 5.4(b), $\overline{q}$ refers to a (potentially empty) set of qubits and $\overline{q} \mapsto_s |\psi\rangle$ says that in state $s = (\Sigma, \Psi)$, all qubits in $\overline{q}$ are live in Σ and the portion of Ψ that corresponds to qubits $\overline{q}$ is in state $|\psi\rangle$. In the case where $\overline{q}$ is empty, we write **emp**. We adopt 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 to produce (Σ_1, Ψ_1) and (Σ_2, Ψ_2) so that P_1 holds of (Σ_1, Ψ_1), P_2 holds of (Σ_2, Ψ_2), and $(\Sigma, \Psi) = (\Sigma_1 \cup \Sigma_2, \Psi_1 \otimes \Psi_2)$. $\star$ is commutative and $P \star \mathbf{emp} = P$.

The predicate $\overline{q} \mapsto_s |\psi\rangle$ implies that qubits in $\overline{q}$ 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 Figure 5.4(b) 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 (Figure 5.3).

To analyze a program for discard safety, we can use the same initial precondition used to check linear qubit usage (i.e., all input qubits are live and distinct), extended with the condition that $\overline{q} \mapsto_s |\psi\rangle$ where $\overline{q}$ includes (at least) all input qubits and $|\psi\rangle$ is some appropriately-sized vector state. The rules in Figure 5.4(b) 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., $q \mapsto_s |\psi\rangle$). After `applyH` executes, we will have that $q \mapsto_{s'} H \times |\psi\rangle$. We can extend this specification to a larger program that also includes qubits q_1 and q_2: Say that initially $q_1, q_2 \mapsto_s |\phi\rangle$, then after `applyH` executes, we can conclude that $q_1, q_2 \mapsto_{s'} |\phi\rangle \star q \mapsto_{s'} H \times |\psi\rangle$. In other words, q_1 and q_2 are unaffected by `applyH`.

Entailment Reasoning

Using the rules in Figure 5.4(b), we can only produce a $\star$-expression (which guarantees a lack of entanglement) by applying the rule for **meas**. Using the frame rule, we can show that qubits that were previously unentangled remain unentangled after certain operations. These two cases are enough to prove discard safety for programs that measure their qubits before discarding (like our `teleport` example); however, this is not sufficient in the general case where values may be *uncomputed*. 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.

For example, consider the following program, which computes the logical AND of qubits `q1`, `q2`, `q3`, 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);`. This construct is designed to facilitate uncomputation. 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 in a formal proof, we need to (manually) manipulate the matrix expressions inside predicates and selectively apply the following entailment rule:

$$\overline{q_1}, \overline{q_2} \mapsto_s |\psi_1\rangle_{\overline{q_1}} \otimes |\psi_2\rangle_{\overline{q_2}} \Longleftrightarrow (\overline{q_1} \mapsto_s |\psi_1\rangle) \star (\overline{q_2} \mapsto_s |\psi_2\rangle).$$

Applying this rule requires reasoning about a 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`,

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

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

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

We can show that this $CCNOT$ matrix-vector produce can be rewritten as $|\psi'\rangle \otimes |0\rangle$

```
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 5.7: 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 state `bell00`.

for some $|\psi'\rangle$, allowing us to derive that $q_1, q_2, q_3, o \mapsto_s |\psi'\rangle \star a \mapsto_s |0\rangle$. This proves that `anc` is not entangled with the other qubits, and is thus safe to discard.

5.3.3 Functional Correctness

A program is *functionally correct* if for every input, it produces an output that satisfies the provided specification. Chapter 3 presented several examples of functional correctness properties proved about SQIR programs; for the most part, these properties state that a program prepares a particular quantum state. Another example of a functional correctness property, specific to Q#, is that a custom adjoint specialization inverts the original operation.

Due to our choice of semantics, which ignores the probability of different measurement outcomes, there are some properties that we will not be able to verify. For example, we cannot prove that a program returns a result with a particular probability, only that it may return the result. The algorithms we discussed in Section 3.3 (the general case of QPE, Grover's, and Shor's) use probabilities in their statements of correctness, so we would not be able to prove the properties presented in that section, as written. However, the nondeterministic view is sufficient for proving properties like discard safety and linear 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, and measurement-based uncomputation [Ber+19] generates an operation that behaves like an adjoint, regardless of intermediate measurement outcomes.

We support two approaches to proving functional correctness properties in Q*. The first option is to use the pre- and postconditions from Section 5.3.1 to ensure basic program well-formedness, and then prove custom lemmas in F* about the program's semantics using `eval_inst_tree` from Listing 5.3. This is exactly what we did in Chapter 3: 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 5.7.

Alternatively, we can use the rules from Section 5.3.2 to reason not only about discard safety, but also about correctness, following the example of existing quantum

$\{\ q_B \mapsto |0\rangle \star q_M \mapsto |\psi\rangle\ \}$

```
use qAlice = Qubit();
```

$\{\ q_B \mapsto |0\rangle \star q_M \mapsto |\psi\rangle \star q_A \mapsto |0\rangle\ \}$
$\Leftrightarrow \{\ q_A, q_B \mapsto |00\rangle \star q_M \mapsto |\psi\rangle\ \}$

```
Entangle(qAlice, qBob);
```

$\{\ q_A, q_B \mapsto CNOT_{q_A,q_B} \times H_{q_A} \times |00\rangle \star q_M \mapsto |\psi\rangle\ \}$
$\Leftrightarrow \{\ q_A, q_B \mapsto \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \star q_M \mapsto |\psi\rangle\ \}$
$\Leftrightarrow \{\ q_M, q_A, q_B \mapsto |\psi\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)\ \}$

```
let classicalBits = SendMsg(qAlice, qMsg);
```

$\{\ q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star$
$\quad q_B \mapsto \mathbf{disc}(q_A, \mathbf{disc}(q_M, H_{q_M} \times CNOT_{q_M,q_A} \times (|\psi\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))))\ \}$
$\Leftrightarrow \{\ q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star q_B \mapsto Z^{b_1}X^{b_2}|\psi\rangle\ \}$

```
DecodeMsg(qBob, classicalBits);
```

$\{\ q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star$
$\quad (b_1 \wedge b_2 \implies q_B \mapsto X \times Z \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$
$\quad (b_1 \wedge \neg b_2 \implies q_B \mapsto Z \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$
$\quad (\neg b_1 \wedge b_2 \implies q_B \mapsto X \times Z^{b_1}X^{b_2}|\psi\rangle) \wedge$
$\quad (\neg b_1 \wedge \neg b_2 \implies q_B \mapsto Z^{b_1}X^{b_2}|\psi\rangle)\ \}$
$\Leftrightarrow \{\ q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star q_B \mapsto |\psi\rangle\ \}$

