

FORMALLY VERIFIED QUANTUM PROGRAMMING

Robert Rand

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

Steve Zdancewic
Professor of Computer and Information Science

Graduate Group Chairperson

Rajeev Alur
Professor of Computer and Information Science

Dissertation Committee

Stephanie Weirich, Professor of Computer and Information Science, Chair

Val Tannen, Professor of Computer and Information Science

Benjamin Pierce, Professor of Computer and Information Science

Prakash Panangaden, Professor of Computer Science, McGill University

Acknowledgments

A lot of people helped me on the road to this dissertation, none more than my advisor, Steve Zdancewic. From the moment I wandered into his office and asked if we could do research on probability and logic (his response: “Sure!”), to the moment he emailed me and Jennifer Paykin asking if we wanted to work on a quantum computing project (“Sure!”), to the moment I submitted this dissertation, Steve has been endlessly enthusiastic, generous with his time, and willing to explore entirely new horizons.

In a similar vein, I have to thank my friend and collaborator Jennifer Paykin, who jumped into the new and exciting world of quantum programming languages with me. Doing research is an entirely different, and better, experience with a close collaborator, especially one as talented and tireless as Jennifer. *QWIRE*, and therefore this thesis, could not possibly have existed without her.

Thanks to Mike Mislove and the other members of the “Semantics, Formal Reasoning, and Tools for Quantum Programming” research initiative for introducing me to quantum computing and helping me acclimate to the field. Relatedly, thanks to the MFPS and QPL communities, which taught me a great deal and always made me feel welcome. Thanks especially to Peter Selinger, who blazed the path that I tried to follow and happily helped me along it, and Prakash Panangaden, who lent me his knowledge of quantum mechanics and graciously agreed to serve on my thesis committee.

Thanks to Andy Gordon, who hosted me for a summer at Microsoft Research Cambridge. Thanks to my collaborators there: Neil Toronto, Cecily Morrison, Claudio Russo, Simon Peyton-Jones, Abigail Sellen, Felienne Hermans, Advait Sarkar and Rupert Horlick. Also, thanks to Tony Hoare, Cédric Fournet and Georges Gonthier for intellectually stimulating lunch meetings that often segued into extensive discussions of formal verification.

Thanks to PLClub. The Penn Programming Languages group is not just a research group: it is a meeting place for people who love programming languages and are deeply invested in one another’s success. Thanks especially to Stephanie Weirich, Benjamin Pierce, Rajeev Alur, Mayur Naik and Val Tannen for their criticism, encouragement and advice. And thanks to Antal Spector-Zabusky, Leonidas Lampropoulos, Arthur Azevedo de Amorim, Peter-Michael Osera, Vilhelm Sjöberg, Richard Eisenberg, Brent Yorgey, Justin Hsu, Kenny Foner, Dmitri Garbuzov, Yannick Zakowski, Christine Rizkallah, Joachim Breitner, William Mansky, Maxime Dénès, Cătălin Hrițcu, Emilio

Jesús Gallego Arias and Benoît Valiron for their generous help and camaraderie.

Everyone who read this thesis, in part or in full, has my lasting gratitude. That's Steve and Jennifer again, and my committee, Stephanie, Benjamin, Val and Prakash, as well Julien Ross, Yannick Zakowski, Alex Burka and Rivka Cohen. You are all my heroes.

I'd like to end with a tribute to my first advisor and mentor at Penn, the late Dr. Ben Taskar. I can vividly remember receiving a text message from out of the blue saying Ben had passed away. In a daze, I searched the internet for confirmation, coming upon a lone tweet from Ben's own mentor, the legendary Andrew Ng: "RIP Ben Taskar. Machine learning just lost a star." The night sky may be full of lost stars but we still benefit from their light.

ABSTRACT

FORMALLY VERIFIED QUANTUM PROGRAMMING

Robert Rand
Steve Zdancewic

The field of quantum mechanics predates computer science by at least ten years, the time between the publication of the Schrödinger equation and the Church-Turing thesis. It took another fifty years for Feynman to recognize that harnessing quantum mechanics is necessary to efficiently simulate physics and for David Deutsch to propose the quantum Turing machine. After thirty more years, we are finally getting close to the first general-purpose quantum computers based upon prototypes by IBM, Intel, Google and others.

While physicists and engineers have worked on building scalable quantum computers, theoretical computer scientists have made their own advances. Complexity theorists introduced quantum complexity classes like BQP and QMA; Shor and Grover developed their famous algorithms for factoring and unstructured search. Programming languages researchers pursued two main research directions: Small-scale languages like QPL and the quantum λ -calculi for reasoning about quantum computation and large-scale languages like Quipper and Q# for industrial-scale quantum software development. This thesis aims to unify these two threads while adding a third one: *formal verification*.

We argue that quantum programs demand machine-checkable proofs of correctness. We justify this on the basis of the complexity of programs manipulating quantum states, the expense of running quantum programs, and the inapplicability of traditional debugging techniques to programs whose states cannot be examined. We further argue that the existing mathematical models of quantum computation make this an easier task than one could reasonably expect. In light of these observations we introduce *QWIRE*, a tool for writing verifiable, large scale quantum programs.

QWIRE is not merely a language for writing and verifying quantum circuits: it is a *verified* circuit description language. This means that the semantics of *QWIRE* circuits are verified in the Coq proof assistant. We also implement verified abstractions, like ancilla management and reversible circuit compilation. Finally, we turn *QWIRE* and Coq's abilities outwards, towards verifying popular quantum algorithms like quantum teleportation. We argue that this tool provides a solid foundation for research into quantum programming languages and formal verification going forward.

Contents

List of Figures	ix
List of Tables	x
1 The Big Picture	1
1.1 Motivation	1
1.2 Thesis Statement	4
1.3 Outline	4
2 An Introduction to Quantum Computing	7
2.1 Qubits	7
2.2 Example: Quantum Teleportation	11
2.3 Quantum Circuits	12
2.4 Quantum States as Vectors	13
2.4.1 Unitary transformations	14
2.5 Mixed States as Density Matrices	17
2.6 Additional Material	20
3 History	21
3.1 Circuits, QRAM and Classical Control	21
3.2 QCL: A General Purpose Quantum Language	22
3.3 QPL: Semantics for Quantum Programming	23
3.4 Linearity and the Quantum Lambda Calculi	25
3.5 The Quantum IO Monad	27
3.6 Quipper and the Proto-Quippers	27
3.7 Liquid, REVS and Q#	29
3.8 The World of Quantum Programming	30
3.9 Models of Quantum Computation	30
3.10 Formal Verification	31
3.10.1 Quantum Logics	32
3.10.2 Mechanized Verification	32

4	Qwire in Theory	34
4.1	Introduction	34
4.1.1	The Best of Both Worlds: QWIRE	35
4.1.2	Chapter Outline	36
4.2	QWIRE by Example	37
4.3	The QWIRE Circuit Language	40
4.3.1	Circuit Language	40
4.3.2	Host Language	41
4.3.3	Static Semantics	43
4.4	Operational Semantics: Circuit Normalization	45
4.4.1	Type Safety	49
4.5	Denotational Semantics	50
4.5.1	Operational Behavior of <i>run</i>	53
4.6	A Categorical Semantics for QWIRE	53
4.7	Dependent Types	54
4.8	Summary	55
5	Qwire in Practice	57
5.1	Circuits in Coq	57
5.2	Typing QWIRE	59
5.3	De Bruijn Circuits	61
5.4	Matrices and Semantics	63
5.4.1	Complex Numbers	63
5.4.2	The Matrix Library	64
5.4.3	Density Matrices	65
5.5	Denotation of QWIRE	65
5.6	Functional Notations	69
6	Verifying Qwire	73
6.1	Predicates and Preservation	73
6.1.1	Well-Formed Matrices	73
6.1.2	Unitarity	74
6.1.3	Pure and Mixed States	75
6.1.4	Superoperator Correctness	76
6.2	Towards Compositionality	77
6.3	Future Work on QWIRE’s Metatheory	78
7	Verifying Quantum Programs	80
7.1	Verifying Matrices	80
7.1.1	Tossing Coins	80
7.1.2	Teleport	80
7.1.3	Deutsch’s Algorithm	82
7.2	Matrix Families and Induction	83

7.2.1	Many Coins	83
7.3	Algebraic Reasoning about Circuits	84
7.3.1	A Unitary and Its Adjoint	84
7.4	Equational Rewriting	85
8	Reversibility	87
8.1	Ancillae and Assertions	87
8.2	<i>Safe</i> and <i>Unsafe</i> Semantics	90
8.3	Syntactically Valid Ancillae	92
8.4	Compiling Oracles	94
8.5	Quantum Arithmetic in \mathcal{Q} WIRE	97
8.6	Next Steps for Reversible Computation	98
9	Automation	100
9.1	Typechecking Circuits	100
9.1.1	Monoid	104
9.1.2	Validate	105
9.2	Arithmetic	105
9.3	Linear Algebra	106
9.3.1	Matrix Properties	106
9.3.2	Solving Matrix Equalities	108
9.4	Denoting Circuits	109
9.5	Tactics Reference	110
10	Details: \mathcal{Q}wire Within	113
10.1	An outline of \mathcal{Q} WIRE	113
10.2	Matters of Trust	116
11	Open Wires and Loose Ends	119
11.1	Computing in \mathcal{Q} WIRE	119
11.2	Connecting \mathcal{Q} WIRE	121
11.3	The Future of \mathcal{Q} WIRE	121
11.3.1	Verified Optimization and Compilation	122
11.3.2	Error Awareness and Error Correction	123
11.3.3	Verified Algorithms and Cryptography	123
A	Solutions to Exercises	125
B	\mathcal{Q}wire in Theory: Proofs	130
B.1	Type safety and normalization	130
B.2	Soundness of denotational semantics	134

C	Qwire Documentation	137
C.1	Prelim.v	137
C.2	Monad.v	137
C.3	Monoid.v	137
C.4	Matrix.v	137
C.5	Quantum.v	137
C.6	Contexts.v	137
C.7	HOASCircuits.v	138
C.8	DBCircuits.v	138
C.9	Denotation.v	138
C.10	HOASLib.v	138
C.11	SemanticLib.v	138
C.12	HOASExamples.v	138
C.13	HOASProofs.v	138
C.14	Equations.v	138
C.15	Composition.v	138
C.16	Ancilla.v	139
C.17	Symmetric.v	139
C.18	Oracles.v	139
	Bibliography	140

List of Figures

2.1	A quantum teleportation circuit	13
3.1	A QPL program for tossing a quantum coin	24
3.2	A quantum teleportation circuit	26
4.1	A \mathcal{Q} WIRE implementation of quantum teleportation without dynamic lifting.	38
4.2	Typing rules for \mathcal{Q} WIRE.	44
4.3	Operational semantics of concrete circuits.	48
4.4	Denotational semantics of circuits.	52
5.1	Rearranged Typing Rules	60
5.2	A zipper for denoting controlled unitaries	68
5.3	The Deutsch-Jozsa circuit	71
5.4	Flipping coins with dynamic lifting	72
7.1	The superdense coding protocol with boolean inputs	81
7.2	Tossing n coins	83
8.1	Quantum oracles implementing the boolean \wedge and \vee . The \oplus gates represent negation, and \bullet represents control.	87
8.2	An non-unitary quantum oracle for $(a \vee b) \wedge (c \vee d)$	88
8.3	A unitary quantum oracle for $(a \vee b) \wedge (c \vee d)$ with ancillae	88
8.4	Compiling $b_1 \wedge b_2$ on 3 qubits	96
8.5	A quantum adder	97
9.1	Typing a simple \mathcal{Q} WIRE program	100
9.2	Manually typechecking <code>cnot12</code>	101
9.3	Typechecking <code>cnot12</code> using <code>evars</code>	102

List of Tables

8.1	Reversibility assumptions	90
9.1	A summary of <i>QWIRE</i> tactics	112
9.2	A summary of <i>QWIRE</i> databases	112
10.1	A brief summary of the <i>QWIRE</i> Coq development	114
10.2	The claims of <i>QWIRE</i>	117
11.1	Space-efficient constructions of Shor’s algorithm over time	122

Chapter 1

The Big Picture

1.1 Motivation

In 1936, Alonzo Church wrote a programming language for a machine that didn't exist. His lambda calculus (Church, 1936a) influenced many of the programming languages that would emerge with the advent of the programmable computer in the 1940s. This, of course, wasn't Church's goal: He had in his sights Hilbert's *Entscheidungsproblem*, which asked whether one could write an algorithm to prove arbitrary statements in first-order logic. Church (1936b) and Turing (1937) both answered this problem in the negative and, in doing so, proposed "universal" models of computation that fundamentally ignored the scientific revolution of ten years prior.

While Church and Turing's models of computation were able to express which problems were computable in theory, they were woefully ill-equipped to describe which problems were solvable in practice, even using the language of complexity theory. Richard Feynman first recognized this in 1982 (Feynman, 1982), pointing out that a Turing machine was seemingly incapable of efficiently simulating physics, since physics obeys the mathematically complex laws of quantum mechanics. This shortcoming was both surprising and disappointing: You would expect that we could simulate basic physical processes on our so-called universal computers.

David Deutsch (1985) addressed this problem three years later. He proposed a "Quantum Turing Machine" that could serve as the basis for quantum complexity theory. Complexity theorists ran with this idea, introducing the classes BQP (Bounded-Error Quantum Polynomial-Time) (Bernstein and Vazirani, 1997) and QMA (Quantum Merlin-Arthur) (Watrous, 2000), and studying their relationships to the classical and probabilistic complexity classes. These studies led to breakthrough quantum algorithms, including Shor's algorithm (1994) for computing prime factors and discrete logarithms, and Grover's algorithm (1996) for unstructured search. While Grover's algorithm led only to a quadratic speedup, Shor's algorithm factored numbers in polynomial time, suggesting that BQP (the quantum analogue of polynomial time, or P) is strictly larger than P. This suggestion was reinforced by subsequent results,

such as Raz and Avishay’s (2018) proof of an oracle separation between BQP and the polynomial hierarchy. By now, Quantum Complexity Theory is featured in standard complexity theory textbooks (for example, Arora and Barak (2009)) and even books aimed at a popular audience (Aaronson, 2013).

While the complexity theorists and algorithms designers have done an impressive job of telling future quantum programmers *what* to program, programming languages researchers have lagged behind in telling them *how* to program. This presents a dilemma. When general-purpose quantum computers see the light of day (soon, judging by recent efforts at technology giants Google¹, IBM², Intel³, Microsoft⁴ and Alibaba⁵, as well as the start-up Rigetti Computing⁶), people will program them. And if they are forced to invent ad-hoc programming languages to address the challenges of the moment, we risk introducing design flaws that will plague generations of quantum programmers.

Reassuringly, programming languages research has a head start on actual quantum computers. Several important paradigms have gained traction within the quantum programming languages community, including the *QRAM* model (Knill, 1996), in which a quantum computer is used as a sort of oracle by a connected classical computer, and Selinger’s (2004a) refinement called *quantum data, classical control*, in which data may be quantum, but a program’s control flow is always classical. A special case of Selinger’s approach is the *quantum circuit model*, in which a classical computer constructs quantum circuits and sends them to a quantum computer for execution.

The twenty-two years since Knill wrote his “Conventions for Quantum Pseudocode” (Knill, 1996) have seen the proliferation of quantum programming languages, which we can loosely classify into two groups:

1. Academic programming languages for reasoning about quantum programs
2. High level languages for implementing complex quantum programs on future quantum devices

In the first category we have languages like QPL (Selinger, 2004a), λ_q (van Tonder, 2004), and the Quantum Lambda Calculus (Selinger and Valiron, 2009). QPL

¹<https://www.technologyreview.com/s/604242/googles-new-chip-is-a-stepping-stone-to-quantum-computing-supremacy>

²<https://www.technologyreview.com/s/607887/ibm-nudges-ahead-in-the-race-for-quantum-supremacy>

³<https://www.technologyreview.com/s/603165/intel-bets-it-can-turn-everyday-silicon-into-quantum-computings-wonder-material>

⁴<http://www.nature.com/news/inside-microsoft-s-quest-for-a-topological-quantum-computer-1.20774>

⁵<https://medium.com/syncedreview/alibaba-launches-11-qubit-quantum-computing-cloud-service-ad7f8e02cc8>

⁶<https://www.forbes.com/sites/alexknapp/2018/09/07/rigetti-computing-takes-small-step-toward-cloud-services-in-big-leap-for-quantum-computing>

was one of the first proposed quantum programming languages, with a denotational semantics given in terms of string diagrams and density matrices. This language was used as a model language in a number of subsequent works, such as Kakutani’s (2009) Hoare-like logic QHL and D’Hondt and Panangaden’s (2006) Quantum Weakest Preconditions. Van Tonder’s λ_q and Selinger and Valiron’s Quantum Lambda Calculus are both adaptations of Church’s lambda calculus to a quantum setting. λ_q focuses on expressivity and sketches a proof of equivalence to Yao’s (1993) quantum circuit model and thereby to the quantum Turing machine. Selinger and Valiron’s calculus, by contrast, focuses on the language’s linear type system, which enforces the *no-cloning theorem* of quantum mechanics by guaranteeing that each qubit is used exactly once. None of these languages are designed for practical quantum computing, however.

By contrast, quantum circuit languages like QCL (Ömer, 2000, 2003), Quipper (Green et al., 2013a,b), Liquid (Wecker and Svore, 2014), and Q# (Svore et al., 2018) are designed for efficient, general-purpose quantum computing. QCL is a C-like language with support for both classical and quantum computing, where Quipper and Liquid are embedded in Haskell and F#, respectively, and are capable of using these languages’ features and libraries to construct complex families of quantum circuits. Q# (Svore et al., 2018) is a recent standalone successor to Liquid, meant to reduce reliance on F# and provide a programming environment targeted exclusively at quantum computing. All of these languages provide optimized compilation to low-level circuits and can simulate quantum computation. Unfortunately, they lack important features of QPL and the Quantum Lambda Calculus, such as denotational semantics and type systems that guarantee circuits are well-formed.

Given the cost and expressive power of quantum computing, we need a programming language that fits in both of these categories. It should take advantage of existing programming languages research, which strongly suggests using the quantum circuit model adopted by Quipper and Liquid, as well as their abstractions and optimizations. It must ensure that any quantum program sent to the quantum computer represents a valid quantum mechanical operation, as guaranteed by the Quantum Lambda Calculus’ linear type system. And finally, it must be provably safe and easy to reason about, in the style of QPL.

This last requirement is partly born of necessity: Quantum programs are tremendously difficult to understand and implement, almost guaranteeing that they will have bugs. And traditional approaches to debugging will not help us: We cannot set breakpoints and look at our qubits without collapsing the quantum state. Even techniques like unit tests and random testing will be impossible to run on classical machines and too expensive to run on quantum computers – and failed tests are unlikely to be informative. But this requirement is also born of opportunity: The underlying mathematics of quantum computing is well understood, and it is easier to model mathematically than classical programs with probabilistic operations. This lowers the cost of formal verification, and it can make the most powerful form of

program specification into the most convenient as well.

1.2 Thesis Statement

*Quantum programming is not only amenable to formal verification:
it demands it.*

The overarching goal of this thesis is to write and verify quantum programs together. Towards that end, we introduce a quantum programming language called `QWIRE` and embed it inside the Coq proof assistant. We give it a linear type system to ensure that it obeys the laws of quantum mechanics and a denotational semantics to prove that programs behave as desired. We also formalize the metatheory of `QWIRE` to ensure that the language itself is well-behaved and only supports physically realizable computation. We use `QWIRE`'s rich type theory and semantic guarantees to implement features that could not be safely implemented in other languages, from circuit families to assertive terminations. And, naturally, we use `QWIRE` to verify existing algorithms from the quantum computing literature.

1.3 Outline

We begin by introducing the basics of quantum computing (Chapter 2). We then take a look at the history of quantum programming languages, from van Tonder's (2003) quantum lambda calculus to powerful modern languages like Quipper (Green et al., 2013a) and Q# (Svore et al., 2018). That brings us to the core of the thesis, the quantum circuit language `QWIRE`.

`QWIRE` is an embedded circuit generation language for quantum computing. In Chapter 4, we give `QWIRE` a denotational semantics in terms of density matrices, a linear type system that guarantees circuits are well formed, and a *dynamic lifting* operation for communication between a classical and a quantum computer.

We embed `QWIRE` inside the Coq proof assistant (Chapter 5), tying its variables to those of Coq's programming language Gallina, and implementing a typechecking algorithm as a Coq tactic. We also provide a direct translation to `QWIRE`'s semantics in terms of density matrices, allowing us to prove properties of generated circuits. These density matrices rely on our own libraries for linear algebra and quantum information theory, together with Coquelicot's (Boldo et al., 2015) complex number library. We explore `QWIRE`'s metatheory in Chapter 6, showing that well-typed circuits correspond to quantum-mechanically sound functions on quantum states. In chapter Chapter 7, we use the resulting semantics to prove the properties of a number of quantum programs, including quantum teleportation, Deutsch's algorithm (1985), and a variety of coin flipping protocols.

While these chapters assume familiarity with the Coq proof assistant, we hope that they will be accessible to most readers with some knowledge of functional pro-

programming and formal verification. For the reader who would like to understand the Coq programming language better, we recommend our online tutorial (Rand and de Amorim, 2016) or the more comprehensive “Software Foundations,” which also delves deeply into programming languages and type theory, both of which will aid the reader in digesting this work.

With *QWIRE* in hand, we can begin to explore a type-safe approach to high-level quantum programming (Chapter 8, based on Rand et al. (2018a)). We begin by implementing some of the core features of Green *et al.*’s Quipper language using dependent types. Quipper makes heavy use of *ancillas*, temporary qubits that are initialized in some state and discarded in the same state, at least according to the assertions that accompany the discard operation. Unfortunately, Quipper has no way of ensuring that these assertions are true. Worse, the compiler relies upon these assertions, so when an assertion is false, the compiled program is likely to misbehave. We include assertions in the Coq implementation of *QWIRE* and require the programmer to prove them correct before the program will compile. We also provide an assertion-using compiler from boolean expressions to *QWIRE* circuits, and we prove that the generated circuits compute the same functions as the provided expressions.

All of this work requires a significant amount of reasoning, from proving that circuits are well typed to showing that they compute the desired function. As we built *QWIRE*, we noticed which tasks demanded a lot of our own time and attempted to automate them, for our benefit and that of future users. Chapter 9 discusses the various forms of automation present in *QWIRE*, from the monoidal solver and disjointness checker used by our linear typechecker to the range of tactics used in proving matrix equalities.

Given that most of this dissertation discusses the ideas that underlie *QWIRE*, in Chapter 10 we pause the high-level exposition and delve into the Coq development itself. There we discuss the assumptions that we use in *QWIRE* and attempt to justify them. We also include every proof and assumption in this thesis in Appendix C, which we urge the reader to consult, especially if an English-language description proves ambiguous. We also encourage the reader to step through the Coq development itself, available at <https://github.com/inQWIRE/QWIRE>.