```
// deallocate qAlice (done implicitly)
```

$\{\ q_M \mapsto |b_1\rangle \star q_B \mapsto |\psi\rangle\ \}$

Figure 5.5: 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.

separation logics [Le+22; Zho+21]. This is our preferred approach to proof, as it more closely ties the program with its specification; we are currently working to implement this approach by building on top of F*'s implementation of a concurrent separation logic, Steel [Fro+21]. Figure 5.5 sketches how we can use the pre- and postconditions for discard safety to verify correctness of quantum teleportation. We show all intermediate reasoning steps, aside from application of the frame rule, and we make use of the properties of $\star$ and $\mapsto$ discussed in Section 5.3.2.

5.4 Related Work

Linear Qubit Usage Several quantum languages including Selinger and Valiron's quantum lambda calculus [SV09], $\mathcal{Q}$WIRE [PRZ17], Proto-Quipper [FKS20], and

Silq [Bic+20] enforce linear qubit usage using linear type systems. So far, industry languages like Q# have not adopted linear typing, but Singhal et al.'s work on $\lambda_{Q\#}$ [Sin+22], an idealized version of Q#, explores extensions to the type system that give similar guarantees.

Discard Safety Our approach to enforcing discard safety is based on existing quantum separation logics [Le+22; Zho+21]. Several prior works have used different techniques to enforce similar properties. ScaffCC defines a simple entanglement analysis for circuits consisting of n-controlled X gates (X, $CNOT$, $CCNOT$, $\ldots$) [Jav+15, Sec 7.1], which tracks sets of potentially entangled qubits and automatically recognizes uncomputation. Silq's type system [Bic+20] enforces that qubits are uncomputable (i.e., unentangled) before they go out of scope, restricting to the same (classical) gate set as ScaffCC. ReVerC [ARS17] and *Re*$\mathcal{Q}$WIRE [Ran+19] aim to formally verify classical reversible circuits, which requires ensuring that discarded values are appropriately uncomputed. 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 our rules in Figure 5.4(b) and the dynamic checks take the place of our manual entailment proofs. Other static approaches to tracking entanglement include Rand et al. [Ran+21], Honda [Hon15], and Perdrix [Per08].

Functional Correctness Compared to prior work on verifying functional correctness of quantum programs (e.g., SQIR, $\mathcal{Q}$BRICKS [Cha+21], and QHL [Yin12], discussed in Section 3.2), Q* supports a higher-level source language, and interfaces with the popular language Q#, making it more realistic for non-expert users to develop verified quantum code. Q* also focuses on providing automation for proving general well-formedness properties, like linear qubit usage and discard safety, rather than specific proofs of functional correctness.

Chapter 6

Conclusion and Future Work

The goal of this dissertation was 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. To do this, we 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 approach to supporting formal verification in the high-level language Q#. There is plenty left to do; 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 3.3 is the textbook presentation; in practice, it is more popular to consider an approximate QFT (see Section 4.5).

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, as we did for Shor's algorithm in Section 3.3.3. It also requires considering issues 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 $\mathcal{Q}$WIRE, SQIR, $\mathcal{Q}$BRICKS, and (to some extent) QWhile, it would be interesting to verify programs written in higher-level languages like Silq [Bic+20] or Q# [Svo+18], as we aim to do in our work on $Q^{\star}$.

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. $\mathcal{Q}$BRICKS includes a size predicate, and such a predicate can easily be added to SQIR (as we did for Shor's in Section 3.3.3), but so far such predicates have only been used to estimate the number of gates in a (parameterized) circuit—they have not been used to prove optimal 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+20]. 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 a 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 moving forward.

Appendix A

Full VOQC Evaluation Results

This chapter contains the full results of the evaluation from Section 4.4.

Staq Benchmarks Tables A.1 and A.2 show the results of running Qiskit, t|ket⟩, and VOQC (using the IBM gate set) on the Staq benchmarks [AG20]. Tables A.3 to A.5 show the results of running Staq, PyZX, and VOQC (using the RzQ gate set). Table A.6 shows the running times for each tool. In each row, for each type of gate, we shade the cell of the best-performing optimizer. The geometric mean reduction for each gate type is given in the last row. Cases where PyZX hit our timeout (so we ran full_reduce, but not full_optimize) are marked with stars.

Nam Benchmarks Tables A.7 to A.10 show the results of running VOQC on the "Arithmetic and Toffoli" and "QFT and Adders" benchmarks used by Nam et al. [Nam+18]. All results were obtained using a laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Catalina. For timings, we take the median of three trials. We do not re-run Nam et al. (which is proprietary software), but instead report the results from their paper; their results were collected on a similar machine with 8 GB RAM running OS X El Capitan. Their implementation is written in Fortran.

Table A.1: Reduced total gate counts for the IBM gate set on the Staq benchmarks. Shaded cells mark the best performance.

Name	**Original**	**Qiskit**	**t|ket⟩**	**VOQC**
adder_8	934	820	806	643
barenco_tof_3	60	53	52	46
barenco_tof_4	120	104	100	89
barenco_tof_5	180	155	148	132
barenco_tof_10	480	410	388	347
csla_mux_3	170	154	141	146
csum_mux_9	448	403	366	308
cycle_17_3	9738	8397	7753	5963
gf2^4_mult	243	206	206	190
gf2^5_mult	379	318	319	289
gf2^6_mult	545	454	454	408
gf2^7_mult	741	614	614	547
gf2^8_mult	981	804	806	703
gf2^9_mult	1223	1006	1009	882
gf2^10_mult	1509	1238	1240	1080
gf2^16_mult	3885	3148	3150	2691
grover_5	831	669	605	526
ham15-low	443	406	390	351
ham15-med	1272	1105	1015	820
ham15-high	5308	4589	4371	3532
hwb6	257	236	223	205
mod_adder_1024	4285	3721	3542	2832
mod_mult_55	119	110	100	83
mod_red_21	272	237	226	191
mod5_4	65	56	56	53
qcla_adder_10	539	477	441	408
qcla_com_7	463	407	363	292
qcla_mod_7	920	813	752	666
qft_4	179	97	81	94
rc_adder_6	200	177	159	141
tof_3	45	41	40	36
tof_4	75	68	66	58
tof_5	105	95	92	80
tof_10	255	230	222	190
vbe_adder_3	160	133	121	100
Geo. Mean Red.	–	14.4%	18.5%	28.5%

Table A.2: Reduced two-qubit gate counts for the IBM gate set on the Staq benchmarks. Shaded cells mark the best performance.

Name	**Original**	**Qiskit**	**t|ket⟩**	**VOQC**
adder_8	409	385	383	337
barenco_tof_3	24	24	24	22
barenco_tof_4	48	48	46	44
barenco_tof_5	72	72	68	66
barenco_tof_10	192	192	178	176
csla_mux_3	80	69	69	72
csum_mux_9	168	168	168	168
cycle_17_3	3915	3903	3723	3001
gf2^4_mult	99	99	99	99
gf2^5_mult	154	154	154	154
gf2^6_mult	221	221	221	221
gf2^7_mult	300	300	300	300
gf2^8_mult	405	405	402	405
gf2^9_mult	494	494	494	494
gf2^10_mult	609	609	609	609
gf2^16_mult	1581	1581	1581	1581
grover_5	288	288	288	248
ham15-low	236	236	226	220
ham15-med	534	533	498	434
ham15-high	2149	2143	2110	1853
hwb6	116	115	111	108
mod_adder_1024	1720	1702	1702	1402
mod_mult_55	48	48	48	40
mod_red_21	105	105	105	93
mod5_4	28	28	28	28
qcla_adder_10	233	213	213	207
qcla_com_7	186	174	174	148
qcla_mod_7	382	366	366	338
qft_4	46	44	38	46
rc_adder_6	93	81	81	73
tof_3	18	18	18	16
tof_4	30	30	30	26
tof_5	42	42	42	36
tof_10	102	102	102	86
vbe_adder_3	70	58	58	54
Geo. Mean Red.	–	2.3%	3.8%	9.8%

Table A.3: Reduced total gate counts for the RzQ gate set on the Staq benchmarks. Shaded cells mark the best performance. Some PyZX results are marked with a star* to indicate that full_optimize exceeded our time limit, so we only used full_reduce.

Name	Original	Staq	PyZX	VOQC
adder_8	934	756	911	682
barenco_tof_3	60	48	53	50
barenco_tof_4	120	90	88	95
barenco_tof_5	180	132	184	140
barenco_tof_10	480	342	460	365
csla_mux_3	170	174	291	156
csum_mux_9	448	294	508	308
cycle_17_3	9738	7143	11157*	6314
gf2^4_mult	243	242	277	192
gf2^5_mult	379	366	650	291
gf2^6_mult	545	540	1309*	410
gf2^7_mult	741	714	1843*	549
gf2^8_mult	981	968	2610*	705
gf2^9_mult	1223	1178	3162*	885
gf2^10_mult	1509	1484	4118*	1084
gf2^16_mult	3885	3800	11627*	2695
grover_5	831	678	696	586
ham15-low	443	417	650	373
ham15-med	1272	989	1205	880
ham15-high	5308	3967	5084*	3739
hwb6	257	237	274	221
mod_adder_1024	4285	3401	5572*	3039
mod_mult_55	119	107	123	90
mod_red_21	272	235	340	214
mod5_4	65	53	27	56
qcla_adder_10	539	445	894*	438
qcla_com_7	463	329	473	314
qcla_mod_7	920	725	1426	723
qft_4	179	170	191	156
rc_adder_6	200	173	256	157
tof_3	45	40	55	40
tof_4	75	65	83	65
tof_5	105	90	121	90
tof_10	255	215	338	215
vbe_adder_3	160	101	155	101
Geo. Mean Red.	–	15.4%	-27.8%	23.2%

Table A.4: Reduced two-qubit gate counts for the RzQ gate set on the Staq benchmarks. Shaded cells mark the best performance. Some PyZX results are marked with a star* to indicate that full_optimize exceeded our time limit, so we only used full_reduce.

Name	**Original**	**Staq**	**PyZX**	**VOQC**
adder_8	409	382	609	337
barenco_tof_3	24	20	28	22
barenco_tof_4	48	38	44	44
barenco_tof_5	72	56	116	66
barenco_tof_10	192	146	299	176
csla_mux_3	80	88	201	72
csum_mux_9	168	126	371	168
cycle_17_3	3915	3351	6229*	3001
gf2^4_mult	99	141	211	99
gf2^5_mult	154	209	526	154
gf2^6_mult	221	327	789*	221
gf2^7_mult	300	423	1162*	300
gf2^8_mult	405	603	1745*	405
gf2^9_mult	494	713	2065*	494
gf2^10_mult	609	927	2779*	609
gf2^16_mult	1581	2427	8368*	1581
grover_5	288	263	395	248
ham15-low	236	252	477	220
ham15-med	534	483	821	434
ham15-high	2149	1870	2343*	1853
hwb6	116	115	163	108
mod_adder_1024	1720	1584	2813*	1402
mod_mult_55	48	42	82	40
mod_red_21	105	94	211	93
mod5_4	28	28	14	28
qcla_adder_10	233	209	382*	207
qcla_com_7	186	146	312	148
qcla_mod_7	382	334	1012	338
qft_4	46	56	72	46
rc_adder_6	93	81	162	73
tof_3	18	16	31	16
tof_4	30	26	48	26
tof_5	42	36	68	36
tof_10	102	86	213	86
vbe_adder_3	70	54	105	54
Geo. Mean Red.	–	0.6%	-91.7%	9.8%

Table A.5: Reduced T-gate counts for the RzQ gate set on the Staq benchmarks. Shaded cells mark the best performance. Some PyZX results are marked with a star* to indicate that full_optimize exceeded our time limit, so we only used full_reduce.

Name	**Original**	**Staq**	**PyZX**	**VOQC**
adder_8	399	179	167	215
barenco_tof_3	28	16	16	16
barenco_tof_4	56	28	28	28
barenco_tof_5	84	40	40	40
barenco_tof_10	224	100	100	100
csla_mux_3	70	62	47	64
csum_mux_9	196	84	76	84
cycle_17_3	4529	1821	1821*	1821
gf2^4_mult	112	66	50	68
gf2^5_mult	175	113	92	115
gf2^6_mult	252	148	150*	150
gf2^7_mult	343	215	217*	217
gf2^8_mult	448	262	264*	264
gf2^9_mult	567	348	351*	351
gf2^10_mult	700	406	410*	410
gf2^16_mult	1792	1032	1040*	1040
grover_5	336	166	166	172
ham15-low	161	97	97	97
ham15-med	574	242	211	248
ham15-high	2457	1021	1019*	1049
hwb6	105	75	74	75
mod_adder_1024	1995	1011	1011*	1011
mod_mult_55	49	37	28	35
mod_red_21	119	73	72	73
mod5_4	28	8	8	16
qcla_adder_10	238	162	162*	164
qcla_com_7	203	95	92	95
qcla_mod_7	413	237	226	249
qft_4	91	50	66	67
rc_adder_6	77	47	47	47
tof_3	21	15	15	15
tof_4	35	23	23	23
tof_5	49	31	31	31
tof_10	119	71	71	71
vbe_adder_3	70	24	24	24
Geo. Mean Red.	–	45.4%	47.1%	43.1%

Table A.6: Median running times (in seconds) on Staq benchmarks.

Name	# qubits	# gates	Qiskit	t|ket⟩	Staq	PyZX	VOQC
adder_8	24	934	1.43	4.47	0.07	19.16	0.05
barenco_tof_3	5	60	0.09	0.18	<0.01	2.24	<0.01
barenco_tof_4	7	120	0.17	0.38	<0.01	2.54	<0.01
barenco_tof_5	9	180	0.26	0.58	<0.01	0.58	<0.01
barenco_tof_10	19	480	0.63	1.49	0.02	63.04	0.02
csla_mux_3	15	170	0.29	0.75	0.01	72.74	<0.01
csum_mux_9	30	448	0.58	1.52	0.02	45.87	0.01
cycle_17_3	35	9738	13.08	41.56	1.63	498.51*	5.74
gf2^4_mult	12	243	0.32	1.04	0.01	57.25	0.01
gf2^5_mult	15	379	0.50	1.65	0.02	0.23	0.01
gf2^6_mult	18	545	0.70	2.22	0.04	0.43*	0.02
gf2^7_mult	21	741	1.13	3.03	0.08	0.73*	0.04
gf2^8_mult	24	981	1.28	4.16	0.14	1.48*	0.07
gf2^9_mult	27	1223	1.62	5.04	0.22	1.94*	0.11
gf2^10_mult	30	1509	2.09	6.22	0.35	2.95*	0.17
gf2^16_mult	48	3885	5.28	17.37	3.30	25.41*	1.24
grover_5	9	831	1.01	2.37	0.03	34.30	0.01
ham15-low	17	443	0.67	2.69	0.02	119.85	0.01
ham15-med	17	1272	1.88	4.53	0.07	261.71	0.05
ham15-high	20	5308	7.53	20.59	0.33	23.31*	0.66
hwb6	7	257	0.38	1.04	0.01	58.54	<0.01
mod_adder_1024	28	4285	5.95	15.56	0.37	171.43*	1.31
mod_mult_55	9	119	0.18	0.47	<0.01	5.62	<0.01
mod_red_21	11	272	0.37	0.90	0.01	54.76	0.01
mod5_4	5	65	0.10	0.18	<0.01	2.17	<0.01
qcla_adder_10	36	539	0.80	1.97	0.04	1.86*	0.02
qcla_com_7	24	463	0.65	1.50	0.02	79.69	0.01
qcla_mod_7	26	920	1.42	3.33	0.08	411.36	0.06
qft_4	5	179	0.19	0.30	<0.01	14.83	<0.01
rc_adder_6	14	200	0.31	0.92	0.01	14.38	<0.01
tof_3	5	45	0.07	0.15	<0.01	8.68	<0.01
tof_4	7	75	0.11	0.24	<0.01	13.22	<0.01
tof_5	9	105	0.15	0.33	<0.01	25.43	<0.01
tof_10	19	255	0.35	0.81	0.01	75.79	0.01
vbe_adder_3	10	160	0.24	0.51	<0.01	19.21	<0.01
Geo. Mean	–	–	0.60	1.53	0.02	14.58	0.01

Table A.7: Reduced total gate counts on the "Arithmetic and Toffoli" circuits. The reported VOQC time only includes optimization time. Nam (H) results were not available for the large benchmarks. Red cells indicate programs found to have been optimized incorrectly [Kv19b, Section 2].

Name	Orig. Total	Nam (L) Total	Nam (L) t(s)	Nam (H) Total	Nam (H) t(s)	VOQC Total	VOQC t(s)
adder_8	900	646	0.004	**606**	0.101	682	0.048
barenco_tof_3	58	42	<0.001	**40**	0.001	50	0.001
barenco_tof_4	114	78	<0.001	**72**	0.001	95	0.002
barenco_tof_5	170	114	<0.001	**104**	0.003	140	0.003