Some of this work appears in two prior papers by Paykin, Rand, and Zdancewic: Paykin et al. (2017) and Rand et al. (2017). The first paper, which introduces the *QWIRE* language and its type system and denotational semantics, forms the basis for Chapter 4. The second paper contains early details of *QWIRE*’s implementation, along with proofs of some basic circuit equalities. Many of the more interesting *QWIRE* proofs, particularly those about its metatheory, have yet to be published.

We developed *QWIRE* together with Jennifer Paykin, Steve Zdancewic, and Dong-Ho Lee at the University of Pennsylvania. Though every collaborator was involved in multiple aspects of *QWIRE*’s development, we can briefly describe the focuses of each author. This author’s main contributions were towards the semantics of *QWIRE*, its implementation, and its applications towards formal verification, as explored in this

thesis. Paykin designed the language itself and its linear type system, as part of a line of investigation into linear types, including the linear/producer/consumer model of classical linear logic (Paykin and Zdancewic, 2016), the linearity monad (Paykin and Zdancewic, 2017), and her dissertation (Paykin, 2018). Zdancewic, our advisor, helped flesh out most of the ideas underlying *QWIRE*, primarily when they consisted of sketches on whiteboards. Finally, Lee contributed to the study of reversible computing in *QWIRE* (Rand et al., 2018a), providing the quantum adder discussed in Chapter 8, as well as a compiler from *QWIRE* to lower level “quantum assembly” languages that we discuss in Chapter 11. That chapter also discusses future directions for *QWIRE*, from verified optimizations to quantum error correction.

Chapter 2

An Introduction to Quantum Computing

In this chapter, we introduce the basics of quantum computation, starting with simple qubits and proceeding to concepts like superposition, entanglement, and unitary transformations. We only assume knowledge of basic linear algebra and try to elide any concepts not directly relevant to this dissertation, particularly those related to the physics of quantum computation. To assist the unfamiliar reader, we have included a number of exercises, which should help them internalize ideas as we present them. Solutions to these exercises are provided in Appendix A.

2.1 Qubits

Qubits, a pun on the ancient unit of measure, “cubit,” are the quantum analogue of bits. While qubits can take on a variety of configurations, called *states*, the two simplest correspond to the binary 0 and 1 and are written $|0\rangle$ and $|1\rangle$. We call these the *basis states*. They may represent different amounts of charge on a wire, or base particles rotating clockwise versus counterclockwise—as in classical computing, the physical implementation of qubits does not concern us.

Conveniently, when describing operations on qubits, we can describe their behavior on basis states and then lift this behavior to more complicated quantum states. One common “classical” operation on qubits is Wolfgang Pauli’s X (or *NOT*) operator, which behaves like classical negation:

$$\begin{aligned}X|0\rangle &= |1\rangle \\X|1\rangle &= |0\rangle\end{aligned}$$

We concatenate qubits using the tensor operator \otimes . For instance, a $|1\rangle$ qubit next to a $|0\rangle$ qubit can be written as $|1\rangle \otimes |0\rangle$, or $|10\rangle$ for short. Note that these qubits are *ordered*: There is a first qubit (the $|1\rangle$) and a second qubit (the $|0\rangle$). We can now

define the controlled-not (or *CNOT*) operator on two qubit states:

$$\begin{aligned}CNOT |00\rangle &= |00\rangle \\CNOT |01\rangle &= |01\rangle \\CNOT |10\rangle &= |11\rangle \\CNOT |11\rangle &= |10\rangle\end{aligned}$$

When the first qubit is $|1\rangle$, the second qubit is negated; otherwise, both qubits are unchanged. The meaning of “controlled-not” should therefore be apparent: The first qubit *controls* whether the second is negated or not. This notion of control can be generalized as follows: For any operator f on k qubits, a “controlled” f is an operator on $k+1$ qubits that applies f if the first qubit is $|1\rangle$ and otherwise applies the identity function. Note that if we iterate the “control” operation, the function will be applied only if all of the controlling qubits (or “controls”) are $|1\rangle$. The controls themselves are never altered by this operation.

Superposition We now introduce our first “quantum” operation, the Hadamard H , on one-qubit quantum states:

$$\begin{aligned}H |0\rangle &= \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle \\H |1\rangle &= \frac{1}{\sqrt{2}} |0\rangle - \frac{1}{\sqrt{2}} |1\rangle\end{aligned}$$

The scaling factors to the left of $|0\rangle$ and $|1\rangle$ are complex numbers called *amplitudes*, and the expression $\alpha |0\rangle + \beta |1\rangle$ represents a *superposition*, a weighted combination of $|0\rangle$ and $|1\rangle$. This has no analogue in classical physics, but for our purposes “a weighted combination” will suffice. The + symbol here behaves like addition: It is commutative and associative and obeys distributive laws, with both scaling and tensor being a form of multiplication. Additionally, $|\psi\rangle$ and $-|\psi\rangle$ cancel each other out, as we will see in the following example.

Applying an operator to a qubit in a superposition of $|0\rangle$ and $|1\rangle$ is the same as applying it to each of the basis qubits, as in the following example:

$$\begin{aligned}
H\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) &= \frac{1}{\sqrt{2}}H|0\rangle + \frac{1}{\sqrt{2}}H|1\rangle \\
&= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) + \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) \\
&= \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle + \frac{1}{2}|0\rangle - \frac{1}{2}|1\rangle \\
&= \frac{1}{2}|0\rangle + \frac{1}{2}|0\rangle \\
&= |0\rangle
\end{aligned}$$

We see that the Hadamard operator can take a non-basis state to a basis state (this will be true of all our operators).

Exercise 1. Show that H is its own inverse.

Entanglement Things become more interesting once we combine H and $CNOT$ operators.

Consider a $CNOT$ applied to two qubits: $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $|0\rangle$:

$$\begin{aligned}
CNOT\left[\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes |0\rangle\right] &= CNOT\left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle\right) \\
&= \frac{1}{\sqrt{2}}(CNOT|00\rangle) + \frac{1}{\sqrt{2}}(CNOT|10\rangle) \\
&= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle
\end{aligned}$$

In this new state, the two qubits are intertwined with one another: It is no longer possible to express this state as the tensor product of two distinct one-qubit states. We call this state of affairs *entanglement*, and it is closely related to the *dependence* of two random variables in probability theory.

Measurement To see where probability enters the picture, we have to introduce the measurement operator, $meas$, which, unlike X , H and $CNOT$, can only be expressed as a function on the whole qubit, not its components:

$$meas(\alpha|0\rangle + \beta|1\rangle) = \begin{cases} |0\rangle & \text{with probability } |\alpha|^2 \\ |1\rangle & \text{with probability } |\beta|^2 \end{cases}$$

The *modulus* of a complex number $|a + bi|$ is $\sqrt{a^2 + b^2}$, so whenever the imaginary component is zero, taking the modulus squared is the same as squaring the number.

Note that in any single qubit state $\alpha|0\rangle + \beta|1\rangle$, we have $|\alpha|^2 + |\beta|^2 = 1$, allowing us to translate amplitudes into probabilities.

We can see that measuring $|0\rangle$ always yields $|0\rangle$ and similarly for $|1\rangle$, since the basis state $|0\rangle$ is really $1|0\rangle + 0|1\rangle$. Measuring $H|0\rangle$ or $H|1\rangle$ will yield each basis state with one-half probability, since $\frac{1}{\sqrt{2}}^2 = (-\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$. Measurement is idempotent: Once we have measured a qubit, it enters the basis state $|0\rangle$ or $|1\rangle$, so measuring it a second time has no impact.

We can easily extend this notion of measurement to a multi-qubit system. If we have the state $\sum_i \alpha_i |i\rangle$, where $|i\rangle$ ranges over the basis states $|0\dots 0\rangle$ through $|1\dots 1\rangle$, the probability that measurement returns $|i\rangle$ is $|\alpha_i|^2$.

What if we want to measure one qubit in a multiple qubit system? Let $\sum_i \alpha_i |i\rangle$ represent the part of the state in which the qubit to be measured is $|0\rangle$ and $\sum_j \beta_j |j\rangle$ represent the part in which the qubit is $|1\rangle$. Then the probability p_0 of measuring our qubit as $|0\rangle$ is $\sum_i |\alpha_i|^2$, yielding the state

$$\frac{1}{\sqrt{p_0}} \sum_i \alpha_i |i\rangle$$

and similarly for p_1 and $|1\rangle$. The scaling factor $\frac{1}{\sqrt{p_i}}$ renormalizes the quantum state so that the squares of the amplitudes still add up to 1.

For example, suppose we want to measure the first qubit in the state

$$\frac{1}{3}|00\rangle + \frac{2+i}{3}|01\rangle + \frac{1}{\sqrt{3}}|11\rangle.$$

We can break this up into $\frac{1}{3}|00\rangle + \frac{2+i}{3}|01\rangle$ and $\frac{1}{\sqrt{3}}|11\rangle$. The probability of measuring the qubit as $|0\rangle$ is

$$\left|\frac{1}{3}\right|^2 + \left|\frac{2+i}{3}\right|^2 = \frac{1}{9} + \frac{4+1}{9} = \frac{6}{9}$$

and the probability of measuring $|1\rangle$ is $\left|\frac{1}{\sqrt{3}}\right|^2 = \frac{1}{3}$. Hence when we measure a $|0\rangle$ we obtain the state

$$\sqrt{\frac{3}{2}} \left(\frac{1}{3}|00\rangle + \frac{2+i}{3}|01\rangle \right)$$

and when we measure a $|1\rangle$ we obtain

$$\sqrt{\frac{3}{1}} \frac{1}{\sqrt{3}} |11\rangle = |11\rangle.$$

Exercise 2. Now try measuring the second qubit in both of these cases. Verify that the distribution of results is the same as if we had measured the whole system at once.

Exercise 3. Verify that after measuring a qubit, the norm of the quantum state is still one.

2.2 Example: Quantum Teleportation

We can now introduce a simple quantum protocol known as *quantum teleportation*. First, we will introduce one more single-qubit gate:

$$\begin{aligned} Z|0\rangle &= |0\rangle \\ Z|1\rangle &= -|1\rangle \end{aligned}$$

The setup for quantum teleportation is as follows: Alice and Bob share a Bell pair, a pair of qubits in the entangled state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, with Alice holding the first qubit (a) and Bob the second (b). We will annotate these qubits with a and b for readability. Though they may be separated by some distance, they are entangled, and hence we use a single quantum state to represent them. Alice wants to send the state of some third qubit q in the state $\alpha|0\rangle + \beta|1\rangle$ to Bob but she has no quantum channel, only classical channels that can transmit non-quantum bits.

Hence, we begin with the state

$$(\alpha|0\rangle + \beta|1\rangle)_q \left(\frac{1}{\sqrt{2}}|0_a0_b\rangle + \frac{1}{\sqrt{2}}|1_a1_b\rangle \right)$$

which we can expand to

$$\frac{1}{\sqrt{2}} \left(\alpha|0_q0_a0_b\rangle + \alpha|0_q1_a1_b\rangle + \beta|1_q0_a0_b\rangle + \beta|1_q1_a1_b\rangle \right).$$

Alice first applies a *CNOT* from q (the controlling qubit) to a , obtaining the following state:

$$\frac{1}{\sqrt{2}} \left(\alpha|0_q0_a0_b\rangle + \alpha|0_q1_a1_b\rangle + \beta|1_q\mathbf{1}_a0_b\rangle + \beta|1_q\mathbf{0}_a1_b\rangle \right)$$

She then applies a Hadamard to q , obtaining

$$\begin{aligned} \frac{1}{\sqrt{2}} * \frac{1}{\sqrt{2}} & \left((\alpha|0_q0_a0_b\rangle + \alpha|1_q0_a0_b\rangle) + (\alpha|0_q1_a1_b\rangle + \alpha|1_q1_a1_b\rangle) + \right. \\ & \left. (\beta|0_q1_a0_b\rangle - \beta|1_q1_a0_b\rangle) + (\beta|0_q0_a1_b\rangle - \beta|1_q0_a1_b\rangle) \right) \end{aligned}$$

Alice's final step is to measure her qubits and then send the results of the mea-

measurements, which can each be encoded using a classical bit, to Bob. Let's rearrange some terms to simplify our calculations:

$$\frac{1}{2} \left(|0_q 0_a\rangle (\alpha |0\rangle + \beta |1\rangle)_b + |0_q 1_a\rangle (\alpha |1\rangle + \beta |0\rangle)_b + |1_q 0_a\rangle (\alpha |0\rangle - \beta |1\rangle)_b + |1_q 1_a\rangle (\alpha |1\rangle - \beta |0\rangle)_b \right)$$

We can now look at the four possible outcomes of measurement:

Case 1: Alice measured $|0_q 0_a\rangle$. This case occurs with probability $|\frac{1}{2}\alpha|^2 + |\frac{1}{2}\beta|^2$. Since $|\alpha|^2 + |\beta|^2 = 1$, this is equal to $\frac{1}{4}$, and we rescale by the square root of that probability, or $\frac{1}{2}$. Hence we arrive at the state:

$$|0_q 0_a\rangle (\alpha |0\rangle + \beta |1\rangle)_b$$

Bob's qubit is in precisely the state of Alice's original qubit, and the teleportation is complete.

Case 2: Alice measured $|0_q 1_a\rangle$. This also occurs with probability $\frac{1}{4}$. Bob obtains the state

$$|0_q 1_a\rangle (\alpha |1\rangle + \beta |0\rangle)_b$$

He applies an X to his qubit and obtains the desired state.

Case 3: Alice measured $|1_q 0_a\rangle$. The state of the system is

$$|1_q 0_a\rangle (\alpha |0\rangle - \beta |1\rangle)_b$$

The Z operation will flip the sign of the $|1\rangle$.

Case 4: Alice measured $|1_q 1_a\rangle$. The state is

$$|1_q 1_a\rangle (\alpha |1\rangle - \beta |0\rangle)_b$$

Bob first applies an X to obtain the state of case 3, and then applies a Z to obtain Alice's original quantum state.

2.3 Quantum Circuits

We will often represent sequences of quantum operations using *quantum circuits*. In this model, qubits travel along wires, and unitary operators are represented as gates

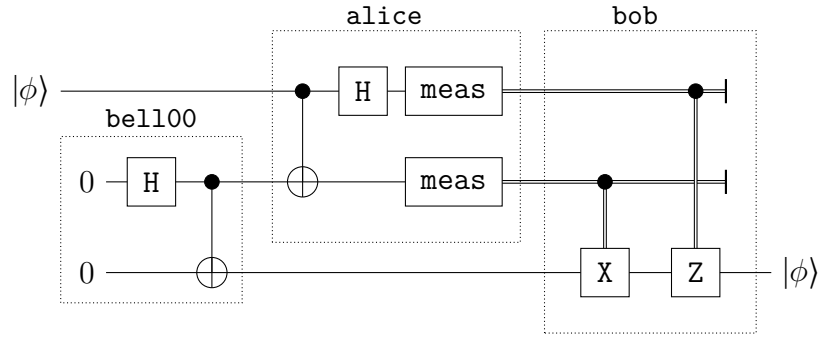


Figure 2.1: A quantum teleportation circuit

being applied to those wires.

Figure 2.1 shows the teleportation protocol from the previous section as a circuit. We explicitly initialize the bell pair $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ in the part of the circuit labeled “bell00”. The *CNOT* gate is given a special representation as a NOT (the \oplus on the bottom wire) “controlled” by the circle on the middle wire). Note that other gates can be controlled as well, as in the gates in the segment labeled “bob.” Alice applies a *CNOT* from the qubit to be transmitted to her member of the Bell pair, then applies a Hadamard and measures both qubits. We represent the output of this measurement using double lines, which represent classical bits. Bob uses these bits to control whether he applies *X* and *Z*, thereby obtaining the state of Alice’s original qubit.

2.4 Quantum States as Vectors

The exposition in the previous sections is somewhat lacking: It gives us a sense of what a quantum state is, but it does not describe what operations are quantum-mechanically valid, aside from the few operations given. To get a better sense of what operations are possible, we will switch to *vector notation*.

For a single qubit, the vector corresponding to $\alpha|0\rangle + \beta|1\rangle$ is

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

For the more general quantum state $\alpha_0|00\dots 0\rangle + \alpha_1|00\dots 1\rangle + \dots + \alpha_{n-1}|11\dots 1\rangle$ we have the following vector:

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{n-1} \end{pmatrix}$$

Note that the k^{th} element in the matrix is the coefficient of $|k_b\rangle$, where k_b is the binary representation of k . This makes it easy to switch between representations.

What is the meaning of $|\phi\rangle \otimes |\psi\rangle$ in vector notation? The tensor, or *Kronecker product*, takes an $m \times n$ matrix A and $o \times p$ matrix B and returns a $mo \times np$ matrix with copies of B tiled inside A . Visually, we have:

$$\begin{pmatrix} A_{1,1}B_{1,1} & \dots & A_{1,1}B_{1,p} & \dots & \dots & A_{1,n}B_{1,1} & \dots & A_{1,n}B_{1,p} \\ \vdots & \ddots & \vdots & \dots & \dots & \vdots & \ddots & \vdots \\ A_{1,1}B_{o,1} & \dots & A_{1,1}B_{o,p} & \dots & \dots & A_{1,n}B_{o,1} & \dots & A_{1,n}B_{o,p} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{m,1}B_{1,1} & \dots & A_{m,1}B_{1,p} & \dots & \dots & A_{m,n}B_{1,1} & \dots & A_{m,n}B_{1,p} \\ \vdots & \ddots & \vdots & \dots & \dots & \vdots & \ddots & \vdots \\ A_{m,1}B_{o,1} & \dots & A_{m,1}B_{o,p} & \dots & \dots & A_{m,n}B_{o,1} & \dots & A_{m,n}B_{o,p} \end{pmatrix}$$

Note that in the vector case, multiplying an m length vector by an n length vector returns a vector of length mn . However, the general case of the Kronecker product will prove useful, as we will see shortly.

Exercise 4. Write $(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$ as a vector, first by taking the Kronecker product directly and then by simplifying the expression and transforming it into vector notation. Confirm that both results are equal.

2.4.1 Unitary transformations

What kind of operations are valid on quantum states? Let us begin by noting that quantum states $|\phi\rangle$ correspond to *unit vectors*, or vectors of norm 1:

$$\| |\phi\rangle \| = \sqrt{|\alpha_1|^2 + \dots + |\alpha_n|^2} = 1.$$

A valid quantum operation should preserve this property and the size of the vector. We call such operations *unitary*.

Unitary transformations correspond precisely to *unitary matrices*, square matrices U satisfying the following:

$$UU^\dagger = I = U^\dagger U$$

Here I is the identity matrix, and the adjoint of U , U^\dagger , is a generalization of the transpose that takes the complex conjugate $\overline{A_{ij}}$ of all the elements. That is, it flips the sign of the imaginary components of all the elements. For an example, let's take

the transpose of Wolfgang Pauli's Y matrix:

$$Y^\dagger = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}^\dagger = \begin{pmatrix} \bar{0} & \bar{-i} \\ \bar{-i} & \bar{0} \end{pmatrix} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = Y$$

We see that Y happens to be its own adjoint.

Let us now multiply Y by its adjoint to confirm that it is unitary:

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} 0 + (-i)i & 0 + 0 \\ 0 + 0 & i(-i) + 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Theorem 1. For any unitary matrix U and unit vector ϕ , $\|U|\phi\rangle\| = 1$

Proof. The norm of $|\phi\rangle$ can be represented as the product of $|\phi\rangle^\dagger$ and $|\phi\rangle$, also written $\langle\phi|\phi\rangle$. Technically, this is a 1×1 matrix, but it is isomorphic to a scalar. We call the adjoint matrix $\langle\phi|$ a *bra* and $|\phi\rangle$ a *ket*. Hence the norm of $U|\phi\rangle$ can be written as

$$\begin{aligned} \|U|\phi\rangle\| &= (U|\phi\rangle)^\dagger (U|\phi\rangle) \\ &= \langle\phi|U^\dagger U|\phi\rangle \\ &= \langle\phi|\phi\rangle \\ &= 1 \end{aligned}$$

□

Here are the matrix forms of the operators H , X , and Z :

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The controlled version of any n -qubit gate U is represented by the block matrix

$$\left(\begin{array}{c|c} I_{2^n} & 0 \\ \hline 0 & U \end{array} \right)$$

The *CNOT* is simply the controlled X gate, and the controlled *CNOT* is called the Toffoli (*TOF* or *CCNOT*) gate.

Exercise 5. Note that H , X , Z , and *CNOT* are all their own adjoints. Verify that these matrices are unitary.

Exercise 6. Show that H , X , Z , and *CNOT* have the behavior described in the previous section. That is, show that when applied to basis vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ they produce the claimed output.

What does it mean mathematically to apply an operation to a single qubit in a multi-qubit system? Suppose we want to apply an H to the second qubit of a three

qubit system. We pad H on either side with 2×2 identity matrices and then apply $I_2 \otimes H \otimes I_2$ to the quantum state. An arithmetic identity called the *Kronecker mixed product* says that $(A \otimes B)(C \otimes D) = AC \otimes BD$ (provided the dimensions line up), so if the target system is separable into $|\psi\rangle_1 |\phi\rangle |\psi\rangle_2$, we get

$$\begin{aligned} (I_2 \otimes H \otimes I_2)(|\psi\rangle_1 |\phi\rangle |\psi\rangle_2) &= (I_2 |\psi\rangle_1) \otimes (H |\phi\rangle) \otimes (I_2 |\psi\rangle_2) \\ &= |\psi\rangle_1 (H |\phi\rangle) |\psi\rangle_2 \end{aligned}$$

as we would hope.

In general, a quantum computer can be expected to implement some number of unitary operations as primitive gates, though the precise set of primitives will differ based on architecture. It is important, however, that the implemented set be *universal* for quantum computation – that is, it can be used to efficiently approximate any unitary transformation. Here, the following two theorems are relevant:

Theorem 2 (Solovay-Kitaev). *If a given set of gates can approximate any 2×2 unitary matrix, it can approximate any such matrix efficiently (with a sequence of gates of length $\log^{3.97}(1/\epsilon)$ where ϵ is the allowed error).*

Theorem 3 (Shi (2003)). *Any unitary transformation U on n qubits can be approximated using Hadamard and Toffoli gates or CNOT and any single qubit gate g such that $g^2 \neq I$.*

The Solovay-Kitaev¹ theorem (Dawson and Nielsen, 2005; Nielsen and Chuang, 2010), which was generalized to multiple qubit matrices, tells us that universal gate sets can generally be interchanged with little loss of efficiency, provided that they consist of gates on small numbers of qubits. Hence, the specific choice of gate set is likely to depend on a given quantum computer’s hardware. Shi (2003) gives a number of strong candidates for universal sets to use in practice; Aharonov (2003) discusses the variety of gate sets known to be universal.

Many functions do not correspond to valid unitary transformation and therefore cannot be applied to quantum states. One important such function is the subject of the *no-cloning theorem*:

Theorem 4 (No Cloning). *There is no unitary transformation that copies the state of an arbitrary qubit (that is, takes $|\phi\rangle|0\rangle$ to $|\phi\rangle|\phi\rangle$).*

This theorem is due to Dieks (1982) and Wootters and Zurek (1982); a succinct proof is given on page 523 of Nielsen and Chuang (2010). The no-cloning theorem will motivate our use of linear types (which prevent us from copying terms) in the QWIRE quantum circuit language (Chapter 4).