barenco_tof_10	450	294	0.001	**264**	0.012	365	0.019
csla_mux_3	170	161	<0.001	**155**	0.009	158	0.003
csum_mux_9	420	294	<0.001	**266**	0.009	308	0.006
gf2^4_mult	225	**187**	0.001	**187**	0.009	192	0.006
gf2^5_mult	347	296	0.001	296	0.020	**291**	0.012
gf2^6_mult	495	**403**	0.003	**403**	0.047	410	0.025
gf2^7_mult	669	555	0.004	555	0.105	**549**	0.045
gf2^8_mult	883	712	0.006	712	0.192	**705**	0.070
gf2^9_mult	1095	891	0.010	891	0.347	**885**	0.119
gf2^10_mult	1347	**1070**	0.009	**1070**	0.429	1084	0.183
gf2^16_mult	3435	2707	0.065	2707	5.566	**2695**	1.347
gf2^32_mult	13593	10601	1.834	10601	275.698	**10577**	26.808
gf2^64_mult	53691	41563	58.341	–	–	**41515**	546.887
gf2^128_mult	213883	165051	1744.746	–	–	**164955**	9841.797
gf2^131_mult	224265	173370	1953.353	–	–	**173273**	10877.112
gf2^163_mult	346533	267558	4955.927	–	–	**267437**	27612.565
mod5_4	63	**51**	<0.001	**51**	0.001	56	<0.001
mod_mult_55	119	91	<0.001	91	0.002	**90**	0.002
mod_red_21	278	184	<0.001	**180**	0.008	214	0.005
qcla_adder_10	521	411	0.002	**399**	0.044	438	0.018
qcla_com_7	443	**284**	0.001	**284**	0.016	314	0.013
qcla_mod_7	884	636	0.004	624	0.077	**723**	0.058
rc_adder_6	200	142	<0.001	**140**	0.004	157	0.003
tof_3	45	**35**	<0.001	**35**	<0.001	40	<0.001
tof_4	75	**55**	<0.001	**55**	<0.001	65	0.001
tof_5	105	**75**	<0.001	**75**	0.001	90	0.002
tof_10	255	**175**	<0.001	**175**	0.004	215	0.006
vbe_adder_3	150	**89**	<0.001	**89**	0.001	101	0.002
Geo. Mean Red.	–	24.6%		26.4%		19.2%	

Table A.8: Total gate count reduction on Quipper adder circuits. VOQC's H and T counts are identical to Nam (L) and (H), but the total Rz and $CNOT$ counts are higher due to Nam et al.'s specialized Toffoli decomposition. The difference between Nam (L) and Nam (H) is entirely due to $CNOT$ count. Our initial gate counts are higher than those reported by Nam et al. because we do not have special handling for +/- control Toffoli gates; we simply consider the standard Toffoli gate conjugated by additional X gates.

	Original	**Nam (L)**		**Nam (H)**		**VOQC**	
n	**Total**	**Total**	**t(s)**	**Total**	**t(s)**	**Total**	**Opt. t(s)**
8	585	239	0.001	190	0.006	352	0.02
16	1321	527	0.003	414	0.018	784	0.12
32	2793	1103	0.014	862	0.066	1648	0.63
64	5737	2255	0.057	1758	0.598	3376	3.30
128	11625	4559	0.244	3550	4.697	6832	16.37
256	23401	9167	1.099	7134	34.431	13744	79.74
512	46953	18383	5.292	14302	307.141	27568	394.74
1024	94057	36815	25.987	28638	2446.336	55216	1894.41
2048	188265	73679	145.972	57310	23886.841	110512	9307.36
Avg. Red.		63.7%		71.6%		45.7%	

Table A.9: Results on QFT circuits. Exact timings and gate counts are not available for Nam (L) or Nam (H), but our results are consistent with those reported in Nam et al. [Nam+18, Figure 1].

	Original			**VOQC**			
n	$CNOT$	R_z	H	$CNOT$	R_z	H	**Opt. t(s)**
8	56	84	8	56	42	8	<0.01
16	228	342	16	228	144	16	<0.01
32	612	918	32	612	368	32	0.01
64	1380	2070	64	1380	816	64	0.07
128	2916	4374	128	2916	1712	128	0.39
256	5988	8982	256	5988	3504	256	2.34
512	12132	18198	512	12132	7088	512	15.69
1024	24420	36630	1024	24420	14256	1024	106.71
2048	48996	73494	2048	48996	28592	2048	674.11
Avg. Red.				0%	59.3%	0%	

Table A.10: Results on QFT-based adder circuits. Final gate counts are identical for VOQC and Nam (L).

	Original			**VOQC**				**Nam (L)**
n	*CNOT*	R_z	H	*CNOT*	R_z	H	**Opt. t(s)**	**t(s)**
8	184	276	16	184	122	16	<0.01	<0.001
16	716	1074	32	716	420	32	0.02	0.001
32	1900	2850	64	1900	1076	64	0.13	0.002
64	4268	6402	128	4268	2388	128	0.90	0.004
128	9004	13506	256	9004	5012	256	5.52	0.08
256	18476	27714	512	18476	10260	512	36.80	0.018
512	37420	56130	1024	37420	20756	1024	255.20	0.045
1024	75308	112962	2048	75308	41748	2048	1695.65	0.115
2048	151084	226626	4096	151084	83732	4096	8481.66	0.215
Avg. Red.				0%	61.8%	0%		

Appendix B

Full VQO Evaluation Results

This chapter contains the full results of the evaluation from Section 4.5.

Preliminaries For each arithmetic operator, we present up to three versions: QFT uses a quantum Fourier transform-based circuit for addition and subtraction [Dra00]; $AQFT$ uses the same circuits, but with an approximate QFT in place of the QFT; and $TOFF$ uses a ripple-carry adder [MS12], which uses classical controlled-controlled-not (Toffoli) gates.

To get gate counts for $\mathcal{O}$QASM operators, we translate to SQIR and then to OpenQASM 2.0 [Qis17]; to get gate counts for Quipper circuits, we convert to OpenQASM using a tool from Bian [Bia20]. We run all OpenQASM circuits through VOQC to ensure that the outputs account for inefficiencies in automatically-generated circuit programs (e.g., no-op gates inserted in the base case of a recursive function). VOQC outputs the final result using its IBM gate set $\{U_1, U_2, U_3, CNOT\}$. We define all the arithmetic operations in for arbitrary input sizes; the limited sizes in our experiments (8 and 16 bits) are to account for inefficiencies in VOQC. For the largest circuits we consider (the modular multipliers), running VOQC takes about 10 minutes.

Comparing $\mathcal{O}$QASM and Quipper Overall, Figure B.1(a–c) shows that operator implementations in $\mathcal{O}$QASM consume resources comparable to those available in Quipper, often using fewer qubits and gates, both for Toffoli- and QFT-based operations. In the case of the QFT adder, the difference in results is due to the Quipper-to-OpenQASM converter: Bian [Bia20] decomposes a controlled-Rz gate into a circuit that uses 19 single-qubit gates, 12 two-qubit gates, and an ancilla qubit (after VOQC decompositions). In contrast, VQO's decomposition for controlled-Rz uses 3 single-qubit gates, 2 two-qubit gates, and no ancilla qubits. In the other cases (all Toffoli-based circuits), we made choices when implementing the oracles that improved their resource usage.

Comparing QFT- and Toffoli-based Arithmetic The results show that the QFT-based implementations always use fewer qubits and often use fewer gates. We found during evaluation that gate counts are highly sensitive to decompositions used

	# qubits	# gates	Verified
$\mathcal{O}$QASM TOFF	33	423	✓
$\mathcal{O}$QASM QFT	32	1206	✓
$\mathcal{O}$QASM QFT (const)	16	756 ± 42	✓
Quipper TOFF	47	768	
Quipper QFT	33	6868	
Quipper TOFF (const)	31	365 ± 11	

(a) Addition circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
$\mathcal{O}$QASM TOFF	49	11265	6 / 74
$\mathcal{O}$QASM TOFF (const)	33	1739 ± 367	3 / 31
$\mathcal{O}$QASM QFT	48	4339	4 / 138
$\mathcal{O}$QASM QFT (const)	32	1372 ± 26	4 / 158
Quipper TOFF	63	8060	
Quipper TOFF (const)	41	2870 ± 594	

(b) Multiplication circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
$\mathcal{O}$QASM TOFF (const)	49	28768	16 / 397
$\mathcal{O}$QASM QFT (const)	34	15288	5 / 412
$\mathcal{O}$QASM AQFT (const)	34	5948	4 / 323
Quipper TOFF	98	37737	

(c) Division/modulo circuits (16 bits)