¹This result was announced by both Solovay and Kitaev, but there is no corresponding publication; see Dawson.

2.5 Mixed States as Density Matrices

At this point, we want to expand our notion of *quantum states*. So far, a quantum state has simply been a 2^n -length complex vector, whose norm is equal to one. From here on we will call this a *pure state*. Note, however, that one of our operations—measurement—takes a pure state to a *distribution* over pure states. Since we would like to talk about these distributions directly, we will refer to them as *mixed states*.

Consider what happens if we apply a Hadamard H to the basis state $|0\rangle$ and then measure it. We will obtain a distribution over $|0\rangle$ and $|1\rangle$, with $\frac{1}{2}$ probability assigned to each outcome. We can (naively) write this as

$$\left\{ \left(\frac{1}{2}, |0\rangle \right), \left(\frac{1}{2}, |1\rangle \right) \right\}$$

where the real numbers on the left represent probabilities and the quantum states on the right represent measurement outcomes.

What if, without looking at the outcome of the measurement, we then apply another Hadamard to the state? We should obtain

$$\left\{ \left(\frac{1}{2}, H|0\rangle \right), \left(\frac{1}{2}, H|1\rangle \right) \right\}$$

which expands out to

$$\left\{ \left(\frac{1}{2}, \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \right), \left(\frac{1}{2}, \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \right) \right\}.$$

An interesting theorem of quantum mechanics says that the two distributions mentioned above are *physically indistinguishable*. For instance, if we measured either of the distributions above and examined the outcome, we would obtain $|0\rangle$ with probability $\frac{1}{2}$ and $|1\rangle$ with probability $\frac{1}{2}$. If we applied a Hadamard to either state, we would simply toggle between the two indistinguishable states, since $HH|0\rangle = |0\rangle$ and likewise for $|1\rangle$. Ideally, any language for talking about these states would identify them, but here the language of probability distributions over quantum states fails us.

Instead, we present an alternative way of representing quantum states that identifies these distributions with each other. A *density matrix* is a square matrix of dimensions $2^n \times 2^n$ that we can use to represent both pure and mixed quantum states. We can convert a pure state $|\phi\rangle$ in vector form to its density matrix representation by multiplying it by its adjoint, commonly written $\langle\phi|$. So, for instance, the basis states $|0\rangle$ and $|1\rangle$ become

$$|0\rangle\langle 0| = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} \bar{1} & \bar{0} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

and

$$|1\rangle\langle 1| = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} \bar{0} & \bar{1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Conveniently, we can check if a density matrix corresponds to a pure state by squaring it. For any pure state ρ in density matrix form, $\rho^2 = \rho$, and this is true only of pure states.

Exercise 7. Prove the first half of the above statement.

What if we want to apply a unitary matrix U to some quantum state ρ in its density matrix form? Let us first consider the case in which $\rho = |\phi\rangle\langle\phi|$ (that is, ρ is a pure state). Applying U to $|\phi\rangle\langle\phi|$ should give us $(U|\phi\rangle)(U|\phi\rangle)^\dagger$. We can distribute the adjoint over multiplication, obtaining $(U|\phi\rangle)(|\phi\rangle^\dagger U^\dagger)$, which we write as $U|\phi\rangle\langle\phi|U^\dagger$.

This gives us the formula for applying a unitary to a density matrix: We multiply the matrix by U on its left and U^\dagger on its right. Since density matrices and unitary operators are always square, if the dimensions are correct on the left, they will match up on the right.

What about measurement? This representation has the advantage of treating measurement as a deterministic operation on density matrices, rather than a probabilistic operation on vectors. Measuring a single qubit can be represented as follows:

$$\text{meas } \rho = |0\rangle\langle 0|\rho|0\rangle\langle 0| + |1\rangle\langle 1|\rho|1\rangle\langle 1|$$

You can think of this operation as adding together the two outcomes of measurement. The left hand side represents the event of measuring $|0\rangle$ and the right-hand side, measuring $|1\rangle$.

Let us return to the example that started this section. We start with the 0 qubit, represented in density matrix form as

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

We then apply a Hadamard operator, obtaining

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \right) = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}.$$

We then measure the state:

$$\begin{aligned} & \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \left(\frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right) \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \left(\frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right) \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \frac{1}{2} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{aligned}$$

As we see from the second line, there is a one-half probability of measuring $|0\rangle$ and one-half-probability of measuring $|1\rangle$. This helps us interpret the result. Each element

along the diagonal represents the probability of obtaining the corresponding basis state upon measurement (when we look at the result). Note that this was also true before we measured the state: Measurement simply removed the elements that weren't along the main diagonal, which indicated that the qubit was in a superposition, rather than a simple distribution.

As we noted earlier, applying another Hadamard to this state doesn't change it:

$$\begin{aligned} & \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \left(\frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right) \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \right) \\ &= \frac{1}{4} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ &= \frac{1}{4} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{aligned}$$

Exercise 8. Show that for a pure state in density matrix form, the i^{th} element along the diagonal is the probability of measuring $|i\rangle$.

Initializing and Discarding Qubits We've covered how to operate on qubits, both by applying unitary operators and by measuring them. But how do we represent adding new qubits to a system or discarding existing qubits?

Let's start with initialization. Adding a qubit to first position in a quantum state takes $|\phi\rangle$ to $|0\rangle \otimes |\phi\rangle$. This corresponds to multiplying by $|0\rangle \otimes I$ on the left, where I is the identity matrix with the same height as the target vector. In density matrix form, this corresponds to taking $(|0\rangle \otimes I)\rho(|0\rangle \otimes I)$. This takes a density matrix of dimensions $2^n \times 2^n$ to one with dimension $2^{n+1} \times 2^{n+1}$.

What about discarding a qubit? We cannot discard arbitrary qubits, because they might be entangled with the rest of the state. However, if we want to discard a $|0\rangle$, we can simply reverse the operation that initializes $|0\rangle$:

$$(\langle 0| \otimes I)(|0\rangle \otimes I)|\phi\rangle = (\langle 0|0\rangle \otimes I)|\phi\rangle = (I_1 \otimes I)|\phi\rangle = |\phi\rangle$$

We can do the same thing for $|1\rangle$. In density matrix form, we write $(\langle 0| \otimes I)\rho(|0\rangle \otimes I)$ or $(\langle 1| \otimes I)\rho(|1\rangle \otimes I)$. Density matrices also allow us to deal with the more general case, where we may not know whether the qubit to be discarded is $|0\rangle$ or $|1\rangle$. We then write the discard operation as $(|0\rangle \otimes I)\rho(|0\rangle \otimes I) + (|1\rangle \otimes I)\rho(|1\rangle \otimes I)$. Note that if the qubit to be discarded is $|0\rangle$, the right hand side will contain a $\langle 1|0\rangle = \mathbf{0}$, and everything to the right of the $+$ will be the zero matrix. The same is true for $|1\rangle$ and the left hand side. This form of discard is convenient, in that it also applies to a superposition—in this case, the qubit is implicitly being measured and then discarded. This accounts for the similarity to *meas* above: Measurement can be thought of as measuring and

discarding a qubit, then adding a new qubit in the measured state.

This should be sufficient to understand the semantics of quantum programs, which we will introduce in the upcoming sections.

2.6 Additional Material

In this chapter, we have tried to cover those aspects of quantum computing, and only those aspects, that are relevant to this thesis. This means that we have avoided a lot of terminology that a reader might expect to see in a paper on quantum computing. Most introductions to quantum computing will refer to *Hilbert spaces*, vector spaces that possess an inner product ($\langle\phi|\psi\rangle$ for quantum states, often written $\langle\phi|\psi\rangle$). Infinite-dimensional Hilbert spaces are of substantial interest to quantum physicists and have additional properties; however, quantum computation only deals with the specific case of finite-dimensional complex-valued vectors.

We also never explicitly discussed *interference*. Interference refers to two amplitudes canceling one another out. We saw this when we applied a Hadamard twice and obtained $\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle + \frac{1}{2}|0\rangle - \frac{1}{2}|1\rangle = |0\rangle$. From a physical standpoint, this is an interesting phenomenon (how are two non-zero probability events combining to form a zero-probability event?), but from a mathematical standpoint, it's captured in the mundane statement, “superposition behaves like addition”.

Qubits are often visualized as points on a sphere, called the *Bloch sphere*, where the poles correspond to $|0\rangle$ and $|1\rangle$. Using this model, we can visualize unitary operators as rotations around the sphere. In particular, Pauli's X , Y , and Z matrices are so named because they represent rotations around the x , y , and z axes. For our purposes, however, the vector and matrix representations of qubits have proven more useful, and geometric discussions would complicate the picture.

Readers interested in the details of quantum computing and information theory beyond those presented here are advised to consult Nielsen and Chuang's standard text on the subject (2010) or John Watrous's “Theory of Quantum Information” (2018).

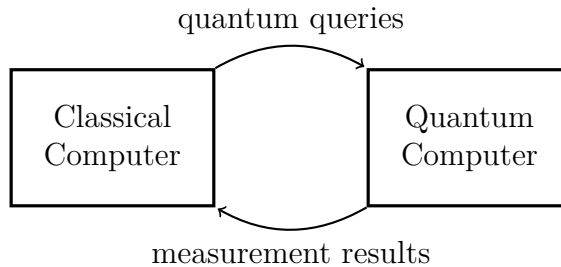
Chapter 3

A Brief History of Quantum Programming and Verification

3.1 Circuits, QRAM and Classical Control

The simplest and most influential model of quantum computation is the *quantum circuit model* (Deutsch, 1989), in which quantum operations are represented as quantum circuits like those in Section 2.3. Unfortunately, as Knill (1996) noted, circuits alone are insufficient to describe many quantum algorithms. In particular, many quantum algorithms assume a sort of control flow in which quantum operations are executed, the results are measured, and classical computations are run on the results. This cycle is often repeated as many times as necessary. In response, Knill proposed a set of guidelines for writing pseudocode for quantum algorithms, revolving around the *Quantum Random Access Machine*, or *QRAM*.

The QRAM model assumes that a quantum program has distinct sets of quantum and non-quantum (*classical*) registers. The quantum registers are very limited in their use: We can initialize quantum registers, apply unitary operations to them, and measure them, turning them into classical registers. The results of the measurement can then be used to perform classical computations. Implicitly, all of the quantum operations are performed on a quantum processor, while the remaining operations are performed locally, as in the following diagram:

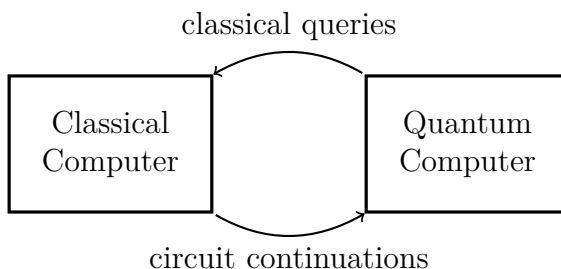


This model describes low-level languages like QASM (Balensiefer et al., 2005),

OpenQASM (Cross et al., 2017), and Quil (Smith et al., 2016) fairly precisely: Each has distinct registers used for quantum computation, which the controlling machine may measure and use in its computation. These “quantum assembly” languages are used in practice to run quantum computations on IBM’s 20-qubit quantum computer (the IBM Quantum Experience) and Rigetti Computing’s online Quantum Processing Units.

Knill’s pseudocode guidelines suggest that quantum registers might also be used as the guard in an IF-THEN-ELSE block, without detailing how to ensure such operations are quantum mechanically valid. In “Towards a Quantum Programming Language,” Selinger (2004a) proposes a narrower model for quantum computing known as *quantum data, classical control*. This widely adopted model states that the control flow of a quantum program should always be classical: Though we take advantage of the superposition of *data*, such as qubits, we should not attempt to run superpositions of *programs*. Instead, a quantum program should use classical conditionals, loops, and recursion. Most of the languages discussed below follow this framework, with the exception of QML (Altenkirch and Grattage, 2005). We will discuss the alternative approach, known as *quantum control*, in Section 3.9.

So far, we’ve portrayed the interaction between classical and quantum processors as unidirectional: The classical processor sends operations to the quantum computer, which returns measurement results. However, this doesn’t tell the whole story. Occasionally, a quantum computation will itself depend upon some complicated classical computation, which we wouldn’t want to run on a dedicated quantum processor. This classical computation, in turn, might depend on some measurement results. To deal with this, Green et al. (2013a) introduce the notion of *dynamic lifting*. This operation is initialized by the quantum processor, which sends data to the classical processor and asks it to compute the remainder of the quantum operation (in Quipper’s case, a circuit) for the QRAM to execute.



Many of the languages we will look at implement some form of dynamic lifting.

3.2 QCL: A General Purpose Quantum Language

In two masters theses (Ömer, 1998; Ömer, 2000) and a PhD thesis (Ömer, 2003), Bernhard Ömer paved a path that future quantum programming languages would

follow. His QCL, an imperative-style language, included support for advanced features like management of scratch qubits (or *ancillae*) and automatic circuit reversal.

QCL adheres pretty closely to Knill’s (1996) guidelines for quantum programming languages. It assumes that execution of a quantum program is primarily guided by a classical computer, which has an attached QRAM device for quantum operations.

In QCL, quantum operations are applied to sets of quantum registers. There are various restricted forms of registers that allow for constant-valued qubits or registers used exclusively for scratch space. These `qscratch` registers are garbage-collected by a procedure that returns them to the $|0\rangle$ state using Bennett’s (1973) technique. We discuss this technique in detail in Chapter 8.

QCL also allows for a sort of quantum conditional. We can annotate any function with the keyword `cond` to control all of its operations by a control qubit provided as an argument. This necessarily restricts the function being controlled—it must correspond to a unitary transformation on all its qubits; moreover, it cannot reference the control qubit itself. This is accomplished via strict scoping rules. This construction is then extended to allow for a limited quantum if statement and a bounded quantum while loop.

Even more so than Knill (1996), QCL established a baseline for what quantum programming languages should be able to do as well as a yardstick against which other languages could compare themselves (see, for instance, Rüdiger (2006); Green et al. (2013a)).

3.3 QPL: Semantics for Quantum Programming

Selinger (2004a) introduced QPL (also called QFC, for Quantum Flowchart Language), the first quantum programming language with a well-defined denotational semantics, including a semantics for recursion given in terms of least-fixpoints. QPL is a *flowchart language*: The primary representation of a QPL program is a directed graph with branching corresponding to conditional statements (cycles correspond to loops). QPL also has a more standard representation as a functional programming language with an imperative-style syntax. That is, while it is written in an imperative style, it has no notion of state and its only side-effects are measurement and non-termination.

The semantics of QPL is given in terms of 2^n -tuples of partial density matrices, where n is the number of classical bits in the system. (A *partial* density matrix is one in which the trace does not add up to one, and hence corresponds to a subdistribution on pure states.) Each member of this tuple has an index from $00\dots 0$ to $11\dots 1$, representing the value of all the bits in the system, so the 010^{th} matrix corresponds to a state in which only the second bit is 1. Each density matrix encodes the probability of this configuration in its trace. By contrast, the individual density matrices themselves encode the amplitudes of the program’s qubits. It’s worth noting that an equivalent semantics can be given using a single $2^{n+m} \times 2^{n+m}$ density matrix, where n and m are

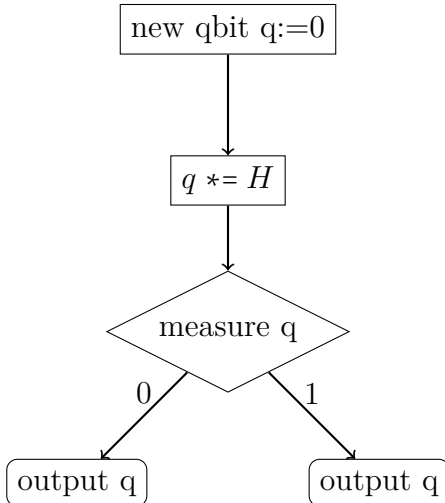


Figure 3.1: A QPL program for tossing a quantum coin

the number of bits and qubits, respectively, by treating bits as simply qubits that are always in one of the basis states. This representation is rejected in the paper in favor of economy of *representation*, since large sparse matrices take up a lot of space, but it is more *conceptually* parsimonious (easy to explain and reason about) and was adopted as the semantics for QPL by Kakutani (2009).

QPL syntax prohibits cloning qubits: While QCL and other languages fail at runtime if the user attempts to use the same qubit as both arguments to a *CNOT* gate, QPL doesn't allow aliasing and can therefore check that the arguments are distinct at compile time.

While representing circuits as flowcharts didn't catch on among quantum programming languages, QPL conveyed some important ideas that gained wide currency. Instead of including a quantum analogue of the IF statement, QPL treats measurement itself as a branching construct, as can be easily seen in Figure 3.1. QPL was also adopted as a target for verification efforts: Kakutani's (2009) quantum Hoare logic QHL reasons about QPL programs, and QPL programs are given a weakest pre-expectation semantics by D'Hondt and Panangaden (2006). D'Hondt and Panangaden's WP semantics is dual to the given semantics for QPL and was used in a subsequent line of work on quantum logics (Ying, 2011; Ying et al., 2017; Li and Ying, 2018).

Muerer's cQPL (2005) extends QPL with the ability to communicate with a classical computer, in line with the QRAM model, and a compiler to C code via QCL. Nagarajan et al. (2007) provide a compiler to an instruction set for a Sequential Quantum Random Access Memory (SQRAM) machine, based on Knill's QRAM. Unfortunately, neither contribution was sufficient to make QPL into a general purpose quantum programming language. Instead, QPL made a lasting contribution to the

areas of denotational semantics for quantum programs and compile time restrictions on cloning qubits. This work would form the basis for the quantum lambda calculi.

3.4 Linearity and the Quantum Lambda Calculi

Unlike in classical computation, where Alan Turing’s eponymous machines (1937) and Church’s lambda calculus (1936b) were invented simultaneously, David Deutsch’s quantum Turing machine (Deutsch, 1985) is widely considered the “founding paper” of quantum computing (though Feynman (1982), Benioff (1980), and Albert (1983) presaged it). The more popular quantum circuit model was also introduced by Deutsch (1989); Yao (1993) would demonstrate the two models’ equivalence. Quantum versions of the lambda calculus would follow, beginning with van Tonder’s (2004) λ_q .

λ_q attempts to address four issues with mixing classical and quantum computation:

1. reversibility of computation,
2. linearity of quantum states,
3. equational reasoning, and
4. completeness, or equivalence to the quantum Turing machine model.

All quantum computations must necessarily correspond to reversible functions. Unfortunately, many terms in the lambda calculus reduce to the same value, making it impossible to recover the original input (thereby making the computation irreversible and quantum mechanically impossible). Van Tonder addresses this by modifying the reduction rules in his calculus so that every reduction rule leaves a history, which allows us to recover the original lambda term.

The calculus also deals with linearity by introducing two kinds of lambda abstractions: $\lambda x.t$ takes a linear variable to some expression t , in which x must appear exactly once. By contrast, $\lambda!x.t$ may use x non-linearly and hence may only be applied to a non-linear term. Since λ_q has no type system, a non-linear lambda applied to a linear term is simply stuck.

Van Tonder defines an equational proof system for λ_q , allowing us to reason about the equivalence of λ_q expressions. He also sketches a proof of equivalence between the lambda calculus and Deutsch’s quantum Turing machine. In one direction, he shows that a quantum TM can simulate the reduction of a λ_q expression; in the other, he shows that λ_q can express arbitrary quantum circuits, which Yao (1993) showed to be equivalent to quantum Turing machines.

In subsequent work, van Tonder and Dorca (2003) developed a type system and denotational semantics for λ_q . Unfortunately, the categorical semantics, given in terms of “Hilbert bundles,” had a flaw, and the paper was withdrawn. This brings us to Selinger and Valiron’s lambda calculus, which successfully incorporated these features.

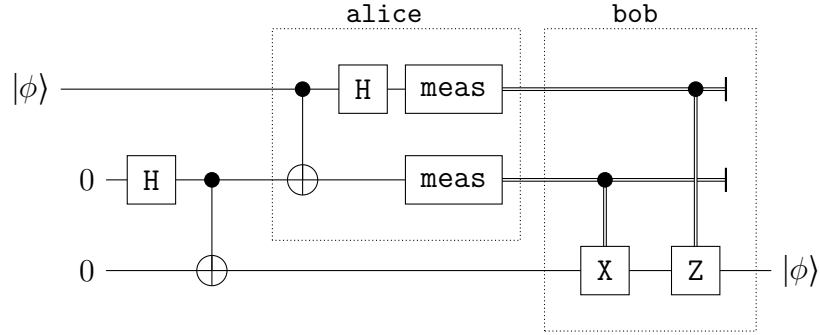


Figure 3.2: The teleportation function, composed of the entangled functions “alice” and “bob”

Selinger and Valiron’s (2006; 2008; 2009) Quantum Lambda Calculus is primarily concerned with *quantum functions* and linear type systems for typing these functions. To get a sense for quantum functions, let’s take another look at the teleport circuit (Figure 3.2). Normally we would think of the `alice` circuit as a function from two qubits to two bits, and we would think of `bob` as a function from a qubit and two bits to a qubit. But this isn’t quite accurate: Quantum teleportation only works when Alice and Bob share a pair of entangled qubits, or *Bell pair*, produced by the sub-circuit at the bottom left. These should be treated as a shared resource, not a separate pair of inputs. The `alice` function then takes a single qubit and returns a pair of bits, while `bob` takes a pair of bits and returns a qubit. We can then say that for any qubit q , $\text{bob}(\text{alice } q) = q$. Since these two functions share entangled qubits, we call them entangled.

Another unique feature of these two quantum functions is that they’re limited in their use. The `alice` circuit can clearly only be used once, since it measures (and thereby destroys) its half of the Bell Pair. By contrast, `bob` never measures its half, but it outputs that qubit and, hence, can no longer use it. This inspires the development of a type system to guarantee that quantum data, including quantum functions, is never duplicated.

To be precise, Selinger and Valiron’s lambda calculus incorporates an *affine* type system that ensures that quantum data is used at most once. This prohibits the programmer from copying data, whether in the form of qubits or quantum functions. The type system also provides a `!` (bang) operator to indicate types that may be used multiple times, along with a subtyping system that allow terms of type $!A$ to be used wherever an A is called for. The quantum lambda calculus has an operational semantics and a categorical semantics, which are valuable but not of direct relevance to this thesis, and proofs of type soundness (progress and preservation).

These lambda calculi, and their type systems in particular, had significant impact on subsequent programming languages, especially Quipper and our language, \mathcal{Q} WIRE. Before we talk about Quipper and \mathcal{Q} WIRE, though, we should address another lan-

guage that strongly influenced both.

3.5 The Quantum IO Monad

Altenkirch and Green’s (2010) Quantum IO Monad (QIO) took some early steps towards embedding quantum computation inside a functional host language. QIO was influenced by the quantum meta-language QML (Altenkirch and Grattage, 2005), also embedded inside Haskell, but had less ambitious goals: Instead of constructing a new language that mixes quantum and classical computing (including a quantum if statement), QIO provides a monadic interface for the Haskell programmer to run quantum programs. Treating quantum computing as a monad has several advantages:

1. It neatly separates quantum and classical computation, preventing the programmer from attempting to misuse quantum data in a classical program;
2. it gives QIO access to the full power of Haskell, including its typeclasses and libraries; and
3. it doesn’t restrict QIO to Haskell alone.