	# qubits	# gates	Verified
$\mathcal{O}$QASM TOFF (const)	41	56160	✓
$\mathcal{O}$QASM QFT (const)	19	18503	✓

(d) Modular multiplication circuits (8 bits)

Figure B.1: Comparison of $\mathcal{O}$QASM and Quipper arithmetic operators. In the "const" case, one argument is a classically-known constant parameter. For (a–b) we present the average ($\pm$ standard deviation) over 20 randomly selected constants c with $0 < c < 2^{16}$. For division/modulo, x mod n, we only consider the case when $n = 1$, which results in the maximum number of circuit iterations; the Quipper version assumes n is a variable, but uses the same number of iterations as the constant case when $n = 1$. In (d), we use the constant 255 ($= 2^8 - 1$) for the modulus and set the other constant to 173 (which is invertible mod 255). Quipper supports no QFT-based circuits aside from an adder, and does not have a built-in operation for modular multiplication (which is different from multiplication followed by modulo in the presence of overflow). "QC time" is the time (in seconds) for QuickChick to run 10,000 tests.

Precision	# gates	Error
16 bits (full)	1206	± 0
15 bits	1063	± 1
14 bits	929	± 3

(a) Varying the precision in a 16-bit adder

# iters.	TOFF	QFT	AQFT
1	1798	1794	1717
4	7192	4432	3488
8	14384	8017	4994
12	21576	11637	5684
16	28768	15288	5948

(b) Gate counts for TOFF vs. QFT vs. AQFT division/modulo circuits

Figure B.2: Effects of approximation

to convert many-qubit gates to one- and two-qubit gates[1]: Using a more naïve decomposition of the controlled-Toffoli gate (which simply computes the controlled version of every gate in the standard Toffoli decomposition) increased the size of our Toffoli-based modular multiplication circuit by 1.9x, and a similarly naïve decomposition of the controlled-controlled-Rz gate increased the size of our QFT-based modular multiplication circuit by 4.4x. We also found that gate counts (especially for the Toffoli-based circuits) are sensitive to choice of constant parameter: The QFT-based constant multiplication circuits had between 1320 and 1412 gates, while the Toffoli-based circuits had between 988 and 2264.

Comparing QFT- and AQFT-based Arithmetic Figure B.2(a) shows the effect of replacing QFT with AQFT in the QFT adder from Figure B.1(a). As expected, a decrease in precision leads to a decrease in gate count. On the other hand, our testing framework demonstrates that this also increases error (measured as absolute difference accounting for overflow, maximized over randomly-generated inputs). Random testing over a wider range of inputs suggests that dropping b bits of precision from the exact QFT adder always induces an error of at most $\pm 2^b - 1$. This suggests that the "approximate adder" is not particularly useful on its own, as it is effectively ignoring the least significant bits in the computation. However, it computes the most significant bits correctly: if the inputs are both multiples of 2^b then an approximate adder that drops b bits of precision will always produce the correct result.

A useful application of this adder is in the modulo/division circuit from Figure B.1(c), which relies on an addition subcomponent, but does not need every bit to be correctly added. Replacing the addition subcomponent with its approximate version save resources, while, our testing assures, does not introduce incorrect behavior. Figure B.2(b) shows the required resources of the approximate division/modulo circuit for different divisor/modulo values (which affects the number of iterations used in the underlying algorithm). With the maximum number of iterations (16 for divisor/modulo 1), the AQFT-based circuit uses 61.1% fewer gates than the QFT-based implementation and 79.3% fewer gates than the Toffoli-based implementation.

[1] See the decompositions we use in Section 3.1.4. Note that per our definitions, Rz is the same as U_1.

	# qubits	# gates
OQIMP (x, y const)	16	16
OQIMP TOFF (x const)	33	1739 ± 376
OQIMP QFT (x const)	16	1372 ± 26
OQIMP TOFF	33	61470
OQIMP QFT	32	25609

(a) Fixed-precision circuits for $\frac{x*y}{M}$ with $M = 5$ (16 bits)

	# qubits
OQIMP TOFF	418
OQIMP QFT	384
Quipper	6142

(b) Sine circuits (64 bits)

Figure B.3: Effects of partial evaluation

$\mathcal{O}$QIMP Oracles and Partial Evaluation As discussed in Section 4.5.2, one of the key features of $\mathcal{O}$QIMP is *partial evaluation* during compilation to $\mathcal{O}$QASM. The simplest optimization similar to partial evaluation happens for a binary operation $x := x \odot y$, where y is a constant value. Figure B.1 hints at the power of partial evaluation for this case—all constant operations (marked "const") generate circuits with significantly fewer qubits and gates. Languages like Quipper take advantage of this by producing special circuits for operations that use classically-known constant parameters.

Partial evaluation takes this one step further, pre-evaluating as much of the circuit as possible. For example, consider the fixed precision operation $\frac{x*y}{M}$ where M is constant and a natural number, and x and y are two fixed precision numbers that may be constants. This is a common pattern, appearing in many quantum oracles (recall the $\frac{8^n*x}{n!}$ in the Taylor series decomposition of sine). In Quipper, this is expression compiled to $r_1 = \frac{x}{M}; r_2 = r1 * y$. The $\mathcal{O}$QIMP compiler produces different outputs depending on whether x and y are constants. If they both are constant, $\mathcal{O}$QIMP simply assigns the result of computing $\frac{x*y}{M}$ to a quantum variable. If x is a constant, but y is not, $\mathcal{O}$QIMP evaluates $\frac{x}{M}$ classically, assigns the value to r_1, and evaluates r_2 using a constant multiplication circuit. If they are both quantum variables, $\mathcal{O}$QIMP generates a circuit to evaluate the division first and then the multiplication.

In Figure B.3(a) we show the size of the circuit generated for $\frac{x*y}{M}$ where zero, one, or both variables are classically known. It is clear that more classical variables in a program lead to a more efficient output circuit. If x and y are both constants, then only a constant assignment circuit is needed, which is a series of `X` gates. Even if only one variable is constant, it may lead to substantial savings: In this example, if x is constant, the compiler can avoid the division circuit and use a constant multiplier instead of a general multiplier. These savings quickly add up: Figure B.3(b) shows the qubit size difference between our implementation of sine and Quippers'. Both the TOFF and QFT-based circuits use fewer than 7% of the qubits used by Quipper's sine implementation.

Bibliography

[Ale+20] Thomas Alexander et al. "Qiskit pulse: programming quantum computers through the cloud with pulses". In: *Quantum Science and Technology* 5.4 (Oct. 2020). DOI: 10.1088/2058-9565/aba404.