This last point was clearly illustrated by Green’s thesis (2010), which embedded the Quantum IO Monad inside Agda. Agda (Bove et al., 2009), like Coq, is a dependently-typed programming language that can also be used as a proof assistant. The Agda implementation of QIO uses dependent types to prevent some instances of qubit copying; for instance, applying a *CNOT* to the wires x and y requires a proof that x and y are distinct. However, Green embedded QIO in Agda mainly with an eye towards formal verification.

QIO includes a `USem` structure for representing the semantics of unitary operators. This is easily defined upon the classical fragment of unitary gates—gates like X , *CNOT*, and *CCNOT* that can be expressed as functions on bits. For an arbitrary unitary gate U , it defines the semantics of U in terms of its effect on each of the basis states. Unfortunately, due to Agda’s lack of automation or a real or complex number library (the author wrote a small axiomatic library of his own), it proved difficult to prove that unitaries are, in fact, *unitary*, or to prove any properties of quantum circuits. Nevertheless, QIO presaged our own efforts in this area and influenced Quipper, from which we took substantial inspiration.

3.6 Quipper and the Proto-Quippers

The Quipper programming language (Green et al., 2013a,b) was developed in the context of IARPA’s QCS project (IARPA, 2010), which challenged programming language developers to “accurately estimate and reduce the computational resources

required to implement quantum algorithms on a realistic quantum computer.” Javadi-Abhari *et al.*’s Scaffold (2012; 2015), a powerful imperative quantum programming language, was also developed as part of the QCS initiative. Quipper, like QML and QIO, is embedded within Haskell, but its primary focus is on scalable quantum computing. Towards that end, Quipper has a number of advanced features, including the ability to compile classical programs to quantum circuits, optimize circuits, and simulate quantum computation within Haskell. Quipper was initially used to implement seven complex quantum algorithms and study their resource requirements (Green *et al.*, 2013a; Smith *et al.*, April 2014). Siddiqui *et al.* (2014) also used Quipper to program a number of popular quantum algorithms, including Grover’s and Shor’s.

More than most languages, Quipper is transparent about its weaknesses. Quipper describes *circuit families*, broad classes of circuits that can be instantiated with a variety of different inputs. Unfortunately, this means that we only know the shape of these circuits (their inputs and outputs) at runtime. Quipper lacks both linear types and dependent types, the first of which can guarantee that qubits are not cloned, while the second can describe precise circuit families by giving their dependently typed signatures at compile time. On account of being embedded inside Haskell, Quipper lacks a denotational semantics, which makes it difficult to reason about Quipper programs. Quipper also makes heavy use of *assertive terminations*, where the programmer asserts that a qubit is in a specific state, but has no way of guaranteeing that these assertions are true. These assertions are used throughout the Quipper development, particularly in its compiler from classical to quantum programs.

These flaws directly influenced the development of our language, *QWIRE*. In fact, the Quipper papers can be read as a series of challenges to quantum programming language designers to which this thesis is a response. Chapter 4 takes up the challenge of including both linear and dependent types in a quantum programming language and giving that language a formal semantics. Chapter 5 addresses the issue of embedding a Quipper-like language inside a proof assistant, and Chapters 6 and 7 explore the payoff of doing so. Chapter 8 tackles the question of guaranteeing ancillae are used correctly and writing a verified compiler from classical programs to quantum circuits.

Having posed these challenges, the designers of Quipper also sought to answer them, in the context of a series of self-contained (non-embedded) languages called *Proto-Quippers*¹. The original Proto-Quipper (Ross, 2015) came equipped with a linear type system, inspired by Selinger and Valiron’s quantum lambda calculus (Selinger and Valiron, 2009). This type system treats all types as linear by default and uses the bang operator (!) to denote a persistent type. A subtyping relation allows us to use persistent variables in place of linear variables, though never the reverse. Proto-Quipper also allows us to package (or *box*) functions on qubits into circuits that are treated as data, and then to *unbox* these circuits later. As we will see, *QWIRE* uses

¹We feel that these languages should be called *Meta-Quippers*, for “that which comes after Quipper.” The Proto-Quipper authors seemingly intended that these language should eventually evolve into an idealized Quipper, with all the functionality of Quipper and all the underlying challenges addressed.

boxing and unboxing in a similar fashion.

Another version of Proto-Quipper, called Proto-Quipper M (Rios and Selinger, 2017), endows Proto-Quipper with a categorical semantics. Lindenhovius et al. (2018) extend Proto-Quipper M with general recursion and give it an abstract categorical model. This model takes inspiration from Rennela and Staton’s (2017) categorical model for EWire, itself an extension of QWIRE, completing the loop.

3.7 Liquid, Revs and Q#

The final languages we would like to dwell upon here are Microsoft’s Liquid (generally typeset as LIQ*U*i|)) and Q#, as well as the related REVS programming language. Liquid (Wecker and Svore, 2014) runs with Quipper’s idea of embedding a quantum circuit generating core inside a general purpose functional language, in this case F#. Liquid describes not only a language but a system for programming a quantum computer: Its major focuses are compilation, simulation and resource analysis. Liquid includes a variety of powerful circuit optimizations which have been proven, outside of Liquid, to maintain the semantics of the optimized circuit. It also provides multiple ways of simulating circuits to allow for more efficient simulation where possible (for instance, on Clifford group circuits).

An important outgrowth of the Liquid project was the REVS platform (Parent et al., 2017) for reversible computing. REVS is designed to compile classical programs to reversible circuits made out of Toffoli (*CCNOT*), *CNOT*, and *X* gates. REVS is not explicitly quantum, but it can export circuits to Liquid. Like Quipper’s compilation, REVS uses ancilla qubits and assertive terminations for efficiency. It also uses a strategy called *pebble games* to substantially optimize the compiled circuits. From the perspective of this thesis, the most interesting part of this work is the more lightly optimizing compiler REVERC (Amy et al., 2017). REVERC shares a source and target language with the REVS compiler, but it is formally verified in the F* programming language (Swamy et al., 2011). This guarantees that ancillae are terminated correctly and that compiled circuits correspond to their input functions. This strongly influenced QWIRE’s own approach to ancillae and compilation, discussed in Chapter 8.

As part of a recent turn away from embedded languages, Microsoft Research released a successor to Liquid called Q#. Q# is probably the most powerful stand-alone language for quantum computing. Officially, Q# has no model of a quantum circuit, instead treating qubits as first-class objects. However, it isn’t difficult to read a Q# program as generating a circuit, and Q# programs can indeed be compiled to quantum circuits — much of the documentation even uses this abstraction. However, Q# often eschews the circuit model in favor of describing quantum-classical algorithms like the repeat-until-success loop. Q# makes some distinctions between simulated quantum programs and those run on a quantum computer. For instance, it allows the programmer to make assertions about the state of a qubit, like those in Quipper,

which will be checked by the simulator and ignored by a real quantum computer, since checking these assertions requires inspecting the quantum state without measuring it. This capability augments $Q\#$'s facility for managing ancillae, including borrowing arbitrary qubits to return them to their initial state, but still doesn't guarantee that ancillae are used correctly, as REVERC does in the classical setting. As a highly expressive standalone language with support for polymorphism and a variety of quantum-specific abstractions, $Q\#$ stands on the frontier of quantum programming languages, albeit in a way that is largely orthogonal to Proto-Quipper and $QWIRE$.

3.8 The World of Quantum Programming

In this chapter, we have avoided giving a full account of the history of quantum programming languages in favor of focusing on the languages that most influenced our own work. Some of the early languages that we have neglected to discuss include Stephen Blaha's (2002) Quantum C Language and a language by Bettelli et al. (2003) known colloquially as the "Q" language. We also neglected some recent work on quantum lambda calculi, including the work of Altenkirch et al. (2007) and Vizzotto et al. (2009a,b, 2013).

More recently, a variety of Python libraries have been developed for quantum computing, most prominently Project Q (Steiger et al., 2018) and Rigetti Computing's PyQUIL (Rigetti Computing). Scaffold and the Scaffold compiler (Javadi-Abhari et al., 2012, 2015; Heckey et al., 2015) also provide a powerful quantum computing platform. Other interesting recent languages include the Blackbird language for photonic quantum computing (Killoran et al., 2018), QUMIN (Singh et al., 2017), IQu (Paolini et al., 2017), and qPCF (Paolini and Zorzi, 2017).

We also haven't discussed more limited circuit description languages designed for execution on a quantum computer, though these are critical in practice. One of the earliest such languages was QASM (Balensiefer et al., 2005), designed to be a universal target for quantum programming languages. QASM was recently superseded by OpenQASM (Cross et al., 2017), which is used in a number of popular simulators and IBM's online quantum computer (IBM, 2017). Rigetti's QUIL (Smith et al., 2016) is part of their broader FOREST platform and directly targets Rigetti's own quantum cloud. Ongoing work by Dong-Ho Lee (to be discussed in Section 11.2) uses OpenQASM as a compilation target for $QWIRE$.

3.9 Models of Quantum Computation

In this chapter, we have discussed a fairly narrow, though popular, approach to quantum computing: a quantum circuit model with classical control. However, a number of alternative models exist. The notion of quantum control, in which multiple programs may be run in superposition with one another, is hotly debated. For a negative

view on the prospects for quantum control, see Badescu and Panangaden (2015). This view, and the argument that quantum control is in the general case impossible, has failed to dissuade certain researchers in the field. Ying et al. (2012, 2014) proposes notions of quantum alternation and quantum recursion. QML (Altenkirch and Grattage, 2005) includes a basic form of quantum branching subject to strong constraints; indeed, notions of quantum branching go back to the original QRAM paper (Knill, 1996). More recently, Sabry et al. (2018) proposed a language with quantum loops and recursion, but without measurement, in which all programs are guaranteed to terminate and shown to correspond to valid quantum computations.

A few other approaches don't use quantum circuits at all. A *one-way quantum computer* (Raussendorf and Briegel, 2001, 2002) can do general-purpose quantum computation using only an initial fully-entangled state and single qubit measurements. The *measurement calculus* (Danos et al., 2007, 2009) can be viewed as a programming language for this kind of one-way quantum computer. Inspired by the category theory of quantum computation, *categorical quantum mechanics* (Abramsky and Coecke, 2004; Coecke and Kissinger, 2017) models a quantum process as an undirected graph, leading to languages like the ZX-Calculus (Coecke and Duncan, 2008; Backens, 2014) and ZW-Calculus (Hadzihasanovic, 2015, 2017). One crucial contribution of these languages is their equational theories and extensive rewriting systems, which are used to optimize ZX-diagrams in the Quantomatic tool (Kissinger, 2011; Fagan and Duncan, 2018). Further afield lies D-Wave's approach to quantum computing, which is based on adiabatic quantum computing (Albash and Lidar, 2018) and is targeted at specific problems like simulated annealing (an approach to finding maxima for functions) (Johnson et al., 2011). The D-Wave machines are programmed using *quantum machine instructions* in a variety of programming languages (D-Wave Systems, Inc, 2013).

Unfortunately, most surveys of quantum programming languages are at least seven years old, a lifetime in this field. For more information on early quantum programming languages, we refer the reader to surveys by Selinger (2004b), Gay (2006), Rüdiger (2006), and Mischczak (2011). For more up-to-date information, the discussion section of Svore et al. (2018) may prove enlightening.

3.10 Formal Verification

The area of formally verified quantum computing is less developed than that of quantum programming languages. Nevertheless, it is worth looking at two approaches that are relevant to this dissertation: logical systems for reasoning about quantum programs and mechanized proofs of results from quantum information theory.

3.10.1 Quantum Logics

A variety of approaches have been taken towards verifying quantum programs, the most common being program logics. These include quantum versions of guarded command language (Sanders and Zuliani, 2000), dynamic logics (Brunet and Jorrand, 2004; Baltag and Smets, 2006, 2011), and Hoare logics (Chadha et al., 2006; Kakutani, 2009; Ying, 2011; Ying et al., 2017). Two of these logics are of particular interest to us.

Ying’s (2011) quantum Hoare logic reasons about *quantum weakest preconditions*. A quantum weakest precondition (D’Hondt and Panangaden, 2006) adapts the notion of a weakest precondition to a quantum setting in much the same way that Kozen’s (1985) weakest pre-expectations generalize preconditions to a probabilistic setting. In both settings, the claims are arithmetic: The judgment $\{P\} c \{Q\}$ says that for any initial state σ , $P(\sigma) \leq Q(\llbracket c \rrbracket \sigma)$ (plus a probability of non-termination, in the partial case) for some suitable order \leq . In the probabilistic case, this is just the standard order on real numbers, and P and Q are measurable functions. In the quantum case, P and Q come from a class of matrices called *observables*, and \leq is the *Löwner partial order* on matrix traces. Ying (2011) embeds this in a logic, later extended with proof techniques to guarantee program termination (Ying et al., 2017). This logic was embedded within the Isabelle/HOL proof assistant (Nipkow et al., 2002), allowing for mechanized proofs about quantum programs (Liu et al., 2016).

Another interesting logic was also mechanized using Isabelle/HOL. Unruh’s (2018) Quantum Relational Hoare Logic (QRHL) is based upon Barthe *et al.*’s (2012) Probabilistic Relational Hoare Logic, used in the EasyCrypt cryptographic tool. QRHL makes assertions about the relationships of two quantum programs to one another, with the goal of proving indistinguishability. These assertions allow us to prove the security of quantum security protocols and show that classical protocols are secure in the quantum setting. QRHL forms the core of an EasyCrypt-like tool for proving quantum security, available at <https://github.com/dominique-unruh/qrhl-tool>.

3.10.2 Mechanized Verification

We found two other compelling approaches to quantum verification. The first, Boender *et al.*’s (2015) “Formalization of Quantum Protocols using Coq,” attempts to directly prove properties of quantum programs inside the Coq proof assistant. Here, qubits are defined as simple length-2 vectors (then generalized to length- 2^n vectors), and quantum programs are defined directly as transformers on these qubits. This is used to prove the correctness of some quantum programs, though ones without any levels of abstraction.

More recent work by Amy (2018) verifies the correctness quantum circuits by means of *Feynman path integrals* (Feynman and Hibbs, 1965). A path integral calculates the amplitude of a quantum state as the sum over possible paths to that state. In the general case, this would require using integrals, but quantum comput-

ing discretizes qubit states, allowing us to take simple sums. Amy uses these path sums to give a semantics to quantum circuits and proposes an algorithm for simplifying these sums, allowing us to check equivalence. This approach is shown to be complete for Clifford-group circuits, in that simplification will always terminate and produce a unique normal form. In the general case, this procedure is not guaranteed to terminate, but in practice, it has proven the correctness of a number of important programs, including the quantum Fourier transform. It has some drawbacks, though, in that these proofs are only for circuits of fixed size, and the technique is only used to verify simple circuits rather than complex quantum programs. Interestingly, this verification is all done in Haskell, which means that it doesn't produce proof terms that can be used in larger proof developments.

This concludes our introduction to quantum programming and verification. In subsequent chapters, we will describe our attempt to apply formal verification in the context of a full-fledged quantum programming language, *QWIRE*.

Chapter 4

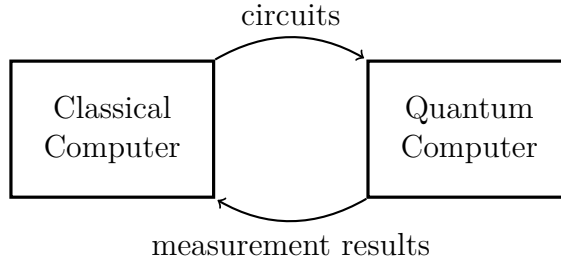
Qwire in Theory

4.1 Introduction

The standard architecture for quantum computers follows the *quantum circuit model*, which presents quantum computations as sequences of gates over qubits. As with classical circuits, quantum circuits exist at a very low level of abstraction, and yet in spite of this, researchers and industry professionals write complex quantum algorithms in state-of-the-art quantum circuit languages like Quipper (Green et al., 2013a), Scaffold (Javadi-Abhari et al., 2012) and Liquid (Wecker and Svore, 2014).

Why is the quantum circuit model so successful? It’s partly because circuits, unlike quantum computation more broadly, are simple and easy to understand. Research into operations that directly interface with quantum data, like qubit-controlled conditionals and recursion, is still in its infancy (Ying, 2014; Badescu and Panangaden, 2015), so programmers cannot be sure that their algorithms using such abstractions are quantum-mechanically valid.

Although circuits manipulate quantum data, they themselves are classical data—a circuit is just a sequence of instructions describing how to apply gates to wires. In practice, this means that circuits can be used to build up layers of abstractions, hiding low-level details. The QRAM model of quantum computing (Knill, 1996) formalizes this intuition by describing how a quantum computer could work in tandem with a classical computer. In the QRAM model, the classical computer handles the majority of ordinary tasks, while the quantum computer performs specialized quantum operations. To communicate, the classical computer sends instructions to the quantum machine in the form of quantum circuits. Over the course of execution, the quantum computer sends measurement results back to the classical computer as needed.



Embedded languages like Quipper, Liquid, the Q language (Bettelli et al., 2003), and the quantum IO monad (Altenkirch and Green, 2010) can be thought of as instantiations of this model. They execute by running host language programs on the classical computer, making specialized calls to the quantum machine. The classical host languages allow programmers to easily build up high-level abstractions out of low-level quantum operations.

However, such abstractions are only worthwhile if the circuits they produce are safe—if they do not cause errors when executed on a quantum computer. Unfortunately, proving that an embedded language produces well-formed circuits is hard because it means reasoning about the entirety of the classical host language. This is frustrating when we care most about the correctness of quantum programs, which we expect to be more expensive and error-prone than the embedded language’s classical programs.

One way of ensuring the safety of circuits is via a strong type system. Type safety for a quantum programming language means that well-formed circuits will not get stuck or “go wrong” when executed on a quantum machine. A subtlety is that this definition implies that the quantum program is implementable in the first place on a quantum computer—that the high-level operational view of the language is compatible with quantum physics. One way of ensuring that the language is implementable is to give a denotational semantics for programs in terms of quantum mechanics.

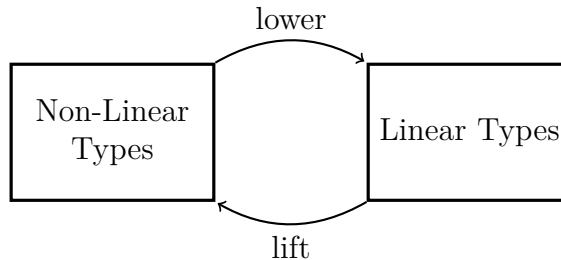
Several quantum programming languages have been proposed with an emphasis on type safety, including Selinger’s QPL language (Selinger, 2004a), the quantum lambda calculus (Selinger and Valiron, 2009), QML (Altenkirch and Grattage, 2005), and Proto-Quipper (Ross, 2015). However, these are toy languages, not designed for real-world, scalable quantum programming.

In this chapter we address the tension between expressive embedded languages and denotationally sound type-safe languages.

4.1.1 The Best of Both Worlds: Qwire

We present a core quantum circuit language in which circuits, equipped with a purely linear type system to ensure type safety, are explicitly separated from an arbitrary classical host language. The circuit language, which we call `QWIRE` (“choir”), comes equipped with an interface to this host language, providing all the benefits of an embedded language while maintaining type safety and soundness.

The quantum lambda-calculus (Selinger and Valiron, 2009) popularized the use of linear types for quantum systems. The “no-cloning” theorem of quantum mechanics states that quantum data cannot be cloned; in a programming environment, linear types ensure that quantum programs do not try to violate this property. However, the programming model should also allow for non-linear programming of ordinary classical data. The quantum lambda calculus addresses this via subtyping, but for \mathcal{Q} WIRE we take an alternative approach inspired by the symmetry between the QRAM model and Benton’s Linear/Non-Linear (LNL) logic (1995), as depicted in the following diagram:



In \mathcal{Q} WIRE, quantum circuits execute on the quantum computer and are given linear types, while host language programs execute on the classical computer and are given ordinary non-linear types.

Structuring the system in this way has several advantages. First, the interface to circuits is minimal, which means that they can easily be reasoned about. Second, the host language is extensible, since changes to the host language don’t induce changes to the circuit language, and vice versa. Third, the relationship between the circuit language and host language can be axiomatized: every circuit can be promoted to the host language via a *box* operator and later *unboxed* for reuse in other circuits. This allows circuits to be treated as classical data structures in the host language, while prohibiting quantum data, such as qubits, from escaping the linear type system.

The axiomatic approach means that the circuit language is relatively independent from the host language. In particular, the host language can be instantiated with a wide range of programming languages depending on the intended use: high-level functional programming languages for developing and reasoning about algorithms; theorem provers for verification of quantum circuits; and perhaps even hardware description languages for deployment with real quantum computers. In Chapter 5 and subsequent chapters, we will focus on one specific host: The Coq proof assistant (Coq Development Team, 2018) and its programming language, Gallina.

4.1.2 Chapter Outline

- We present \mathcal{Q} WIRE, a core quantum circuit language, along with a simple linear type system (Section 4.3) and an equational operational semantics (Section 4.4). In addition to the circuit language itself, we describe a minimal interface to a

classical host language that allows for modularity and communication via the QRAM model.

- We prove that the operational semantics of \mathcal{Q} WIRE is type safe (Theorems 5 and 6), and that all circuits reduce to a small set of normal forms (Theorem 7), depending only on the correctness of the host language.
- We give a denotational semantics in terms of density matrices (Section 4.5) and prove that the operational semantics is sound with respect to it (Theorem 8).
- Throughout, we give examples of circuits written in an archetypal host language with access to \mathcal{Q} WIRE (Section 4.2). We also consider how to extend the host language with case analysis of circuits and dependent types (Section 4.3.2) to express programs that cannot be written in existing circuit languages.

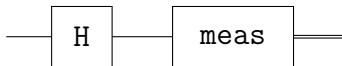
4.2 Qwire by Example

We start by taking a look at some code written in a host language that has access to \mathcal{Q} WIRE circuits¹. Circuits are constructed by a *box* operator that binds the input, represented as a *wire name*, inside of a circuit. Each wire name is identified with a *wire type*, which is either a bit, a qubit, or a (possibly empty) product of wire types.

For example, the identity circuit is written `id := box w => output w` and has the type `Box W W` for any wire type `W`. The wire name `w` in this example is not a regular variable like one would use in a classical programming language. For one, a wire is not first class: it is not by itself a circuit. For another, wire variables can only be used inside a circuit and must be used *linearly*—once it is used, a wire cannot be used again.

Gate application is the most important operation on wires. For example, the following circuit applies a Hadamard gate (\mathbb{H}) to its input wire, followed by a measurement gate. Each gate has an associated input and output type and can only be applied to wires of the appropriate type.

```
hadamard-measure : Box Qubit Bit :=
  box q =>
    q' ← gate H q;
    b ← gate meas q';
    output b
```



Note that we sometimes write `(gate g w)` as shorthand for the `(w' ← gate g w; output w')` that appears in the example.