[Amy19a] Matthew Amy. "Formal Methods in Quantum Circuit Design". PhD thesis. Waterloo, Ontario, Canada: University of Waterloo, Feb. 2019. URL: http://hdl.handle.net/10012/14480.

[Amy19b] Matthew Amy. "Towards Large-Scale Functional Verification of Universal Quantum Circuits". In: *Proceedings of the 15th International Conference on Quantum Physics and Logic (QPL), Halifax, Canada, June 3–7, 2018*. Ed. by Peter Selinger and Giulio Chiribella. Vol. 287. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Jan. 2019, pp. 1–21. DOI: 10.4204/EPTCS.287.1.

[AG20] Matthew Amy and Vlad Gheorghiu. "staq – A full-stack quantum processing toolkit". In: *Quantum Science and Technology* 5.3, 034016 (June 2020). DOI: 10.1088/2058-9565/ab9359. arXiv: 1912.06070. URL: https://github.com/softwareQinc/staq.

[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.

[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: *Programming Languages and Systems*. Ed. by Gilles Barthe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–17. ISBN: 978-3-642-19718-5.

[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.

[Bea03] Stephane Beauregard. “Circuit for Shor’s Algorithm Using 2n+3 Qubits”. In: *Quantum Info. Comput.* 3.2 (Mar. 2003), pp. 175–185. ISSN: 1533-7146.

[Ber08] Daniel J. Bernstein. “ChaCha, a variant of Salsa20”. In: The State of the Art of Stream Ciphers. Lausanne, Switzerland: ECRYPT Network of Excellence in Cryptology, Feb. 2008, pp. 273–278. URL: https://cr.yp.to/papers.html#chacha.

[Ber+19] Dominic W. Berry et al. “Qubitization of Arbitrary Basis Quantum Chemistry Leveraging Sparsity and Low Rank Factorization”. In: *Quantum* 3 (Dec. 2019), p. 208. ISSN: 2521-327X. DOI: 10.22331/q-2019-12-02-208.

[Bha+17] Karthikeyan Bhargavan et al. “Everest: Towards a Verified, Drop-in Replacement of HTTPS”. In: *2nd Summit on Advances in Programming Languages*. May 2017. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf.

[Bia20] Xiaoning Bian. *Compile Quipper quantum circuit to OpenQasm 2.0 program*. https://www.mathstat.dal.ca/~xbian/QasmTrans/. Accessed 8-July-2021. 2020.

[Bic+20] 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 ’20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 286–300. DOI: 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 (QPL), Oxford, U.K., July 15–17, 2015*. Ed. by Chris Heunen, Peter Selinger, and Jamie Vicary. Vol. 195. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Nov. 2015, pp. 71–83. DOI: 10.4204/EPTCS.195.6.

[BLH20] Anthony Bordg, Hanna Lachnitt, and Yijun He. “Isabelle Marries Dirac: a Library for Quantum Computation and Quantum Information”. In: *Archive of Formal Proofs* 2020 (Nov. 2020). URL: https://isa-afp.org/entries/Isabelle_Marries_Dirac.html.

[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. arXiv: 2009.02376.

[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.

[Cas21] Davide Castelvecchi. "Mathematicians welcome computer-assisted proof in 'grand unification' theory". In: *Nature* 595.7865 (July 2021), pp. 18–19. DOI: 10.1038/d41586-021-01627-2.

[Cha21] Chair for Design Automation at the Technical University of Munich. *MQT DDSIM – A quantum circuit simulator based on decision diagrams written in C++*. 2021. URL: https://github.com/iic-jku/ddsim.

[Cha22] Chair for Design Automation at the Technical University of Munich. *MQT QMAP - A tool for Quantum Circuit Mapping written in C++*. 2022. URL: https://github.com/cda-tum/qmap.

[Cha+21] Christophe Chareton et al. "An Automated Deductive Verification Framework for Circuit-Building Quantum Programs". In: *Programming Languages and Systems, ESOP 2021*. Ed. by Nobuko Yoshida. Vol. 12648. Lecture Notes in Computer Science. Cham: Springer International Publishing, Mar. 2021, pp. 148–177. DOI: 10.1007/978-3-030-72019-3_6.

[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.

[Cir18] Cirq Developers. *Cirq*. July 2018. DOI: 10.5281/zenodo.4062499. URL: https://github.com/quantumlib/Cirq.

[CH00] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266.

[Com20] The mathlib Community. "The lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan. 2020). DOI: 10.1145/3372885.3373824.

[Com18] Rigetti Computing. *The Quantum Processing Unit (QPU)*. 2018. URL: https://pyquil-docs.rigetti.com/en/1.9/qpu.html.

[Coq19] The Coq Development Team. *The Coq Proof Assistant, version 8.10.0*. Version 8.10.0. Oct. 2019. DOI: 10.5281/zenodo.3476303.

[Cro+17] Andrew W. Cross et al. *Open Quantum Assembly Language*. July 2017. arXiv: 1707.03429. URL: https://github.com/Qiskit/openqasm/tree/OpenQASM2.x.

[DLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. “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.

[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. DOI: 10.1098/rspa.1992.0167.

[Dev22] F* Developers. *F*: A Proof-Oriented Programming Language*. 2022. URL: https://www.fstar-lang.org/.

[Dra00] Thomas G. Draper. “Addition on a Quantum Computer”. In: *arXiv e-prints* (Aug. 2000). arXiv: quant-ph/0008033 [quant-ph].

[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. arXiv: 1901.10114 [quant-ph].

[Fey82] Richard P Feynman. “Simulating physics with computers”. In: *International journal of theoretical physics* 21.6/7 (1982), pp. 467–488. DOI: 10.1007/BF02650179.

[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. DOI: 10.1007/978-3-642-37036-6_8. URL: https://why3.lri.fr/.

[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.

[FKS20] Peng Fu, Kohei Kishida, and Peter Selinger. “Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract”. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’20. New York, NY, USA: Association for Computing Machinery, July 2020, pp. 440–453. DOI: 10.1145/3373718.3394765.

[Gel20] Alan Geller. *Introducing QIR*. Q# Blog. Sept. 2020. URL: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/.

[GE21] Craig Gidney and Martin Ekerå. “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”. In: *Quantum* 5 (Apr. 2021), p. 433. ISSN: 2521-327X. DOI: 10.22331/q-2021-04-15-433.

[Gon08] Georges Gonthier. “The Four Colour Theorem: Engineering of a Formal Proof”. In: *Computer Mathematics*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 333–333. ISBN: 978-3-540-87827-8.

[Gon+13] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 163–179. ISBN: 978-3-642-39634-2.

[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. arXiv: 0904.2557 [quant-ph].

[Gre10] Alexander S. Green. “Towards a Formally Verified Functional Quantum Programming Language”. PhD thesis. University of Nottingham, July 2010. URL: http://eprints.nottingham.ac.uk/11457/.

[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*. PLDI ’13. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 333–342. DOI: 10.1145/2491956.2462177. arXiv: 1304.3390.