¹We will use Coq-style syntax for consistency with the rest of the thesis, though \mathcal{Q} WIRE could be embedded in any number of host languages, depending on the features desired.

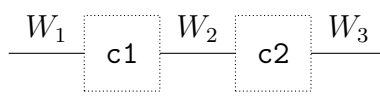
The reason wires must be treated linearly is that applying a gate changes the nature of the wire w . It is meaningless to apply two gates to the same wire, because wires (and in particular qubits) cannot be duplicated. The following code, for example, is absurd:

```
absurd :=
  box w =>
    x ← gate meas w;
    w' ← gate H w;
    output (x,w')
```

Similarly, it is dangerous to implicitly discard references to wires, which might be entangled in a greater quantum system. In \mathcal{Q} WIRE, the `discard` gate explicitly discards a bit-valued wire, whereas qubit-valued wires must be measured before being discarded.

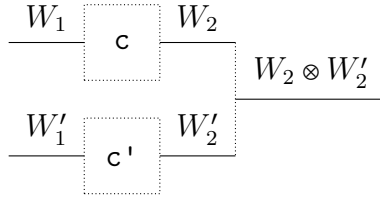
Since gates act on wires and not circuits, the expression `gate meas (gate H w)` is ill-formed. However, circuits can be composed by connecting the output of one circuit to the input of another. This type of composition is most useful when using circuits that have previously been constructed by a `box` operator. Boxed circuits can be *unboxed* by connecting some free input wires to the input of the box. The following function composes two boxed circuits in sequence, resulting in one complete circuit:

```
inSeq (c1 : Box W1 W2) (c2 : Box W2 W2)
: Box W1 W2 :=
  box w1 =>
    w2 ← unbox c1 w1;
    unbox c2 w2
```



The type system ensures that the output wire of the first circuit matches the input wire to the second. More complex composition is also possible. For instance, `inPar` composes any two circuits in parallel, with no restriction on their wire types.

```
inPar (c : Box W1 W2) (c' : Box W1' W2')
: Box (W1 ⊗ W1') (W2 ⊗ W2') =
  box (w1,w1') =>
    w2 ← unbox c w1;
    w2' ← unbox c' w1';
    output (w2,w2')
```



In the host language, we can write functions that compute circuits based on classical values, such as the following initialization function for qubits that determines which initialization gate gets applied.

```
init (b : Bool) : Box One Qubit :=
  if b then box () => gate init1 ()
  else box () => gate init0 ()
```


Definition bell00 :

```
Box One (Qubit⊗Qubit) :=
box () ⇒
  a ← gate init0 ();
  b ← gate init0 ();
  a ← gate H a;
  (a,b) ← gate CNOT (a,b)
  output (a,b)
```

Definition bob :

```
Box (Bit⊗Bit⊗Qubit) Qubit :=
box (x,y,b) ⇒
  (y,b) ← gate (bit-ctrl X) (y,b);
  (x,b) ← gate (bit-ctrl Z) (x,b);
  () ← gate discard y;
  () ← gate discard x;
  output b
```

Definition alice :

```
Box (Qubit⊗Qubit) (Bit⊗Bit) :=
box (q,a) ⇒
  (q,a) ← gate CNOT (q,a)
  q ← gate H q;
  x ← gate meas q;
  y ← gate meas a
  output (x,y)
```

Definition teleport :

```
Box Qubit Qubit :=
box q ⇒
  (a,b) ← unbox bell00 ();
  (x,y) ← unbox alice (q,a);
  q ← unbox bob (x,y,b);
  output q
```

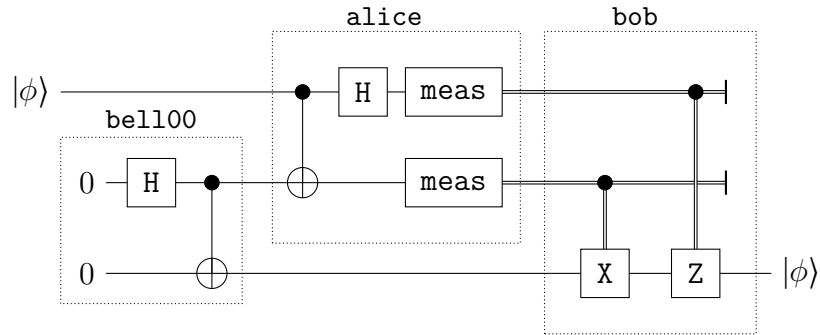


Figure 4.1: A \mathcal{Q} WIRE implementation of quantum teleportation without dynamic lifting.

Quantum Teleportation. The quantum teleportation algorithm of Chapter 2 highlights the relationship between boxed and unboxed circuits. Figure 4.1 shows the quantum teleportation circuit broken up into four parts. Alice is trying to send the input qubit q to Bob. The circuit `bell00` initializes two qubits in the zero state (written `init0`), places qubit a in a superposition of $|0\rangle$ and $|1\rangle$ via the Hadamard (`H`) gate, and entangles it with qubit b by applying a controlled-not (`CNOT`) gate. Qubit a is then given to Alice, and qubit b to Bob. Alice entangles a and q and measures them, outputting a pair of bits x and y . Bob then uses these bits to control an `X` and `Z` gate applied to his own qubit b , thereby placing b in the state of the original qubit q .

Communication via Lifting. In the teleportation example, the bit-valued wires x and y are treated as controls in the `bob` circuit. Intuitively, the bits x and y contain classical information, and so they should be able to be manipulated by the host language. The *dynamic lifting* operation promotes patterns of bits to the host language so that they can be manipulated using classical reasoning principles.² The `bob` circuit could be written instead using dynamic lifting:

```
bob-lift : Box (Bit ⊗ Bit ⊗ Qubit) Qubit :=
  box (w1,w2,q) ⇒
    (x1,x2) ← lift (w1,w2);
    q ← unbox (if x2 then boxed_gate X else id) q;
    q ← unbox (if x1 then boxed_gate Z else id) q;
    output q
```

where `boxed_gate g := box w ⇒ gate g w`.

On the one hand, dynamic lifting produces legible code that is easy to understand because it concentrates more computation in the host language. On the other hand, dynamic lifting is inefficient because the host language code must be run on a classical computer, during which time the quantum computer must remain suspended, waiting for the remainder of the circuit to be computed. Even in the example above, asking the classical computer to decide whether `X` or the identity ought to be applied would be slower than simply applying a controlled-not gate. Though dynamic lifting is not necessary in the case of quantum teleportation, it is an integral part of many quantum algorithms, including quantum error correction, and so must be accounted for coherently.

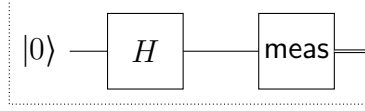
The examples shown so far describe all of the ways to construct circuits in `QWIRE`. However, we would also like to have a *run* operation, which takes a circuit with no input and produces a value. For example, the following code implements a quantum coin toss:

²Dynamic lifting can be applied to qubits as well as bits by implicitly measuring the qubits before producing a host-language value.

```

flip : Bool =
  run (q ← gate init0 ());
      q ← gate H q;
      b ← gate meas q;
      output b)

```



Note that the Coq implementation of \mathcal{Q} WIRE does not include a `run` operation, since Coq programs are not intended to be executed. Moreover, Coq is a pure language, and hence cannot handle input and output or probabilistic operations. However, given that \mathcal{Q} WIRE is host-language agnostic – and even \mathcal{Q} WIRE programs written in Coq should eventually be executed, either on a QRAM device or through extraction to OCaml or Haskell (see Section 11.2) – we will discuss `run` and its semantics in this chapter.

4.3 The \mathcal{Q} wire Circuit Language

We will now introduce the syntax and type theory of \mathcal{Q} WIRE and the interface for integrating \mathcal{Q} WIRE circuits into a host language.

4.3.1 Circuit Language

As shown above, a circuit can be thought of as a sequence of gates on wires. These wires can contain a unit (no data), a bit or qubit, or a pair of wires, as described by the following *wire types*:

$$W ::= \text{One} \mid \text{Bit} \mid \text{Qubit} \mid W_1 \otimes W_2$$

\mathcal{Q} WIRE is parameterized by a collection of gates \mathcal{G} , each equipped with input and output types. We write $\mathcal{G}(W_1, W_2)$ for the set of gates with input W_1 and output W_2 . The gate set could consist of any collection of gates, but in the setting of quantum circuits it is conventional to choose a universal subset $\mathcal{U} \subseteq \mathcal{G}$ of unitary gates such that, for every $u \in \mathcal{U}(W, W)$, we also have

$$\begin{aligned}
&u^\dagger \in \mathcal{U}(W, W) \\
&\text{ctrl } u \in \mathcal{U}(\text{Qubit} \otimes W, \text{Qubit} \otimes W) \\
&\text{bit-ctrl } u \in \mathcal{U}(\text{Bit} \otimes W, \text{Bit} \otimes W)
\end{aligned}$$

Additionally, we assume we have initialization gates for bits and qubits

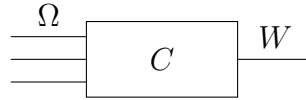
$$\text{new}_0, \text{new}_1 \in \mathcal{G}(\text{One}, \text{Bit}) \quad \text{init}_0, \text{init}_1 \in \mathcal{G}(\text{One}, \text{Qubit})$$

as well as a measurement gate $\text{meas} \in \mathcal{G}(\text{Qubit}, \text{Bit})$ on qubits and a discard gate $\text{discard} \in \mathcal{G}(\text{Bit}, \text{One})$ for bits.

A typing judgment $\Gamma; \Omega \vdash C : W$ specifies when a circuit is well-formed. In this judgment,

- C is a circuit;
- $\Omega = w_1 : W_1, \dots, w_n : W_n$ is a context of input wire names with their wire types;
- $\Gamma = x_1 : A_1, \dots, x_n : A_n$ is a context of host language variables with their host language types; and
- W is the output type of the circuit.

Thus, all well-typed circuits have the following shape:



Wires in \mathcal{Q} WIRE are *linear*, which means that they cannot be duplicated or discarded (though some gates may duplicate or discard classical bits) and when we write Ω, Ω' we assume that Ω and Ω' contain only disjoint wire names. Both Ω and Γ are *unordered* contexts.

The output of a circuit is built up as a *pattern* of its input wires:

$$\frac{\Omega \Rightarrow p : W}{\vdash; \Omega \vdash \text{output } p : W} \quad \begin{array}{c} \Omega \\ \hline \hline \hline \hline \end{array} \quad W$$

A pattern is just a tuple of wires with some wire type:

$$\frac{}{() : \text{One}} \quad \frac{}{w : W \Rightarrow w : W} \quad \frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2}{\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \otimes W_2}$$

A gate can be applied to a pattern of wires when permitted by the signature of the gate. The output of that gate is then decomposed by another pattern. The wires exiting the gate can then be used in the remainder of the circuit.

$$\frac{\begin{array}{c} \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \\ g \in \mathcal{G}(W_1, W_2) \quad \Gamma; \Omega_2, \Omega \vdash C : W \end{array}}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W} \quad \begin{array}{c} \Omega_1 \\ \hline \hline \hline \hline \end{array} \begin{array}{c} \Omega_2 \\ \hline \hline \hline \hline \end{array} \quad \begin{array}{c} \hline \hline \hline \hline \end{array} \quad W$$

We compose circuits by connecting the output of one circuit to the input wires of another. This operation differs from sequential composition in that the second circuit may have additional inputs.

$$\frac{\Gamma; \Omega_1 \vdash C : W \quad \Gamma; \Omega_0, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \quad \begin{array}{c} \Omega_1 \\ \hline \hline \hline \hline \end{array} \quad \begin{array}{c} \hline \hline \hline \hline \end{array} \quad \begin{array}{c} \Omega \\ \hline \hline \hline \hline \end{array} \quad \begin{array}{c} \hline \hline \hline \hline \end{array} \quad W'$$

4.3.2 Host Language

In the QRAM model, a classical computer works together with a quantum computer. The classical computer communicates with the quantum computer by sending it instructions—that is, circuits in \mathcal{QWIRE} . Terms in the host language, meanwhile, describe computations on the classical computer. We refer to the host language as HOST and describe some of its properties.

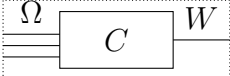
We assume that HOST is statically typed, and we write its types using the meta-variable A . We further assume that the language has unit, boolean and product types, to correspond with \mathcal{QWIRE} 's `One`, `Bit` and tensor wire types. We add to HOST a type representing \mathcal{QWIRE} circuits between two wire types, which we write $\text{Box } W_1 W_2$. Of course, HOST will often contain many other types, including functions and inductive data types, but the interface with \mathcal{QWIRE} does not depend on the particular structure of HOST . For this reason we say that HOST is arbitrary.

Overall, we can summarize the types of HOST as follows:

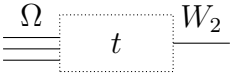
$$A ::= \dots \mid \text{Unit} \mid \text{Bool} \mid A \times A \mid \text{Box } W_1 W_2$$

The typing judgment for HOST terms is written $\Gamma \vdash t : A$, where Γ is a context of variables with their associated types.

Boxing and Unboxing. The `Box` type bridges \mathcal{QWIRE} circuits and HOST terms. The type $\text{Box } W_1 W_2$ is a wrapper around \mathcal{QWIRE} circuits of the form $\Gamma; \Omega \vdash C : W_2$, where the wires in Ω come from a pattern destructuring the input type W_1 .

$$\frac{\Omega \Rightarrow p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \text{box } (p : W_1) \Rightarrow C : \text{Box } W_1 W_2}$$


A boxed term of type $\text{Box } W_1 W_2$ can be turned back into a \mathcal{QWIRE} circuit by describing how to match up the available input wires to the input type of the boxed representation.

$$\frac{\Gamma \vdash t : \text{Box } W_1 W_2 \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \text{unbox } t p : W_2}$$


Lifting. In the QRAM model described above, the quantum computer also communicates with the classical computer by sending it the results of measurement. For example, given a circuit with no input wires and a bit output, *running* that circuit should result in a host language boolean value. (Note that this is probabilistic operation and hence impure. In a pure language we would require a probability monad \mathbb{M} and `run C` would have type $\mathbb{M} \text{ Bit}$.)

$$\frac{\Gamma; \cdot \vdash C : \text{Bit}}{\Gamma \vdash \text{run } C : \text{Bool}}$$

We can generalize this operation so that running a circuit that outputs a qubit implicitly measures that qubit and returns the corresponding boolean. In fact, this relationship generalizes to any wire type, which can be *lifted* to a classical type as follows:

$$\begin{array}{ll} |\text{Bit}| = \text{Bool} & |\text{One}| = \text{Unit} \\ |\text{Qubit}| = \text{Bool} & |W_1 \otimes W_2| = |W_1| \times |W_2| \end{array}$$

The *run* operator now has the following form:

$$\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \text{run } C : |W|}$$

Run is a *static lifting* operator, meaning that there is no residual state left on the quantum computer after `run C` has completed. In contrast, *dynamic lifting* describes the case when, over the course of a quantum computation, a subset of the wires are measured and communicated to the classical computer. In this case, the classical computer uses those results to compute the remainder of the quantum circuit, and eventually sends the results to the quantum computer. Dynamic lifting is expensive because while the classical computer is computing the rest of the circuit, the existing state on the quantum computer must continuously undergo error correction to prevent degradation. However, dynamic lifting is a fundamental form of communication between the two machines, needed to implement algorithms like quantum error correction.

We write the dynamic lifting operator $x \leftarrow \text{lift } p; C$ to mean that the wires in p are measured, lifted to the classical computer as the host variable x , and used to compute the circuit C .

$$\frac{\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \leftarrow \text{lift } p; C : W'}$$

The dynamic and static lifting operations are not mutually derivable, as they represent two fundamentally different ways to communicate the results of measurement between the two systems.

4.3.3 Static Semantics

To summarize, the syntax of `QWIRE` circuits and `HOST` terms include the following:

$$\begin{array}{l} (\text{Patterns}) \quad p ::= () \mid w \mid (p, p) \\ (\text{Circuits}) \quad C ::= \text{output } p \mid p_2 \leftarrow \text{gate } g \ p_1; C \mid p \leftarrow C; C \mid x \leftarrow \text{lift } p; C \mid \text{unbox } t \ p \\ (\text{Terms}) \quad t ::= \dots \mid \text{run } C \mid \text{box } (p : W) \Rightarrow C \end{array}$$

The typing rules are summarized in Figure 4.2. Note that we often write `box p ⇒ C` instead of `box (p : W) ⇒ C` when the type of the input pattern is clear. Note that typing contexts are unique for both patterns and circuits:

$$\begin{array}{c}
\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \text{output } p : W} \text{TYPECIRCOUTPUT} \\
\\
\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad \Gamma; \Omega_2, \Omega \vdash C : W}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W} \text{TYPECIRCGATE} \\
\\
\frac{\Gamma; \Omega_1 \vdash C : W \quad \Gamma; \Omega_0, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \text{TYPECIRCLET} \\
\\
\frac{\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \leftarrow \text{lift } p; C : W'} \text{TYPECIRCLIFT} \\
\\
\frac{\Gamma \vdash t : \text{Box } W_1 \ W_2 \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \text{unbox } t \ p : W_2} \text{TYPECIRCUNBOX} \\
\\
\hline
\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \text{run } C : |W|} \text{TYPERUN} \\
\\
\frac{\Omega \Rightarrow p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \text{box } (p : W_1) \Rightarrow C : \text{Box } W_1 \ W_2} \text{TYPEBOX}
\end{array}$$

Figure 4.2: Typing rules for \mathcal{Q} WIRE.

Lemma 1. *If $\Omega_1 \Rightarrow p : W$ and $\Omega_2 \Rightarrow p : W$ then $\Omega_1 = \Omega_2$. If $\Gamma; \Omega_1 \vdash C : W$ and $\Gamma; \Omega_2 \vdash C : W$ then $\Omega_1 = \Omega_2$.*

4.4 Operational Semantics: Circuit Normalization

Circuits in \mathcal{Q} WIRE represent instructions to be executed on a quantum computer: either apply a particular gate or request a dynamic lifting operation. Composition and unbox operations are more like meta-operations: they describe ways to construct complex combinations of gates. In this section we define an operational semantics that eliminates all instances of unboxing and composition, resulting in a small set of normal forms. A \mathcal{Q} WIRE circuit in normal form is identified by two main properties.

First, normal circuits should operate only on bits and qubits, not on the tuples of wires described by arbitrary wire types W . We call a circuit *concrete* when all of its input wires are either bits or qubits:

$$;\mathcal{Q} \vdash C : W \quad \text{where} \quad \mathcal{Q} ::= \cdot \mid \mathcal{Q}, w : \text{Bit} \mid \mathcal{Q}, w : \text{Qubit}.$$

A concrete circuit is called *normal* when it consists only of gate applications, outputs, and dynamic lifting operations.

$$N ::= \text{output } p \mid p_2 \leftarrow \text{gate } g \ p_1; N \mid x \leftarrow \text{lift } p; C$$

Notice that the lifting operator $x \leftarrow \text{lift } p; C$ does not assume that its continuation C is also normal. This is because C has a free host-level variable x that cannot in general be normalized. For example, consider the circuit $x \leftarrow \text{lift } w; \text{unbox } (\text{init } x) ()$: the continuation $\text{unbox } (\text{init } x) ()$ cannot be normalized to init_0 or init_1 since x is not assigned a value until circuit execution time.

In the rest of this section, we define a small-step operational semantics that reduces concrete circuits typed by $;\mathcal{Q} \vdash C : W$ to normal circuits. The operational rules rely on a fairly complex substitution relation, which we briefly address.

Substitution. A substitution $\{p'/p\}$ describes a finite map from wire names to patterns. It is well defined only when p generalizes p' (written $p' \preceq p$) in the following sense:

$$\frac{}{p' \preceq w} \quad \frac{}{() \preceq ()} \quad \frac{p'_1 \preceq p_1 \quad p'_2 \preceq p_2}{(p'_1, p'_2) \preceq (p_1, p_2)}$$

We say $p' < p$ when $p' \preceq p$ and $\neg(p \preceq p')$, and we say p is *concrete* for W when, for all $\Omega \Rightarrow p' : W$, $\neg(p' < p)$.

Lemma 2. *If $\Omega \Rightarrow p : W$ and $\mathcal{Q} \Rightarrow p' : W$, then $p' \preceq p$.*

The substitution map is defined as follows:

$$\begin{aligned} \{()\}/\{()\} &= \emptyset \\ \{p'/w\} &= w \mapsto p' \\ \{(p'_1, p'_2)/\}(p_1, p_2) &= \{p'_1/p_1\}, \{p'_2/p_2\} \end{aligned}$$

A well-defined substitution extends to *total* functions on patterns, circuits, and wire contexts. For patterns, we have:

$$\begin{aligned} ()\{p'/p\} &= () \\ w\{p'/p\} &= \begin{cases} p_0 & \text{if } w \mapsto p_0 \in \{p'/p\} \\ w & \text{otherwise} \end{cases} \\ (p_1, p_2)\{p'/p\} &= (p_1\{p'/p\}, p_2\{p'/p\}) \end{aligned}$$

The operation on circuits is straightforward, assuming the standard notions of capture-avoidance.

$$\begin{aligned} (\text{output } p_0)\{p'/p\} &= \text{output } (p_0\{p'/p\}) \\ (p_2 \leftarrow \text{gate } g \ p_1; C)\{p'/p\} &= p_2 \leftarrow \text{gate } g \ p_1\{p'/p\}; C\{p'/p\} \\ (x \leftarrow \text{lift } p_0; C)\{p'/p\} &= x \leftarrow \text{lift } p_0\{p'/p\}; C\{p'/p\} \\ (\text{unbox } t \ p_0)\{p'/p\} &= \text{unbox } t \ (p_0\{p'/p\}) \\ (p_0 \leftarrow C; C')\{p'/p\} &= p_0 \leftarrow C\{p'/p\}; C'\{p'/p\} \end{aligned}$$

A well-defined substitution $\{p'/p\}$ is *consistent with* w at W if $(w \mapsto p_0) \in \{p'/p\}$ implies that there is some unique³ Ω_0 such that $\Omega_0 \Rightarrow p_0 : W$. A substitution is consistent with a context Ω when, for all $w : W \in \Omega$, it is consistent with w at W .

For wire contexts, suppose $\{p'/p\}$ is consistent with Ω . The substitution $\Omega\{p'/p\}$ is defined by induction on Ω :

$$\begin{aligned} \cdot\{p'/p\} &= \cdot \\ (\Omega', w : W)\{p'/p\} &= \begin{cases} \Omega'\{p'/p\}, \Omega_0 & \text{if } w \mapsto p_0 \in \{p'/p\} \text{ and } \Omega_0 \Rightarrow p_0 : W \\ \Omega'\{p'/p\}, w : W & \text{otherwise} \end{cases} \end{aligned}$$

Lemma 3. *Suppose $p' \preceq p$ where $\Omega \Rightarrow p : W$ and $\Omega' \Rightarrow p' : W$. Then:*

1. *If Ω'' is disjoint from Ω , then $\Omega''\{p'/p\} = \Omega''$.*
2. *$\Omega\{p'/p\} = \Omega'$.*
3. *$(\Omega_1, \Omega_2)\{p'/p\} = \Omega_1\{p'/p\}, \Omega_2\{p'/p\}$.*

³Recall that Ω_0 is uniquely determined by the choice of p_0 and W (Lemma 1).