[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.

[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*. STOC ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219. ISBN: 0897917855. DOI: 10.1145/237814.237866.

[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. Zurich: ETH Zurich, Dec. 2020. DOI: 10.3929/ethz-b-000468201.

[HT18] Luke Heyfron and Earl T. Campbell. “An efficient quantum compiler that reduces T count”. In: *Quantum Science and Technology* 4 (2018). DOI: 10.1088/2058-9565/aad604.

[Hie+21a] Kesha Hietala et al. “A Verified Optimizer for Quantum Circuits”. In: *Proceedings of the ACM on Programming Languages* 5.POPL, 37 (Jan. 2021). DOI: 10.1145/3434318. arXiv: 1912.02250. URL: https://github.com/inQWIRE/SQIR.

[Hie+21b] Kesha Hietala et al. "Proving Quantum Programs Correct". In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2021, 21. DOI: 10.4230/LIPIcs.ITP.2021.21. URL: https://github.com/inQWIRE/SQIR.

[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.

[Hon15] Kentaro Honda. "Analysis of Quantum Entanglement in Quantum Programs using Stabilizer Formalism". In: *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL), Oxford, U.K., July 15–17, 2015*. Ed. by Chris Heunen, Peter Selinger, and Jamie Vicary. Vol. 195. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Nov. 2015, pp. 262–272. DOI: 10.4204/EPTCS.195.19.

[IBM22] IBM. *IBM Quantum Processor Types (Falcon)*. https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/processors. Accessed: 2022-06-10. 2022.

[Inr21a] Inria. *Interfacing C with OCaml*. https://ocaml.org/manual/intfc.html. Accessed: 2021-04-09. 2021.

[Inr22] Inria. *Library Coq.Reals.Reals*. https://coq.inria.fr/library/Coq.Reals.Reals.html. Accessed: 2022-03-23. 2022.

[Inr21b] Inria, CNRS and contributors. *Program Extraction*. https://coq.inria.fr/refman/addendum/extraction.html. Accessed: 2021-09-24. 2021.

[Jav+12] Ali Javadi-Abhari et al. *Scaffold: Quantum Programming Language*. Tech. rep. TR-934-12. Princeton University, June 2012. URL: https://www.cs.princeton.edu/research/techreps/TR-934-12.

[Jav+15] Ali Javadi-Abhari et al. "ScaffCC: Scalable Compilation and Analysis of Quantum Programs". In: *Parallel Computing* 45 (June 2015), pp. 2–17. DOI: 10.1016/j.parco.2014.12.001. arXiv: 1507.01902.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. USA: Prentice-Hall, Inc., 1993. ISBN: 0130202495.

[Kv19a] 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. arXiv: 1904.04735 [quant-ph].

[Kv19b] Aleks Kissinger and John van de Wetering. "Reducing T-count with the ZX-calculus". In: *arXiv e-prints* (Mar. 2019). arXiv: 1903.10477 [quant-ph].

[Kni96] Emanuel Knill. *Conventions for Quantum Pseudocode*. Tech. rep. LA-UR-96-2724. Los Alamos National Laboratory, June 1996. DOI: 10.2172/366453.

[Lan+07] B. P. Lanyon et al. "Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement". In: *Phys. Rev. Lett.* 99 (25 Dec. 2007), p. 250505. DOI: 10.1103/PhysRevLett.99.250505.

[Le+22] Xuan-Bach Le et al. "A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic". In: *Proceedings of the ACM on Programming Languages* 6.POPL, 36 (Jan. 2022). DOI: 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.

[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.

[Li+21] Liyi Li et al. "Verified Compilation of Quantum Oracles". In: *arXiv e-prints* (Dec. 2021). arXiv: 2112.06700 [quant-ph].

[Liu+19a] Junyi Liu et al. "Formal Verification of Quantum Algorithms Using Quantum Hoare Logic". In: *Computer Aided Verification (CAV '19)*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Cham: Springer International Publishing, July 2019, pp. 187–207. DOI: 10.1007/978-3-030-25543-5_12.

[Liu+19b] Junyi Liu et al. "Quantum Hoare Logic". In: *Archive of Formal Proofs* 2019 (Mar. 2019). URL: https://isa-afp.org/entries/QHLProver.html.

[Liu+16] Tao Liu et al. "A Theorem Prover for Quantum Hoare Logic and Its Applications". In: *arXiv e-prints* (Jan. 2016). arXiv: 1601.03835 [cs.LO].

[Lu+07] Chao-Yang Lu et al. "Demonstration of a Compiled Version of Shor's Quantum Factoring Algorithm Using Photonic Qubits". In: *Phys. Rev. Lett.* 99 (25 Dec. 2007), p. 250504. DOI: 10.1103/PhysRevLett.99.250504.

[Luc+12] Erik Lucero et al. "Computing prime factors with a Josephson phase qubit quantum processor". In: *Nature Physics* 8.10 (2012), pp. 719–723. DOI: 10.1038/nphys2385.

[MS12] Igor L. Markov and Mehdi Saeedi. "Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation". In: *Quantum Info. Comput.* 12.5–6 (May 2012), pp. 361–394. ISSN: 1533-7146. arXiv: 1202.6614 [cs.ET].

[Mel20] Guillaume Melquiond. *Interval Package for Coq*. Accessed: 2022-07-20. 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. DOI: 10.1109/ISMVL.2006.35.

[Mon+16] Thomas Monz et al. "Realization of a scalable Shor algorithm". In: *Science* 351.6277 (2016), pp. 1068–1070. DOI: 10.1126/science.aad9480.

[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. 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.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge: Cambridge University Press, Dec. 2010. DOI: 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. arXiv: 1311.1074 [quant-ph].

[Par+21] Anouk Paradis et al. "Unqomp: Synthesizing Uncomputation in Quantum Circuits". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 222–236. DOI: 10.1145/3453483.3454040.

[Par+15] Zoe Paraskevopoulou et al. "Foundational Property-Based Testing". In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Cham: Springer International Publishing, 2015, pp. 325–343. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_22.

[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 ’17. New York, NY, USA: Association for Computing Machinery, Jan. 2017, pp. 846–858. DOI: 10.1145/3009837.3009894.

[Pen+22] Yuxiang Peng et al. *A Formally Certified End-to-End Implementation of Shor’s Factorization Algorithm*. Apr. 2022. arXiv: 2204.07112. URL: https://github.com/inQWIRE/SQIR/tree/main/examples/shor.

[Per08] Simon Perdrix. “Quantum Entanglement Analysis Based on Abstract Interpretation”. In: *Proceedings of the 15th International Symposium on Static Analysis*. SAS ’08. Valencia, Spain: Springer-Verlag, 2008, pp. 270–282. ISBN: 9783540691631. DOI: 10.1007/978-3-540-69166-2_18.

[Per+14] Alberto Peruzzo et al. “A variational eigenvalue solver on a photonic quantum processor”. In: *Nature Communications* 5.1 (2014), p. 4213. DOI: 10.1038/ncomms5213.

[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.

[Pie+18] Benjamin C. Pierce et al. *Software Foundations*. Version 5.6. https://softwarefoundations.cis.upenn.edu/. Electronic textbook, 2018.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. “Translation validation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166. ISBN: 978-3-540-69753-4.