Lemma 4. *Suppose $\{p'/p\}$ is consistent with Ω .*

1. *If $\Omega \Rightarrow p_0 : W$ then $\Omega \{p'/p\} \Rightarrow p_0 \{p'/p\} : W$.*
2. *If $\Gamma; \Omega \vdash C : W$ then $\Gamma; \Omega \{p'/p\} \vdash C \{p'/p\} : W'$.*

Proof. Part 1 is straightforward by induction. Part 2 is similarly by induction on the typing judgment $\Gamma; \Omega \vdash C : W$. The only difficult case concerns the bound patterns in gate and composition substitutions. For example, consider the gate application rule:

$$\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad ; \Omega_2, \Omega \vdash C : W}{; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W}$$

By part 1, we have $\Omega_1 \{p'/p\} \Rightarrow p_1 \{p'/p\} : W_1$, and by the inductive hypothesis we know $\Gamma; (\Omega_2, \Omega) \{p'/p\} \vdash C \{p'/p\} : W$. By α -equivalence, we can assume that the wires in Ω_2 are disjoint from those in p' and p (and therefore from the substitution $\{p'/p\}$), and so by Lemma 3, $(\Omega_2, \Omega) \{p'/p\} = \Omega_2, (\Omega \{p'/p\})$. Thus

$$\Gamma; \Omega_1 \{p'/p\}, \Omega \{p'/p\} \vdash p_2 \leftarrow \text{gate } g \ p_1 \{p'/p\}; C \{p'/p\} : W. \quad \square$$

Operational Semantics. The small-step operational semantics for circuits is written $C \Longrightarrow C'$, and it depends on a similar operational semantics on terms, written $t \longrightarrow t'$. The relation on terms is made up of two parts, $\longrightarrow = \longrightarrow_{\text{H}} \cup \longrightarrow_b$, where

1. $\longrightarrow_{\text{H}}$ is the operational semantics derived from the host language alone, and
2. \longrightarrow_b is the operational semantics for boxed circuits.

It is reasonable to assume that the host language relation $\longrightarrow_{\text{H}}$ treats the type $\text{Box } W_1$ W_2 as an abstract data type, meaning that all terms of the form $\text{box } p \Rightarrow C$ are treated as opaque values by the $\longrightarrow_{\text{H}}$ relation. The relation \longrightarrow_b reduces such a boxed circuit to one of the form $\text{box } p' \Rightarrow N$ where p' is concrete for the type W_1 . Let v^{H} consists of values of HOST extended with boxed circuits as opaque values, and v^{C} consists of HOST's values along with normalized boxed circuits:

$$\begin{aligned} v^{\text{H}} &::= \dots \mid \text{box } p \Rightarrow C \\ v^{\text{C}} &::= \dots \mid \text{box } p \Rightarrow N \end{aligned}$$

We explicitly do not describe the operational behavior of $\text{run } C$ terms in this semantics. Instead, we assume that run operations reduce under $\longrightarrow_{\text{H}}$: The host language can execute or simulate quantum circuits. The semantics of run is distinct from the *construction* of circuits, which is what we are developing in this section. A candidate semantics for circuit evaluation is given in Section 4.5.1, where we provide of a probabilistic operational rule for $\text{run } C$ based on the denotational semantics of circuits.

(Box)

$$\frac{p \text{ is concrete for } W \quad C \Longrightarrow C'}{\text{box } (p: W) \Rightarrow C \longrightarrow_b \text{box } (p: W) \Rightarrow C'} \quad \text{STEPTERMBOXSTRUCTURAL}$$

$$\frac{p' < p \quad p' \text{ is concrete for } W}{\text{box } (p: W) \Rightarrow C \longrightarrow_b \text{box } (p': W) \Rightarrow C \{p'/p\}} \quad \text{STEPTERMBOXETA}$$

(Unbox)

$$\frac{t \longrightarrow t'}{\text{unbox } t \ p \Longrightarrow \text{unbox } t' \ p} \quad \text{STEPCIRCUNBOXSTRUCTURAL}$$

$$\frac{}{\text{unbox } (\text{box } (p: W) \Rightarrow N) \ p' \Longrightarrow N \ {p'/p}} \quad \text{STEPCIRCUNBOX}$$

(Gate)

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p_2 \text{ is concrete for } W_2 \quad C \Longrightarrow C'}{p_2 \leftarrow \text{gate } g \ p_1; C \Longrightarrow p_2 \leftarrow \text{gate } g \ p_1; C'} \quad \text{STEPCIRCGATESTRUCTURAL}$$

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p'_2 < p_2 \quad p'_2 \text{ is concrete for } W_2}{p_2 \leftarrow \text{gate } g \ p_1; C \Longrightarrow p'_2 \leftarrow \text{gate } g \ p_1; C \ {p'_2/p_2}} \quad \text{STEPCIRCGATEETA}$$

(Composition)

$$\frac{C_1 \Longrightarrow C'_1}{p \leftarrow C_1; C_2 \Longrightarrow p \leftarrow C'_1; C_2} \quad \text{STEPCIRCLETSTRUCTURAL}$$

$$\frac{}{p \leftarrow \text{output } p'; C \Longrightarrow C \ {p'/p}} \quad \text{STEPCIRCLETOUTPUT}$$

$$\frac{}{p \leftarrow (p_2 \leftarrow \text{gate } g \ p_1; N); C \Longrightarrow p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow N; C} \quad \text{STEPCIRCLETGATE}$$

$$\frac{}{p' \leftarrow (x \leftarrow \text{lift } p; C'); C \Longrightarrow x \leftarrow \text{lift } p; p' \leftarrow C'; C} \quad \text{STEPCIRCLETMEAS}$$

Figure 4.3: Operational semantics of concrete circuits.

The relations \Longrightarrow on circuits and \longrightarrow_b on boxed circuits are given in Figure 4.3. The structural rules reduce circuits underneath binders. For composition and unboxing, these structural rules are straightforward, in that they don't have any preconditions restricting when they apply. For boxes and gates, on the other hand, the continuations C of the circuit have some additional inputs that are not concrete even if the entire circuit is. For example, in the circuit $w \leftarrow \text{gate CNOT } (w_1, w_2); C$, the continuation C has a compound wire w even though the entire circuit has only concrete wires w_1 and w_2 . To address this issue, the η -expansion rules for gates and boxes show that any such binding is equivalent to one with concrete inputs throughout.

Lemma 5. *If p is concrete for W then there is a unique Q such that $Q \Rightarrow p : W$. Furthermore, for every wire type W there exists p (not necessarily unique) such that p is concrete for W .*

Since an unbox operator is not a normal circuit, we eliminate it via a β rule once its argument t reaches a value of the form $\text{box } p \Rightarrow N$. Similarly, the composition operator reduces its first argument to a normal form before taking a step. When the argument is an output $\text{output } p'$, the composition $p \leftarrow \text{output } p'; C$ uses substitution to take a β -reduction step. On the other hand, when the argument consists of a gate or lifting step, the semantics *commutes* that command to the front of the circuit; we call these operators *commuting conversions*.

4.4.1 Type Safety

We prove type safety with progress and preservation theorems, provided that the relation \longrightarrow_H is also type safe.

Theorem 5 (Preservation). *Suppose \longrightarrow_H satisfies preservation.*

1. *If $\vdash t : A$ and $t \longrightarrow t'$, then $\vdash t' : A$.*
2. *If $;\mathcal{Q} \vdash C : W$ and $C \Longrightarrow C'$, then $;\mathcal{Q} \vdash C' : W$.*

Proof. By induction on the step relation (Appendix B.1). □

Theorem 6 (Progress). *Suppose \longrightarrow_H satisfies progress with respect to the values v^H .*

1. *If $\vdash t : A$ then either t is a value v^C or there is some t' such that $t \longrightarrow t'$.*
2. *If $;\mathcal{Q} \vdash C : W$ then either C is normal or there is some C' such that $C \Longrightarrow C'$.*

Proof. By induction on the typing judgment (Appendix B.1). □

Provided that \longrightarrow_H is strongly normalizing, we can also show that circuits are strongly normalizing.

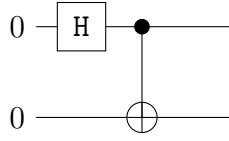
Theorem 7 (Normalization). *Suppose that \longrightarrow_H is strongly normalizing with respect to v^H .*

1. If $\cdot \vdash t:A$, there exists some value v^c such that $t \longrightarrow^* v^c$.
2. If $\cdot; \mathcal{Q} \vdash C:W$, there exists some normal circuit N such that $C \Longrightarrow^* N$.

Proof. By induction on the number of constructors in the term and circuit (Appendix B.1). \square

4.5 Denotational Semantics

In this section we give a denotational semantics for \mathcal{Q} WIRE circuits. The state of a quantum system can be described in terms of the density matrices of Section 2.5. For example, consider the entangled Bell pair produced by the following circuit:



This pair of qubits is represented by the (pure state) density matrix

$$\begin{pmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \end{pmatrix}$$

where the $1/2$ in the top left represents the probability of measuring two zeros, while the $1/2$ in the bottom right represents the probability of measuring two ones. If we measured this system, we would obtain the (mixed state) matrix

$$\begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix}$$

representing a distribution over $|00\rangle$ and $|11\rangle$.

Since a \mathcal{Q} WIRE circuit transforms some state to another, it will be interpreted as a superoperator over density matrices (a function on density matrices that preserves mixed states).

To begin, we define the denotation of a k -(qu)bit wire type as the corresponding set of $2^k \times 2^k$ matrices:

$$\begin{aligned} \llbracket \text{Bit} \rrbracket &= \mathbb{C}^{2,2} & \llbracket \text{One} \rrbracket &= \mathbb{C}^{1,1} \\ \llbracket \text{Qubit} \rrbracket &= \mathbb{C}^{2,2} & \llbracket W_1 \otimes W_2 \rrbracket &= \llbracket W_1 \rrbracket \otimes \llbracket W_2 \rrbracket \end{aligned}$$

In practice, inhabitants of $\llbracket \text{One} \rrbracket$ will be restricted to the 1×1 identity matrix and inhabitants of $\llbracket W \rrbracket$ will all be density matrices. With this, we can treat a gate

$g \in \mathcal{G}(W_1, W_2)$ as a superoperator from $\llbracket W_1 \rrbracket$ to $\llbracket W_2 \rrbracket$. Although the set of gates is a parameter of the system, the denotation of a unitary gate U applied to a matrix ρ should always be $U\rho U^\dagger$. The interpretation of other likely gates is as follows:

$$\begin{aligned} \llbracket \text{new}_0 \rrbracket(c) &= \llbracket \text{init}_0 \rrbracket(c) = c|0\rangle\langle 0| \\ \llbracket \text{new}_1 \rrbracket(c) &= \llbracket \text{init}_1 \rrbracket(c) = c|1\rangle\langle 1| \\ \llbracket \text{discard} \rrbracket\rho &= \langle 0|\rho|0\rangle + \langle 1|\rho|1\rangle \\ \llbracket \text{meas} \rrbracket\rho &= |0\rangle\langle 0|\rho|0\rangle\langle 0| + |1\rangle\langle 1|\rho|1\rangle\langle 1| \end{aligned}$$

\mathcal{Q} WIRE circuits are specified by an unordered context of input wires Ω . However, we can equally well think of Ω as an *ordered* context, along with an explicit permutation rule to change the order of the wires.⁴

$$\frac{\Gamma; \Omega' \vdash C : W \quad \pi : \Omega \equiv \Omega'}{\Gamma; \Omega' \vdash C : W}$$

Permutations are defined inductively.

$$\frac{}{\epsilon : \Omega \equiv \Omega} \quad \frac{}{\text{swap } \Omega_1 \ \Omega_2 : \Omega, \Omega_1, \Omega_2, \Omega' \equiv \Omega, \Omega_2, \Omega_1, \Omega'} \quad \frac{\pi_1 : \Omega_1 \equiv \Omega_2 \quad \pi_2 : \Omega_2 \equiv \Omega_3}{\pi_2 \circ \pi_1 : \Omega_1 \equiv \Omega_3}$$

Note that permutations are reflected in the typing judgments of circuits but not in the syntax. We extend the substitution relation to permutations in a natural way, writing $\pi \{p'/p\}$.

$$\begin{aligned} \epsilon \{p'/p\} &= \epsilon \\ (\pi_2 \circ \pi_1) \{p'/p\} &= \pi_2 \{p'/p\} \circ \pi_1 \{p'/p\} \\ (\text{swap } \Omega_1 \ \Omega_2) \{p'/p\} &= \text{swap } (\Omega_1 \{p'/p\}) \ (\Omega_2 \{p'/p\}) \end{aligned}$$

An ordered context of wires is now interpreted as a space of matrices by treating the comma as the tensor product:

$$\llbracket \cdot \rrbracket = \mathbb{C}^{1,1} \quad \llbracket w : W \rrbracket = \llbracket W \rrbracket \quad \llbracket \Omega_1, \Omega_2 \rrbracket = \llbracket \Omega_1 \rrbracket \otimes \llbracket \Omega_2 \rrbracket$$

Although the context of wires can be permuted inside a circuit, it will not be permuted inside a pattern. Therefore, a pattern $\Omega \Rightarrow p : W$ is just a reassociation of the input wires; all permutations must be done outside the pattern. This means that whenever $\Omega \Rightarrow p : W$, it must be the case that $\llbracket \Omega \rrbracket = \llbracket W \rrbracket$.

A permutation $\pi : \Omega \equiv \Omega'$ will be interpreted as a linear isomorphism from $\llbracket \Omega \rrbracket$ to

⁴We elided these details in Section 4.3 as they complicate the operational semantics.

$$\begin{array}{c}
\frac{\Omega \Rightarrow p : W}{\cdot; \Omega \vdash \text{output } p : W} \quad \llbracket \cdot; \Omega \vdash \text{output } p : W \rrbracket = \mathbf{I}^* \\
\\
\frac{\cdot; \Omega \vdash C : W \quad \pi : \Omega \equiv \Omega'}{\cdot; \Omega \vdash C : W} \quad \llbracket \Omega \vdash C : W \rrbracket = \llbracket \Omega' \vdash C : W \rrbracket \circ [\pi]^* \\
\\
\frac{\cdot \vdash t : \text{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\cdot; \Omega \vdash \text{unbox } t \ p : W_2} \quad \llbracket \Omega \vdash \text{unbox } t \ p : W' \rrbracket = \llbracket t : \text{Circ}(W, W') \rrbracket \\
\\
\frac{g \in \mathcal{G}(W_1, W_2) \quad \Omega_1 \Rightarrow p_1 : W_1 \quad \cdot; \Omega_2, \Omega \vdash C : W}{\cdot; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W} \quad \llbracket \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W \rrbracket \\
= \llbracket \Omega_2, \Omega \vdash C : W \rrbracket \circ (\llbracket g \rrbracket \otimes \mathbf{I}^*) \\
\\
\frac{\Omega \Rightarrow p : W \quad x : |W|; \Omega' \vdash C : W'}{\cdot; \Omega, \Omega' \vdash x \leftarrow \text{lift } p ; C : W'} \quad \llbracket \Omega, \Omega' \vdash x \leftarrow \text{lift } p ; C : W' \rrbracket \\
= \sum_{\vdash |W|} \llbracket \Omega' \vdash C \{v/x\} : W' \rrbracket \circ (\llbracket v : |W| \rrbracket^\dagger \otimes \mathbf{I}^*) \\
\\
\frac{\cdot; \Omega_1 \vdash C : W \quad \cdot; \Omega_0, \Omega_2 \vdash C' : W'}{\cdot; \Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W'} \quad \llbracket \Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W' \rrbracket \\
= \llbracket \Omega_0, \Omega_2 \vdash C' : W' \rrbracket \circ (\llbracket \Omega_1 \vdash C : W \rrbracket \otimes \mathbf{I}^*)
\end{array}$$

Figure 4.4: Denotational semantics of circuits.

$\llbracket \Omega' \rrbracket$, written $\llbracket \pi \rrbracket$, as follows:

$$\begin{aligned}
\llbracket \epsilon \rrbracket &= I & \llbracket \pi_2 \circ \pi_1 \rrbracket &= \llbracket \pi_2 \rrbracket \circ \llbracket \pi_1 \rrbracket \\
\llbracket \text{swap } \Omega_1 \ \Omega_2 \rrbracket(v_0 \otimes v_1 \otimes v_2 \otimes v_3) &= (v_0 \otimes v_2 \otimes v_1 \otimes v_3)
\end{aligned}$$

Lemma 6. *If $\pi : \Omega \equiv \Omega'$ and $\{p'/p\}$ is consistent with Ω , then $\llbracket \pi \{p'/p\} \rrbracket = \llbracket \pi \rrbracket$.*

Proof. Straightforward by induction on the permutation. \square

For $\cdot; \Omega \vdash C : W$, we write $\llbracket \Omega \vdash C : W \rrbracket$ for its interpretation as a superoperator between $\llbracket \Omega \rrbracket$ and $\llbracket W \rrbracket$. Furthermore, for $\cdot \vdash t : \text{Box } W_1 \ W_2$, we write $\llbracket t \rrbracket$ for $\llbracket \Omega \vdash C : W_2 \rrbracket$ where $t \xrightarrow{*}_{\text{H}} \text{box } p \Rightarrow C$ in the host language and $\Omega \Rightarrow p : W_1$. This operation is functional exactly when the host language semantics is strongly normalizing.

The interpretation of circuits is defined in Figure 4.4.

Lemma 7. *If $\cdot; \Omega \vdash C : W$ and $\{p'/p_0\}$ is consistent with Ω , then*

$$\llbracket \Omega \{p'/p\} \vdash C \{p'/p\} : W \rrbracket = \llbracket \Omega \vdash C : W \rrbracket.$$

Proof. By induction on the typing judgment. The proof is almost completely straightforward because the interpretation of circuits does not depend on the content of

patterns. □

Theorem 8 (Soundness). *If $\cdot; \mathcal{Q} \vdash C : W$ and $C \Longrightarrow C'$, then*

$$\llbracket \mathcal{Q} \vdash C : W \rrbracket = \llbracket \mathcal{Q} \vdash C' : W \rrbracket.$$

Proof. By induction on the typing judgment (Appendix B.2). □

4.5.1 Operational Behavior of *run*

In Section 4.4, we left the semantics of the *run* operator up to the the host language and corresponding quantum device or simulator. Given the denotational semantics described in this section, however, we can specify the correctness of *run C* as a probabilistic operation.

First, we will need to define a lifting from select HOST level terms (the constructors of **Unit**, **Bool** and $A \times A$) to basis states that follows naturally from our lifted wire types (Section 4.3.2):

$$\begin{aligned} [\ast] &= I_1 \\ [\text{false}] &= |0\rangle \langle 0| \\ [\text{true}] &= |1\rangle \langle 1| \\ [(v_1, v_2)] &= [v_1] \otimes [v_2] \end{aligned}$$

Now, if $\cdot; \cdot \vdash C : W$, then

$$\llbracket \cdot \vdash C : W \rrbracket I_1$$

is a density matrix in $\llbracket W \rrbracket$. We can write this density matrix as

$$\begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{pmatrix}$$

Recall that the elements α_{ii} along the diagonal correspond to basis states, and therefore to terms v_i in the host language (via lowering). Hence, for each i , we say that the probability of C being v_i is α_{ii} , written $\text{prob}(C = v_i) = \alpha_{ii}$. The operational semantics rule for *run C* can be summarized with respect to this relation: *run C* steps to v_i with probability α_{ii} .

$$\frac{\text{prob}(C = v_i) = \alpha_{ii}}{\text{run } C \longrightarrow^{\alpha_{ii}} v_i}$$

4.6 A Categorical Semantics for Qwire

In “Classical Control and Quantum Circuits in Enriched Category Theory,” Rennela and Staton (2017) provide a categorical semantics for EWire, which slightly general-

izes \mathcal{Q} WIRE. \mathcal{E} Wire is shown to be an *enriched category* in which the morphisms of one category become objects in another. In context, we can view a circuit as a function (morphism) from wires types to wire types, but we also treat circuits as terms (objects) in the host language. As a result, we can impose a layered categorical structure on \mathcal{Q} WIRE: Circuits correspond to an enriched symmetric monoidal category and the host language corresponds to Cartesian closed category with the `run` monad connecting the two.

This categorical framework allows the authors to consider adding features to \mathcal{Q} WIRE, such as recursion, which is modeled using directed complete partial orders (DCPOs). This builds on previous work by Rennela (2014) that uses W^* -algebras to model quantum computation. Further work by Lindenhovius et al. (2018) builds upon this semantics in the contexts of the Proto-Quipper language (Ross, 2015; Rios and Selinger, 2017).

4.7 Dependent Types

One powerful feature that we might request from the host language is support for dependent types. Consider the quantum Fourier transform, which is a circuit with n inputs and n outputs. It is natural for the wire types of the Fourier circuit to reflect this dependency on n . In the language of dependent types, it might have the signature

`fourier` : $\Pi(n:\mathbb{N}). \text{Box } (n \otimes \text{Qubit}) (n \otimes \text{Qubit})$

where `tensor` (\otimes) is a type-level function that duplicates the argument wire type (`Qubit`) some number of times (defined below).

Combining linear and dependent types is still an area of active research (Krishnaswami et al., 2015; McBride, 2016), but thanks to the separation between the circuit and host languages, we can get away with a limited form of dependent types due to Krishnaswami et al. (2015). Under this strategy, types can depend on terms, but only terms of *classical* (non-linear) type. These include dependencies on wire types themselves.

To be precise, let \mathcal{W} be the kind of wire types, and consider an indexed hierarchy of host language types \mathcal{A}_i . We define the following well-formedness judgment: first, \mathcal{W} has type \mathcal{A}_i for any index i , and \mathcal{A}_i has type \mathcal{A}_{i+1} :

$$\frac{}{\Gamma \vdash \mathcal{W} : \mathcal{A}_i} \quad \frac{}{\Gamma \vdash \mathcal{A}_i : \mathcal{A}_{i+1}}$$

In addition, we introduce a new host-language type $\Pi(x:A).B$ with the following well-formedness condition:

$$\frac{\Gamma \vdash \mathcal{W} : \mathcal{A}_i \quad \Gamma, x : A \vdash B : \mathcal{A}_i}{\Gamma \vdash \Pi(x : A).B : \mathcal{A}_i}$$

Π types have the usual introduction and elimination rules:

$$\frac{\Gamma, x : A_1 \vdash t : A_2}{\Gamma \vdash \lambda x. t : \Pi(x : A_1). A_2} \quad \frac{\Gamma \vdash t : \Pi(x : A_1). A_2 \quad \Gamma \vdash t' : A_1}{\Gamma \vdash t t' : A_2\{t'/x\}}$$

A more thorough analysis of this type structure is needed, but it is beyond the scope of this chapter.

A Dependent Quantum Fourier Transform. Under this framework, we can start with the type-level function `tensor`, a recursive function which pattern matches on a natural number n :

```
tensor (n : ℕ) (W : WType) : WType :=
  match n with
  | 0      => One
  | 1      => W
  | 1 + n' => W ⊗ tensor n' W
end
```

We write $n \otimes W$ for `tensor n W`.

Next, we use these length-indexed tuples to write a dependently-typed quantum Fourier transform in the style of Green et al. (2013b). Our version of the Fourier circuit ensures that the number of qubits in the input and output are always the same.

First, we define the rotation circuits. We assume the presence of a family of gates `RGate m` that rotates its input along the z -axis by $\frac{2\pi i}{2^m}$ (Green et al., 2013b). The `rotations` circuit takes two natural number inputs: m , the argument given to the controlled R gates; and n , the number of bits in the input.

```
rotations (m:ℕ) : Π(n:ℕ). Box ((n+1) ⊗ Qubit) ((n+1) ⊗ Qubit) :=
  fun n => match n with
    | 0      => id
    | 1      => id
    | 1 + n' => box (c, (q, qs)) =>
      (c, qs) ← unbox rotations m n' (c, qs);
      (c, q)  ← gate (control (RGate (2+m-n')))(c, q);
      output (c, (q, w))
  end
```

The quantum Fourier transform can now be defined in a type-safe way:

```
qft : Π(n:ℕ). Box (n ⊗ Qubit) (n ⊗ Qubit) :=
  fun n =>
  match n with
  | 0 => id
  | 1 => hadamard
  | 1 + n' => box (q, w) =>
```

```

      w ← unbox qft n' w;
      unbox rotations (S n') n' (q,w)
end

```

where hadamard = box w => gate H w.

4.8 Summary

QWIRE is a minimal and highly modular core circuit language. It is minimal in that *QWIRE* has only five distinct commands, two of which are eliminated in the normalization procedure. It is modular in that *QWIRE* isn't attached to any specific programming language. We expect that the *QWIRE* interface will be useful in dependently-typed host languages like Coq for verification and formal analysis of circuits, in higher-order functional languages like Haskell, OCaml or F#, or potentially even in imperative languages like Python, Java, or C.

QWIRE uses linear types to enforce no-cloning, but it does not allow them to spill over into the host language. This is crucial because linear types are the most natural way to enforce no-cloning, but they are difficult to integrate into existing languages. *QWIRE* gets the best of both worlds by ensuring that circuits are linearly typed while allowing an arbitrarily powerful type system in the classical host language.

As a circuit description language, *QWIRE* is a low-level piece in the development of sophisticated quantum programming languages. Ultimately however, all quantum computation will boil down to circuit generation, necessitating the use of a circuit language like *QWIRE*. Having *QWIRE* as a safe, small circuit language is an excellent building block on which to rest the complex world of quantum computation.

In the following chapters, we will focus on a particular implementation of *QWIRE* inside the Coq proof assistant and what we gain from this embedding, in terms of dependently typed structures and a formally verified metatheory.

Chapter 5

Qwire in Practice

This chapter discusses the implementation of `QWIRE` inside the Coq proof assistant (Coq Development Team, 2018). Coq consists of both a programming language, called Gallina, and an interactive system for proving properties of Gallina programs. Gallina features *dependent types*, which may depend on both terms and other types from the language. For instance, we can define the type `Sized_List 7 B`, for lists of length 7 that contain boolean values. Gallina is also restrictive in that all Gallina programs must terminate, and hence only restricted forms of recursion are permitted. While the Coq proof language may appear to be separate from Gallina, it actually constructs dependently typed Gallina terms, which correspond to proofs of desired properties. Coq also includes an untyped programming language, called Ltac, which is used to construct Coq proofs. We assume some familiarity with Coq in this dissertation, and refer the interested reader to the online textbook / Coq library “Software Foundations” (Pierce et al., 2018) or our Coq tutorial with Arthur Azevedo de Amorim (Rand and de Amorim, 2016).

Implementing `QWIRE` forced us to confront issues like variable representation, compilation of circuits to functions on density matrices, and implementing matrices and complex numbers in Coq. This chapter draws on Rand et al. (2017) along with a CoqPL workshop presentation on phantom types for quantum programs (Rand et al., 2018b).

5.1 Circuits in Coq

Wires and Gates As we saw in the previous chapter, at its core a `QWIRE` circuit is a sequence of gates applied to wires. Each wire is described by a *wire type* `W`, which is either the unit type (no data), a bit or qubit, or a tuple of wire types. In Coq we represent wire types as an inductively defined data type `WType` as follows:

Inductive `WType := One | Bit | Qubit | Tensor : WType → WType → WType.`

We use the Coq notation `W1 ⊗ W2` for `Tensor W1 W2`.

Wires in a circuit can be collected into *patterns* corresponding to a given wire type, written `Pat W`. A pattern is just a nested tuple of wires of base types, meaning that all the variables in a pattern have type `Bit` or `Qubit`.

```
Inductive Pat : WType → Set :=
| unit   : Pat One
| qubit  : Var → Pat Qubit
| bit    : Var → Pat Bit
| pair   : ∀ {W1 W2}, Pat W1 → Pat W2 → Pat (W1 ⊗ W2).
```

The type `Var` is identical to `N`. We use the notation `()` to refer to the unit pattern and `(p1,p2)` to refer to `pair p1 p2`. Note that the wire types corresponding to `unit`, `qubit` and `bit` are fixed; the type of a pair can therefore be inferred from the patterns it contains.

Gates are indexed by a pair of wire types—a gate of type `Gate W1 W2` takes an input wire of type `W1` and outputs a wire of type `W2`. In our setting, gates will include a universal set of unitary gates, as well as initialization, measurement, and control.

```
Inductive Unitary : WType → Set :=
| H      : Unitary Qubit (* Hadamard *)
| X      : Unitary Qubit (* Pauli X *)
| Y      : Unitary Qubit (* Pauli Y *)
| Z      : Unitary Qubit (* Pauli Z *)
| control : ∀ {W} (U : Unitary W), Unitary (Qubit ⊗ W).
```

```
Inductive Gate : WType → WType → Set :=
| U      : ∀ {W} (u : Unitary W), Gate W W
| new0  : Gate One Bit
| init0 : Gate One Qubit
| meas   : Gate Qubit Bit
| discard : Gate Bit One.
```

We define `U` to be a *coercion* from unitaries to gates, meaning that for any `u` of type `Unitary W`, we can simply write `u` for the gate `U u`.

Circuits In constructing our circuits, we represent variable using higher-order abstract syntax (HOAS) (Pfenning and Elliott, 1988), in which variable bindings in an embedded language are represented as functions in the host language. This saves us from having to define and prove the correctness of our own substitution function and gives variables in the embedded language the same weight as variables in the host language.

There are only three syntactic forms for circuits: *output*, *gate application*, and *dynamic lifting*.

```
Inductive Circuit (W : WType) : Set :=
| output : Pat W → Circuit W
| gate   : ∀ {W1 W2},
```

```

      Gate W1 W2 → Pat W1 → (Pat W2 → Circuit W) → Circuit W
| lift   : Pat Bit → (ℬ → Circuit W) → Circuit W.

```

An output circuit output p is just a pattern whose wire type uniquely determines the wire type of the circuit. A gate application, which we write using the notation `gate p2 ← g @p1; c`, is made up of a gate $g : \text{Gate } W_1 W_2$, an input pattern $p1 : \text{Pat } W_1$, and a *continuation* $c : \text{Pat } W_2 \rightarrow \text{Circuit } W$. The intended meaning is that $p1$ is the input to the gate g , and its output is bound to $p2$ in the continuation.

The lift operation, which we write as `lift x ← p; C`, takes as input a bit wire $p : \text{Pat Bit}$ and a function `fun x ⇒ C` that takes a bool and returns a circuit. The intended semantics is that the QRAM will transmit the value of p as a Coq boolean to the classical computer, which will then compute the remainder of the circuit. We use notations to define lifting on qubits (which is equal to a measurement followed by `lift`) and patterns of bits and qubits (see Section 5.6).

A boxed circuit is simply a function from an input pattern to a circuit:

```

Inductive Box w1 w2 : Set :=
| box : (Pat w1 → Circuit w2) → Box w1 w2.

```

The corresponding `unbox` function takes a boxed circuit and a input pattern and returns a `Circuit`:

```

Definition unbox {w1 w2} (b : Box w1 w2) (p : Pat w1) : Circuit w2 :=
  match b with box c ⇒ c p end.

```

Our higher-order abstract syntax also allows us to easily define composition, where Coq performs variables substitution for us:

```

Fixpoint compose {w1 w2} (c : Circuit w1) (f : Pat w1 → Circuit w2) :
  Circuit w2 :=
  match c with
  | output p      ⇒ f p
  | gate g p c'   ⇒ gate g p (fun p' ⇒ compose (c' p') f)
  | lift p c'     ⇒ lift p (fun bs ⇒ compose (c' bs) f)
  end.

```

5.2 Typing Qwire

The previous section should give the reader a feel for the contours of QWIRE's type system. Every pattern, gate and circuit is associated with a given wire type and Coq's typechecker ensures that all the wires types are consistent for a given circuit.

However, there's more to the story. QWIRE programs are *linearly typed*, meaning that every wire must be used exactly once. Additionally, QWIRE patterns and circuits are *extrinsically typed*, meaning that we construct circuits and then prove that they type-check using Coq lemmas. This stands in contrast to an earlier presentation of QWIRE (Rand et al., 2017), in which typing contexts were embedded inside patterns

and circuits. Our prior approach had the advantage of making it impossible to construct ill-typed circuits, but also made `QWIRE` circuits quite bulky and difficult to compute with. We can regain the safety features of that version of `QWIRE` by only exposing *typed circuits* – sigma types consisting of circuits with their typing proofs – to the user.

Typing Contexts A typing context Γ of type `Ctx` is a partial map from variables (represented concretely as natural numbers) to wire types. We implement contexts as lists of `option WTypes`. In this representation, the variable i is mapped to W if the i^{th} element in the list is `Some W`, and is undefined if the list has no i^{th} element or that element is `None`.

The *disjoint merge* (\uplus) operation ensures that the same wire cannot occur twice in one circuit. Mathematically, it is defined on two typing contexts as follows:

$$\begin{aligned} [] \uplus \Gamma_2 &= \Gamma_2 \\ \Gamma_1 \uplus [] &= \Gamma_1 \\ \text{None} \quad &:: \Gamma_1 \uplus \text{None} \quad :: \Gamma_2 = \text{None} \quad :: (\Gamma_1 \uplus \Gamma_2) \\ \text{Some } W &:: \Gamma_1 \uplus \text{None} \quad :: \Gamma_2 = \text{Some } W \quad :: (\Gamma_1 \uplus \Gamma_2) \\ \text{None} \quad &:: \Gamma_1 \uplus \text{Some } W \quad :: \Gamma_2 = \text{Some } W \quad :: (\Gamma_1 \uplus \Gamma_2) \end{aligned}$$

Since disjoint merge is a partial function, we represent it in Coq as a function on possibly invalid contexts, `OCtx = Invalid | Valid Ctx`. For convenience, most operations on contexts are lifted to work with `OCtx` values, and so the type signature of `merge` is `OCtx → OCtx → OCtx`. For convenience, we use the notation $\Gamma == \Gamma_1 \bullet \Gamma_2$ for $\Gamma = \Gamma_1 \uplus \Gamma_2 \wedge \text{is_valid } \Gamma$, where `is_valid` asserts validity.

Definition `is_valid` ($\Gamma : \text{OCtx}$) : $\mathbb{P} := \exists (\Gamma' : \text{Ctx}), \Gamma = \text{Valid } \Gamma'$.

For proofs about contexts, we also provide an inductive relation `merge_ind` and prove that `merge_ind` $\Gamma_1 \Gamma_2 \Gamma$ if and only if $\Gamma == \Gamma_1 \bullet \Gamma_2$. This inductive relation makes it easier to reason about merged typing contexts, though (unlike \uplus) we can no longer compute with it or use it for rewriting.

Typing Patterns Patterns are typed as follows:

- The unit pattern `()` is typed only by the empty context `[]`
- `qubit n` is typed by a *singleton context* consisting of $n - 1$ `Nones` followed by `Some Qubit`.
- `bit n` is typed similarly to `qubit n`
- If `p1` is typed by Γ_1 , `p2` is typed by Γ_2 , and $\Gamma_1 \uplus \Gamma_2$ is valid, then `(p1,p2)` is typed by $\Gamma_1 \uplus \Gamma_2$.

We use the notation $\Gamma \vdash p : \text{Pat}$ for “ Γ types the pattern p .”

$$\begin{array}{c}
\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \text{output } p : W} \text{TYPECIRCOUTPUT} \\
\\
\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad \Gamma; \Omega_2, \Omega \vdash C : W}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W} \text{TYPECIRCGATE} \\
\\
\frac{\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \leftarrow \text{lift } p; C : W'} \text{TYPECIRCLIFT} \\
\\
\frac{\Gamma; \Omega_1 \vdash C : W \quad \Gamma; \Omega_0, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \text{TYPECIRCLET} \\
\\
\frac{\Gamma \vdash t : \text{Box } W_1 \ W_2 \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \text{unbox } t \ p : W_2} \text{TYPECIRCUNBOX} \\
\\
\frac{\Omega \Rightarrow p : W_1 \quad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \text{box } (p : W_1) \Rightarrow C : \text{Box } W_1 \ W_2} \text{TYPEBOX}
\end{array}$$

Figure 5.1: The typing rules for \mathcal{Q} WIRE from Chapter 4. We’ve rearranged the rules to match this presentation.

Typing Qwire Circuits Once our circuits use continuations (in `gate g p1 (fun p2 => c)`), we have to include a form of continuation in the typing judgment as well. In the following definition, we use the notation $\Gamma \vdash c : \text{Circ}$ for `Types_Circuit` Γ `c`.

```

Inductive Types_Circuit : 0Ctx → ∀ {w}, Circuit w → Set :=
| types_output : ∀ {Γ w} {p : Pat w}, Γ ⊢ p :Pat → Γ ⊢ output p :Circ
| types_gate : ∀ {Γ Γ1 Γ1' w1 w2 w} {f : Pat w2 → Circuit w}
  {p1 : Pat w1} {g : Gate w1 w2},
  Γ1 ⊢ p1 :Pat →
  (∀ Γ2 Γ2' (p2 : Pat w2) {pf2 : Γ2' == Γ2 • Γ},
   Γ2 ⊢ p2 :Pat → Γ2' ⊢ f p2 :Circ) →
  ∀ {pf1 : Γ1' == Γ1 • Γ},
  Γ1' ⊢ gate g p1 f :Circ
| types_lift : ∀ {Γ1 Γ2 Γ w } {p : Pat Bit} {f : ℤ → Circuit w},
  Γ1 ⊢ p :Pat →
  (∀ b, Γ2 ⊢ f b :Circ) →
  ∀ {pf : Γ == Γ1 ∪ Γ2},
  Γ ⊢ lift p f :Circ

```

Compare this to the original presentation of the typing rules (reproduced in Fig-

ure 5.1). The rule for output is straightforward: If Γ types a pattern p , it types `output p`. For gate application, we again implement the rule from Figure 5.1, except now we have to quantify over typing contexts Γ_2 that type the continuation. We name the disjoint unions $\Gamma \uplus \Gamma_1$ and $\Gamma \uplus \Gamma_2$ as Γ_1' and Γ_2' , respectively. For lift, we limit ourselves to the Bit case, meaning that Γ_2 merely has to type the continuation whether we measured $|1\rangle$ (`true`) or $|0\rangle$ (`false`).

Note that by our construction, the typing rules for `unbox` and `let` are derived rules, as these operations are themselves derived. We say that a boxed circuit is well typed if the underlying circuit is well typed:

Definition `Typed_Box` $\{W_1 W_2 : WType\}$ $(b : Box W_1 W_2) : Set :=$
 $\forall \Gamma (p : Pat W_1), \Gamma \vdash p : Pat \rightarrow \Gamma \vdash unbox b p : Circ.$

The proof of `TYPECIRCUNBOX` is then immediate. We can likewise show that the composition of two circuits typed by disjoint contexts is itself well-typed, though the proof in this case is slightly more involved:

Lemma `compose_typing` : $\forall \Gamma \Gamma' \Gamma'' W W'$
 $(c : Circuit W) (f : Pat W \rightarrow Circuit W'),$
 $\Gamma \vdash c : Circ \rightarrow$
 $(\forall \Gamma_2 \Gamma_2' (p2 : Pat W_2) \{pf2 : \Gamma_2' == \Gamma_2 \bullet \Gamma'\},$
 $\Gamma_2 \vdash p2 : Pat \rightarrow \Gamma_2' \vdash f p2 : Circ) \rightarrow$
 $\Gamma'' == \Gamma \bullet \Gamma' \rightarrow$
 $\Gamma'' \vdash compose c f : Circ.$

This corresponds to `TYPECIRCLET` in Figure 5.1.

5.3 De Bruijn Circuits

Before we can compile circuits to density matrices, we need an intermediate representation of circuits. While HOAS circuits are easy to write and compose, they are very hard to transform into functions on density matrices. This is because they suffer from two drawbacks: They are neither *concrete* nor *compact*.

We say that a circuit is *concrete* if none of its patterns are bound within the circuit. That is, the patterns in a concrete circuit should look like `(bit 2, qubit 3)`, rather than `(p1,p2)` or simply `p`. In a well-typed boxed circuit, none of the patterns are concrete: They are all either bound at the start of the circuit or in a gate continuation. This makes it difficult to directly denote circuits: `gate CNOT (bit 2, qubit 3) c` is easy to interpret as “apply a *CNOT* to (qu)bits 2 and 3”, but what does `gate CNOT p c` mean?

We say that a circuit is *flat* if the wires it uses are numbered 0 through n for some n . Flatness is important because if I say “apply a *CNOT* to (qu)bits 2 and 3 to my quantum state”, it’s important that my quantum state should contain (qu)bits 2 and 3 and they should be in their expected positions. (That is to say that qubits 0 and 1 should exist.) HOAS circuits cannot be flat, by virtue of not being concrete,

but compiling to flat, concrete circuits poses a much harder challenge than simply compiling to concrete circuits¹.

To address this challenge, we define *de Bruijn circuits*, inspired by de Bruijn indices, where wire initialization automatically binds a wire to the first available numerical index, and discarding shifts the remaining indices downwards².

We will introduce de Bruijn (or DB) circuits, and how to construct them, by example. Here is our `bob` circuit (from our teleportation example in Section 2.2), both in higher-order abstract syntax and compiled to a (prettified) boxed `DeBruijn_Circuit` :

<pre> Definition bob := box (x,y,q) => gate (y,q) <- bit_ctrl X @ (y,q); gate (x,q) <- bit_ctrl Z @ (x,q); gate () <- discard @ x; gate () <- discard @ y; output q. </pre>	<pre> Definition db_bob := db_box (Bit ⊗ Bit ⊗ Qubit) => db_gate (bit_ctrl X) (1, 2); db_gate (bit_ctrl Z) (0, 2); db_gate discard 0; db_gate discard 0; db_output 0. </pre>
---	--

In the `bob` circuit, we begin by binding two bits and a qubit to x , y and q , respectively. We then apply a controlled X from y to q and then a controlled Z from x to q . We then discard x and y (binding the outputs to the empty pattern ()) and output the qubit q .

In the de Bruijn circuit, we begin by allocating two bits and a qubit, implicitly to 0, 1 and 2. We then apply a controlled X from wire 1 to 2 and a controlled Z from 0 to 2. We then discard bit 0. This operation shifts all the subsequent indices, so that the remaining bit is now referred to as 0 and the qubit as 1. Discarding the other bit repeats this shifting operation, leaving our qubit as 0. We then output the qubit.

We see that `discard` effectively closes the scope of a given binder. What opens a binding scope? Clearly, the `db_box` operator at the start of the circuit binds variables, but what if I wish to initialize a qubit in my circuit? In the example above, inserting a `db_gate init0` after the box would create a new qubit bound to 3. If instead we placed the `init0` between the two `discards`, it would bind that qubit to 2, since 2 was freed up by the previous `discard`. The initialization gates `init0`, `init1`, `new0` and `new1` are the only gates that bind new variables; `discard` is the only one that terminates them. (Additional gates for discarding wires will be introduced in Chapter 8.) The remaining gates can be thought of as merely functions on existing variables, though `meas` changes the type of its input variable from a qubit to a bit.

We can now describe the compilation procedure. The first step in compiling HOAS circuits to DB circuits is to concretize the variables, so that we can pattern-match on the circuit's own patterns. The `get_fresh` function takes in a wire type W and a

¹In the latter case, we would simply maintain a natural number n corresponding to the smallest wire number used so far and allocate fresh patterns with indices greater than n .

²These could reasonably be called “reverse de Bruijn circuits”, since the indices increase with successive initializations, but the analogy to de Bruijn indices is loose in either case.

context Γ , and returns a fresh pattern p and context Γ' such that Γ' types p at W :

```

Fixpoint get_fresh w ( $\Gamma$  : Ctx) : Pat w * Ctx :=
  match w with
  | One    $\Rightarrow$  (unit, [])
  | Bit    $\Rightarrow$  (bit (length  $\Gamma$ ), singleton (length  $\Gamma$ ) w)
  | Qubit  $\Rightarrow$  (qubit (length  $\Gamma$ ), singleton (length  $\Gamma$ ) w)
  | w1  $\otimes$  w2  $\Rightarrow$  let (p1,  $\Gamma_1$ ) := get_fresh w1  $\Gamma$  in
                    match  $\Gamma \uplus \Gamma_1$  with
                    | Invalid  $\Rightarrow$  (dummy_pat _, dummy_ctx)
                    | Valid  $\Gamma'$   $\Rightarrow$  let (p2,  $\Gamma_2$ ) := get_fresh w2  $\Gamma'$  in
                                         match  $\Gamma_1 \uplus \Gamma_2$  with
                                         | Invalid  $\Rightarrow$  (dummy_pat _, dummy_ctx)
                                         | Valid  $\Gamma''$   $\Rightarrow$  ((pair p1 p2),  $\Gamma''$ )
                                         end
                    end
  end
end.

```

Note that the input and output contexts of `get_fresh` will never overlap so the `Invalid` branches will never be entered. (We prove this in `get_fresh_merge_valid`.) In fact, the first $|\Gamma|$ elements of the output Γ' will always be `None`. This allows us to define `add_fresh`, which also returns a pattern and a context, except here the context is the $\Gamma \uplus \Gamma'$. This context plays the role of a *state* in constructing the circuit.

The following `hoas_to_db_box` function simply provides a fresh pattern and state for compiling the inner HOAS circuit to a DB circuit:

```

Definition hoas_to_db_box {w1 w2} (B : Box w1 w2) : DeBruijn_Box w1 w2 :=
  match B with
  | box f  $\Rightarrow$  let (p, $\Gamma$ ) := add_fresh w1 [] in
                db_box w1 (hoas_to_db  $\Gamma$  (f p))
  end.