[Pre18] John Preskill. “Quantum computing in the NISQ era and beyond”. In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79.

[Pyt21] Python Software Foundation. *ctypes – A foreign function library for Python*. https://docs.python.org/3/library/ctypes.html. Accessed: 2021-04-09. 2021.

[Qis17] Qiskit Community. *Qiskit: An Open-Source Framework for Quantum Computing*. Mar. 2017. DOI: 10.5281/zenodo.2562110. URL: https://github.com/Qiskit/qiskit.

[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, Nov. 2018. URL: https://repository.upenn.edu/edissertations/3175.

[Ran19] Robert Rand. *Verified Quantum Computing*. Accessed: 2022-07-20. 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 of the 14th International Conference on Quantum Physics and Logic (QPL), Nijmegen, the Netherlands, July 3–7, 2017*. Ed. by Bob Coecke and Aleks Kissinger. Vol. 266. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Feb. 2018, pp. 119–132. DOI: 10.4204/EPTCS.266.8.

[Ran+19] Robert Rand et al. "ReQWIRE: Reasoning about Reversible Quantum Circuits". In: *Proceedings of the 15th International Conference on Quantum Physics and Logic (QPL), Halifax, Canada, June 3–7, 2018*. Ed. by Peter Selinger and Giulio Chiribella. Vol. 287. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Jan. 2019, pp. 299–312. DOI: 10.4204/EPTCS.287.17.

[Ran+21] Robert Rand et al. "Gottesman Types for Quantum Programs". In: *Proceedings of the 17th International Conference on Quantum Physics and Logic (QPL), Paris, France, June 2–6, 2020*. Ed. by Benoît Valiron et al. Vol. 340. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW, Australia: Open Publishing Association, Sept. 2021, pp. 279–290. DOI: 10.4204/EPTCS.340.14.

[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.

[Rin+19] Talia Ringer et al. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends in Programming Languages* 5.2-3 (2019), pp. 102–281. ISSN: 2325-1107. DOI: 10.1561/2500000045.

[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.

[Sel04a] Peter Selinger. "A Brief Survey of Quantum Programming Languages". In: *Proceedings of the 7th International Symposium on Functional and Logic Programming*. Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Vol. 2998. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Apr. 2004, pp. 1–6. DOI: 10.1007/978-3-540-24754-8_1.

[Sel04b] Peter Selinger. “Towards a Quantum Programming Language”. In: *Mathematical Structures in Computer Science* 14.4 (Aug. 2004), pp. 527–586. DOI: 10.1017/S0960129504004256.

[SV09] Peter Selinger and Benoît Valiron. “Quantum Lambda Calculus”. In: *Semantic Techniques in Quantum Computation*. Ed. by Simon J. Gay and Ian Mackie. Cambridge: Cambridge University Press, Nov. 2009, pp. 135–172. DOI: 10.1017/CBO9781139193313.005. URL: https://www.mscs.dal.ca/~selinger/papers/qlambdabook.pdf.

[Sho97] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM J. Comput.* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397. DOI: 10.1137/S0097539795293172.

[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. DOI: 10.1137/S0097539796298637.

[Sin+22] Kartik Singhal et al. “Q# as a Quantum Algorithmic Language”. In: *Proceedings of the 19th International Conference on Quantum Physics and Logic (QPL), Oxford, U.K., June 27–July 1, 2022*. To appear. Open Publishing Association, June 2022. arXiv: 2206.03532. URL: https://ks.cs.uchicago.edu/publication/q-algol/.

[Siv+20] Seyon Sivarajah et al. “t|ket⟩: a retargetable compiler for NISQ devices”. In: *Quantum Science and Technology* 6.1, 014003 (Nov. 2020). DOI: 10.1088/2058-9565/ab8e92. arXiv: 2003.10611. URL: https://github.com/CQCL/pytket.

[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. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 579–588. ISBN: 9781450366694. DOI: 10.1145/3307650.3322262.

[SCZ16] Robert S. Smith, Michael J. Curtis, and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. Aug. 2016. arXiv: 1608.03355. URL: https://github.com/rigetti/quil.

[SHT18] Damian S. Steiger, Thomas Häner, and Matthias Troyer. “ProjectQ: An Open Source Software Framework for Quantum Computing”. In: *Quantum* 2, 49 (Jan. 2018). DOI: 10.22331/q-2018-01-31-49.

[Svo+18] Krysta M. 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*. RWDSL ’18. New York, NY, USA: Association for Computing Machinery, Feb. 2018, 7. DOI: 10.1145/3183895.3183901. arXiv: 1803.00652.

[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.

[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.

[Tao+22] Runzhou Tao et al. "Giallar: Push-Button Verification for the Qiskit Quantum Compiler". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 641–656. DOI: 10.1145/3519939.3523431. arXiv: 2205.00661.

[The22] The INQWIRE Developers. *INQWIRE QuantumLib*. Version 1.0.0. Jan. 2022. URL: https://github.com/inQWIRE/QuantumLib.

[Van+01] Lieven MK Vandersypen et al. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance". In: *Nature* 414.6866 (2001), pp. 883–887. DOI: 10.1038/414883a.

[VW04] Farrokh Vatan and Colin Williams. "Optimal quantum circuits for general two-qubit gates". In: *Phys. Rev. A* 69 (3 Mar. 2004), p. 032315. DOI: 10.1103/PhysRevA.69.032315.

[Wil+08] R. Wille et al. "RevLib: An Online Resource for Reversible Functions and Reversible Circuits". In: *Int'l Symp. on Multi-Valued Logic*. RevLib is available at http://www.revlib.org. 2008, pp. 220–225.

[WBZ19] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. "Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations". In: *Design Automation Conference*. 2019. arXiv: 1907.02026 [quant-ph].

[Xu+22] Mingkuan Xu et al. "Quartz: Superoptimization of Quantum Circuits". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 625–640. DOI: 10.1145/3519939.3523433. arXiv: 2204.09033. URL: https://github.com/quantum-compiler/quartz.

[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.

[Yin12] Mingsheng Ying. "Floyd–Hoare Logic for Quantum Programs". In: *ACM Transactions on Programming Languages and Systems* 33.6, 19 (Jan. 2012). DOI: 10.1145/2049706.2049708.

[YMC22] Charles Yuan, Christopher McNally, and Michael Carbin. "Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs". In: *Proceedings of the ACM on Programming Languages* 6.POPL, 30 (Jan. 2022). DOI: 10.1145/3498691. arXiv: 2205.02287. URL: https://github.com/psg-mit/twist-popl22.

[Zho+21] Li Zhou et al. "A Quantum Interpretation of Bunched Logic & Quantum Separation Logic". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '21. Los Alamitos, CA, USA: IEEE Computer Society, July 2021, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470673. arXiv: 2102.00329.

[Zin+17] Jean-Karim Zinzindohoué et al. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1789–1806. ISBN: 9781450349468. DOI: 10.1145/3133956.3134043.

[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].