```

We can now give the full code for `hoas_to_db`, which essentially threads this state through its computation (but contains several functions that we will need to explain):

```

Fixpoint hoas_to_db {w}  $\Gamma$  (c : Circuit w) : DeBruijn_Circuit w :=
  match c with
  | output p    $\Rightarrow$  db_output (subst_pat  $\Gamma$  p)
  | gate g p f  $\Rightarrow$  let p0 := subst_pat  $\Gamma$  p in
                    let (p', $\Gamma'$ ) := process_gate g p  $\Gamma$  in
                    db_gate g p0 (hoas_to_db  $\Gamma'$  (f p'))
  | lift p f    $\Rightarrow$  let p0 := subst_pat  $\Gamma$  p in
                    let  $\Gamma'$  := remove_pat p  $\Gamma$  in
                    db_lift p0 (fun b  $\Rightarrow$  hoas_to_db  $\Gamma'$  (f b))
  end.

```

Let us begin with the simplest function, `process_gate`, which handles the concretization part of the compilation procedure.

```

Definition process_gate {w1 w2} (g : Gate w1 w2)
  : Pat w1 → Ctx → Pat w2 * Ctx :=
  match g with
  | U _ | BNOT    ⇒ fun p st ⇒ (p,st)
  | init0 | init1 ⇒ fun _ st ⇒ add_fresh Qubit st
  | new0 | new1  ⇒ fun p st ⇒ add_fresh Bit st
  | meas          ⇒ fun p st ⇒ match p with
                        | qubit v ⇒ (bit v, change_type v Bit st)
                        end
  | discard       ⇒ fun p st ⇒ (unit, remove_pat p st)
  end.

```

This function takes in a gate, a concrete input pattern, and a state, then returns a new concrete pattern and an updated state. For unitary gates, which have the same input and output wire types, it simply returns the provided pattern and state. In effect, this says “you can continue using this pattern in the continuation”. Similarly, `process_gate meas` simply changes the given pattern from `qubit v` to `bit v` and updates the state to reflect that change. Initialization produces a fresh bit or qubit and adds it to the end of the state, and `discard` returns the pattern `()` and removes a bit from the state. To be precise, it updates `Some Bit` to `None` in the state — this will be important for what follows.

If we left out the `subst_pats` in the code above, it would convert `bob` to the following circuit³:

```

Definition bob :=
  box (bit 0, bit 1, qubit 2) ⇒
    gate (bit_ctrl X) @ (1,2);
    gate (bit_ctrl Z) @ (0,2);
    gate discard @ 0;
    gate discard @ 1;
    output 2.

```

This circuit is *concrete* but it is not yet *flat*: By the end of the circuit, there is a wire 2 but no 0 or 1. In order to flatten this circuit we need substitution contexts to map variables to their flat equivalents. Fortunately, our `Ctxs`, which served admirably in the role of state, also make good substitution contexts. To see why, note that there are two ways of giving the size of a `Ctx` that differ by whether we count `Nones`, as illustrated by the following example:

```

length [None; Some Bit; None; None; Some Qubit] = 5
size   [None; Some Bit; None; None; Some Qubit] = 2

```

We can similarly define an `index` function that ignores `Nones`. In the above example `index 1 Γ = 0` and `position 4 Γ = 1`. If `index` is given an argument that corresponds

³Note that we don’t have a type for this intermediate representation, which is simply meant to be illustrative.

to a `None` or lies outside the bounds of the list, it returns 0. To perform substitution, we simply map `index` throughout a pattern:

```
Fixpoint  $\sigma$  ( $\Gamma$  : Ctx) {w} (p : Pat w) : Pat w :=
  match p with
  | unit     $\Rightarrow$  unit
  | qubit x  $\Rightarrow$  qubit (index x  $\Gamma$ )
  | bit    x  $\Rightarrow$  bit (index x  $\Gamma$ )
  | pair p1 p2  $\Rightarrow$  pair ( $\sigma$   $\Gamma$  p1) ( $\sigma$   $\Gamma$  p2)
  end.
```

To see how this looks in practice, we return to our `bob` circuit, annotating each line with the value of `σ Γ 2` after each operation:

```
Definition bob :=
  box (bit 0, bit 1, qubit 2)  $\Rightarrow$ 
    gate (bit_ctrl X) @ (1,2);     $\sigma$  [Some Bit; Some Bit; Some Qubit] 2 = 2
    gate (bit_ctrl Z) @ (0,2);     $\sigma$  [Some Bit; Some Bit; Some Qubit] 2 = 2
    gate discard @ 0;              $\sigma$  [None      ; Some Bit; Some Qubit] 2 = 1
    gate discard @ 1;              $\sigma$  [None      ; None      ; Some Qubit] 2 = 0
  output 2.
```

Performing all of the substitutions as we go yields the de Bruijn circuit `db_bob` with which we began this section.

5.4 Matrices and Semantics

To turn our De Bruijn circuits into functions on density matrices, we need libraries for complex numbers and matrices. We begin with our complex number library, a modified version of the Coquelicot “Complex” library (Boldo et al., 2015).

5.4.1 Complex Numbers

Coquelicot defines its complex numbers as simply pairs of Coq reals.

```
Definition  $\mathbb{C}$  :=  $\mathbb{R}$  *  $\mathbb{R}$ .
```

The Coq real number library is *axiomatic*: 0, 1, +, * and other terms and operations are defined using the Coq keyword `Parameter`, which treats them as inhabitants of the given type without requiring them to be further specified. The basic properties of addition and multiplication are then given in terms of axioms, which have no proof. An injection is then established from the integers to the reals.

This representation has advantages and disadvantages. Its main disadvantage is that it is impossible to compute with Coq’s reals, since its basic operations are non-computational. For example, though we can prove that $1 + 1 = 2$, we cannot *simplify* $1 + 1$ to 2. This is a common difficulty with representations of real numbers, including

those of the Mathematical Components (Mahboubi et al., 2016) and C-CORN (Cruz-Filipe et al., 2004) libraries. On the other hand, a major advantage of extending Coq’s reals is that it gives us access to Coq’s automation techniques, including `field`, which converts field expressions into a normal form to check equality, and `fourier` which applies the Fourier-Motzkin algorithm (Fourier, 1826; Motzkin, 1936) to decide real (in)equalities. The Coq standard library also provides the powerful `lra` (linear real arithmetic) tactic which subsumes `field` and `fourier`. We define our own tactics that extend this automation to complex numbers, which we discuss in Chapter 9.

Our additions to Coquelicot are relatively modest. We show that for any complex number c , $c * \bar{c}$ is a real number, which is important for multiplying a matrix by its adjoint, along with a few related lemmas about the complex conjugate. We also prove a number of lemmas about taking square roots, which are important for reasoning about states with Hadamard matrices applied.

Since certain quantum algorithms apply *phase shift* gates, which correspond to the matrix

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

for some real number θ , we define $e^{i\theta}$ using Euler’s formula:

Definition `Cexp` ($\theta : \mathbb{R}$) : $\mathbb{C} := \cos \theta + i * \sin \theta$

We also use this to prove the special case of Euler’s identity, which we write as `Cexp PI = -1`. Note that we cannot define the general case of complex exponentiation as a function, since x^y can have multiple solutions for $x, y \in \mathbb{C}$.

Our modified version of Coquelicot’s complex number library is available in the `QWIRE` Github repository (<https://github.com/inQWIRE/QWIRE>) as `Complex.v`. This file also removes all of Coquelicot’s dependencies, including the `SSReflect` and `Mathematical Components` libraries.

5.4.2 The Matrix Library

The denotational semantics of `QWIRE` is implemented using a matrix library created specifically for this purpose. Matrices are simply functions from pairs of natural numbers to complex numbers.⁴

Definition `Matrix` ($m\ n : \mathbb{N}$) := $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{C}$.

The arguments m and n , which are the dimensions of the matrix, are not mentioned on the right hand of the equals sign, but they are used to define certain operations on matrices, such as the Kronecker product and matrix multiplication, which depend on these dimensions. They are also useful as an informal annotation that aids the programmer, in the vein of *phantom types* (Leijen and Meijer, 1999). We say a matrix is *well-formed* when it is zero-valued outside of its domain.

⁴As a Coq technicality, note that we require the `functional extensionality` axiom to prove matrix equality.

Definition `WF_Matrix {m n} (M : Matrix m n) : \mathbb{P} :=`
 `$\forall i j, i \geq m \vee j \geq n \rightarrow M i j = 0.$`

The library is designed to facilitate reasoning about and computing with matrices. Treating matrices as functions allows us to easily express otherwise complicated matrix operations. Consider the definitions of Kronecker product (\otimes) and adjoint (\dagger), where x^* is the complex conjugate of x :

Definition `kron {m n o p} (A : Matrix m n) (B : Matrix o p) :`
 `Matrix (m*n) (o*p) :=`
 `fun x y => A (x / o) (y / p) * B (x mod o) (y mod p).`
Infix `" \otimes " := kron (at level 40, left associativity).`

Definition `adjoint {m n} (A : Matrix m n) : Matrix n m :=`
 `fun x y => (A y x)*.`

Notation `"A \dagger " := (conj_transpose A) (at level 0).`

These definitions allow us to easily prove properties of matrices, like that the adjoint is involutive, by calling basic automation tactics:

Lemma `adjoint_involutive : $\forall \{m n\} (A : Matrix m n), A^{\dagger\dagger} = A.$`
Proof. `intros. mlra. Qed.`

5.4.3 Density Matrices

We aren't just interested in matrices broadly, but in the specific density matrices and unitary matrices discussed in Chapter 2. We start with some preliminary definitions. A unitary matrix is a well-formed, $n \times n$ matrix A such that $A^\dagger \times A$ is the identity matrix `'I_n`.

Definition `is_unitary {n} (A : Matrix n n) :=`
 `WF_Matrix A \wedge A † \times A = 'I_n.`

A pure state of a quantum system is one that corresponds to a unit vector $|\phi\rangle$. We can specify a well-formed unit vector as

Definition `Pure_State_Vector {n} (ϕ : Matrix n 1): \mathbb{P} :=`
 `WF_Matrix n 1 ϕ \wedge ϕ^\dagger \times ϕ = 'I_1.`

and use this definition to define pure states:

Definition `Pure_State {n} (ρ : Matrix n n) : \mathbb{P} :=`
 `$\exists \phi, \text{Pure_State_Vector } \phi \wedge \rho = \phi \times \phi^\dagger.$`

A density matrix, or mixed state, is a linear combination of pure states representing the probability of each pure state.

Inductive `Mixed_State {n} (ρ : Matrix n n) : \mathbb{P} :=`
`| Pure_S : $\forall \rho, \text{Pure_State } \rho \rightarrow \text{Mixed_State } \rho$`
`| Mix_S : $\forall (p : \mathbb{R}) \rho_1 \rho_2, 0 < p < 1 \rightarrow$`

```

Mixed_State  $\rho_1 \rightarrow$ 
Mixed_State  $\rho_2 \rightarrow$ 
Mixed_State (p .*  $\rho_1$  .+ (1-p) .*  $\rho_2$ ).

```

Note that every mixed state is also well formed, since scaling and addition preserve well-formedness.

A superoperator is a function on square matrices that takes mixed states to mixed states.

Definition Superoperator $m\ n := \text{Matrix } m\ m \rightarrow \text{Matrix } n\ n$.

Definition WF_Superoperator $m\ n$ ($f : \text{Superoperator } m\ n$) :=
 $\forall (\rho : \text{Matrix } m\ m), \text{Mixed_State } \rho \rightarrow \text{Mixed_State } (f\ \rho)$.

Any $m \times n$ matrix A can be lifted to a superoperator from n to m by multiplying an input matrix by A and its adjoint:

Definition super { $m\ n$ } ($A : \text{Matrix } m\ n$) : Superoperator $n\ m :=$
 $\text{fun } \rho \Rightarrow A \times \rho \times A^\dagger$.

Of course, not every matrix so lifted will produce a *well-formed superoperator* that preserves mixed states. We will address which matrices have this property, along with broader questions about well-formed quantum structures, in Chapter 6.

5.5 Denotation of Qwire

Wire Types, Contexts and Patterns In order to interpret circuits as superoperators over density matrices, we will also give types, contexts, gates, and patterns algebraic interpretations. For clarity we write $\llbracket - \rrbracket$ for the denotation of a variety of QWIRE objects, which we express via a Coq type class.

Class Denote source target := { denote : source \rightarrow target }.

Notation " $\llbracket x \rrbracket$ " := (denote x) (at level 10).

We interpret every wire type as the number of Bit or Qubit wires in that type, so $\llbracket \text{Qubit} \otimes (\text{One} \otimes \text{Bit}) \rrbracket = 2$. Contexts are similarly denoted by the number of Bit or Qubit wires they type.

Patterns of type Pat W are interpreted as permutation matrices of dimension $2^{\llbracket W \rrbracket} \times 2^{\llbracket W \rrbracket}$. These matrices are constructed via multiple applications of the swap matrix to the right. In the simple case, we swap two qubits via a series of adjacent swaps. For instance $\text{swap_two } 3\ 0\ 2 = (I_2 \otimes \text{swap})(\text{swap} \otimes I_2)(I_2 \otimes \text{swap})$ swaps the 0th and 2nd qubits in a 3-qubit system.

For more general permutations, we begin by translating the input pattern into a list: $(2,(0,1))$ becomes $[2,0,1]$, which then becomes $[(0,2),(1,0),(2,1)]$. This sequence of pairs is then translated into a sequence of swap matrices that ensure that qubit 2 is put in the zero position and so forth. In this way output p is interpreted as a reordering operation.

Unitaries Every gate of type `Unitary W` corresponds to a unitary matrix of dimension $2^{\llbracket W \rrbracket} \times 2^{\llbracket W \rrbracket}$. The following gates are built-in to `QWIRE`, but additional gates may be added:

$$\llbracket X \rrbracket = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \llbracket Y \rrbracket = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \llbracket Z \rrbracket = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \llbracket H \rrbracket = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \llbracket R_\phi \rrbracket = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$$

We also have `[[ctrl U]]` which is equal to the block matrix

$$\begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & \llbracket U \rrbracket \end{pmatrix}.$$

We can prove that every denoted unitary satisfies the `is_unitary` predicate defined in the previous section:

Lemma `unitary_gate_unitary` : $\forall \{W\} (u : \text{Unitary } W), \text{is_unitary } \llbracket u \rrbracket$.

Gates We can now look at denoting gates in general. We will begin in a simplified setting, where the gate is being applied to a system of the corresponding size in the correct order (for instance, `CNOT` being applied to qubits 0 and 1 of a two qubit system). In the unitary case, we simply apply `U` and its adjoint to the state, written mathematically by

$$\llbracket U \ u \rrbracket \rho = \llbracket u \rrbracket \rho \llbracket u \rrbracket^\dagger$$

or in Coq as

$$\llbracket U \ u \rrbracket \rho = \text{super } U \ \rho$$

We can give the denotation of the remaining gates in Coq, following the denotation given in Section 4.5:

```

Definition denote_gate {W1 W2} (g : Gate W1 W2) :
  Superoperator 2^[[W1]] 2^[[W2]] :=
  match g with
  | U u           => super [[u]]
  | init0 | new0 => super |0⟩
  | init1 | new1 => super |1⟩
  | meas         => Splus (super |0⟩⟨0|) (super |1⟩⟨1|)
  | discard     => Splus (super ⟨0|) (super ⟨1|)
  end.

```

where `Splus` sums over superoperators.

Gates in Context It takes some work to lift this denotation function to the context of denoting gates in a circuit, where we may have additional unused wires or non-adjacent wires input to one gate. We will begin by giving the denotation for single qubit unitaries, which we pad with identity matrices before applying them to a

quantum state. For instance, if we want to apply a unitary U to qubit 3 in a 5-qubit system ρ , this amounts to the operation

$$(I_{2^3} \otimes U \otimes I_{2^1}) \times \rho \times (I_{2^3} \otimes U \otimes I_{2^1})^\dagger,$$

which we write in Coq as

$$\text{super } ('I_{(2^3)} \otimes \llbracket U \rrbracket \otimes 'I_{(2^1)}) \rho.$$

It's similarly easy to describe the semantics of gates like measurement, which we can pad with identity matrices, or initialization, which we add to the end of the circuit:

Definition `apply_meas {n} (k : ℕ) : Superoperator (2^n) (2^n) :=`
`Splus (super (Id (2^k) ⊗ |0⟩⟨0| ⊗ Id (2^(n-k-1))))`
`(super (Id (2^k) ⊗ |1⟩⟨1| ⊗ Id (2^(n-k-1))))).`

Definition `apply_init0 {n} : Superoperator (2^n) (2^(n+1)) :=`
`super (Id (2^n) ⊗ |0⟩).`

Here, initializing a $|0\rangle$ qubit in the 2 qubit system ρ yields $(I_4 \otimes |0\rangle)\rho(I_4 \otimes \langle 0|)$.

What about unitaries with multiple controls? Here things become a bit harder, since we cannot simply pad the unitary on either side, given that the target qubits may not even be adjacent to one another or in the desired order (see, for instance, the gate in Figure 5.2). Instead, we make use of the observation⁵ that

$$\text{control } U = |1\rangle\langle 1| \otimes U + |0\rangle\langle 0| \otimes I$$

where I is the identity matrix with the same dimensions as U . This expression can easily be padded with the identity in the middle, as in

$$|1\rangle\langle 1| \otimes I_4 \otimes U + |0\rangle\langle 0| \otimes I_4 \otimes I$$

or reversed, as in

$$U \otimes |1\rangle\langle 1| + I \otimes |0\rangle\langle 0|$$

when the target is above the control.

This gives us a blueprint for applying unitary gates which may have multiple control qubits to arbitrary systems. First we need to know where we will be placing the controls and where we will be placing the single-qubit gate. We put this information into a zipper structure (see Figure 5.2), containing the controls above the unitary, those after it, and the 2×2 unitary matrix. For instance, suppose we wished to apply `ctrl1 (ctrl1 (ctrl1 Z))` to the control wires 0, 2, 7, and the target wire 4 in an eight-qubit system: We would construct the zipper (`[true; false; true; false]`, `[[Z]]`, `[false; false; true]`).

⁵This observation is due to Kesha Hietala, who proposed the algorithm that follows as an efficient alternative to the liberal use of swap gates (our prior approach).

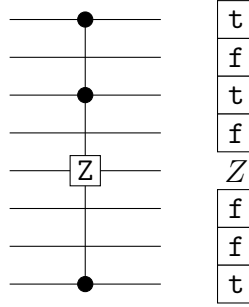


Figure 5.2: A ctrl (ctrl (ctrl Z)) gate applied to 0,2,7 and 4, and it's corresponding zipper.

We would then feed this zipper into the following (symmetric) functions:

```

Fixpoint ctrl_list_to_unitary_l (l r : list  $\mathbb{B}$ ) (u : Square 2) :
  (Square (2^(length l + length r + 1))) :=
  match l with
  | false :: l' => 'I_ 2  $\otimes$  ctrl_list_to_unitary l' r u
  | true  :: l' => |1><1|  $\otimes$  ctrl_list_to_unitary l' r u .+ |0><0|  $\otimes$  'I_ _
  | []      => ctrl_list_to_unitary_r (rev r) u
  end.

```

```

Fixpoint ctrl_list_to_unitary_r (r : list  $\mathbb{B}$ ) (u : Square 2) :
  (Square (2^(length r + 1))) :=
  match r with
  | false :: r' => ctrl_list_to_unitary_r r' u  $\otimes$  Id 2
  | true  :: r' => ctrl_list_to_unitary_r r' u  $\otimes$  |1><1| .+ 'I_ _  $\otimes$  |0><0|
  | []      => u
  end.

```

Note that 'I_ _' infers the correct dimensions for the identity matrix. Note also that the second list is reversed before it is passed to ctrl_list_to_unitary_r, allowing us to read the list from right to left.

In our example, ctrl_list_to_unitary_r [true; false; false] [[Z]] would produce the unitary matrix

$$(Z \otimes I_2 \otimes I_2 \otimes |1\rangle\langle 1|) + (I_8 \otimes |0\rangle\langle 0|).$$

Applying ctrl_list_to_unitary to ([true; false; true; false], Z, [false; false; true]) would then produce

$$|1\rangle\langle 1| \otimes (I_2 \otimes (|1\rangle\langle 1| \otimes (I_2 \otimes (Z \otimes I_2 \otimes I_2 \otimes |1\rangle\langle 1| + I_8 \otimes |0\rangle\langle 0|))) + |0\rangle\langle 0| \otimes I_{32})) + |0\rangle\langle 0| \otimes I_{128}$$

which we could apply to our entire $2^8 \times 2^8$ density matrix.

We could then apply this unitary to the entirety of the state.

Circuits We are now in a position to define the general denotation of the De Bruijn circuits from Section 5.3:

```

Fixpoint denote_db_circuit {w} padding input (c : DeBruijn_Circuit w) :
  Superoperator (2^(padding+input)) (2^(padding+[[w]])) :=
  match c with
  | db_output p                => super (pad (padding+input) [[p]])
  | db_gate w1 w2 g p c' =>
    let input' := (input + [[w2]] - [[w1]]) in
    compose_super (denote_db_circuit padding input' c')
      (apply_gate g (pat_to_list p))
  | db_lift p c' =>
    let k := get_var p in
    Splus
      (compose_super
        (denote_db_circuit padding (input-1) (c' false))
        (super ('I_(2^k) ⊗ ⟨0| ⊗ 'I_(2^(input-k-1)))))
      (compose_super
        (denote_db_circuit padding (input-1) (c' true))
        (super ('I_(2^k) ⊗ ⟨1| ⊗ 'I_(2^(input-k-1)))))
  end.

```

Here `input` is the number of input qubits and `padding` allows us to pad the denotation with identities, which is useful in a number of lemmas.

The denotation of output `p` is simply the denotation of `p`: a reordering of the qubits via `SWAP` gates. Applying a gate consists of composing the `apply_gate` operation above with the denotation of the rest of the circuit. Finally, `lift` applies the discard operations `super ('I_(2^k) ⊗ ⟨0| ⊗ 'I_(2^(input-k-1))`) and `super ('I_(2^k) ⊗ ⟨1| ⊗ 'I_(2^(input-k-1))`), takes the denotation of the rest of the circuit, and sums the results.

This gives us a superoperator on density matrices and allows us to verify properties of our circuits.

5.6 Functional Notations

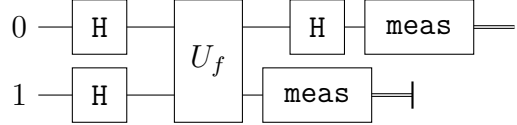
Given that `QWIRE` is embedded in Coq's Gallina functional programming language, it may be awkward for users to write quantum programs in an imperative style. We address this issue by providing two similar syntaxes for `QWIRE` programs: The *imperative syntax* is meant to represent `QWIRE` normal forms and the *functional syntax* is meant to be more usable while normalizing to the imperative form. Both syntaxes are implemented using Coq's notations feature and the second generally supersedes the first.

Imperative Syntax To get a sense for the imperative `QWIRE` syntax, we will present an implementation of Deutsch’s algorithm⁶. Note that throughout this thesis we elide the underscores that serve to differentiate our notations from our constructors and Coq’s built-in functions.

```

Definition deutsch Uf :
  Box One Bit :=
  box () ⇒
    gate x    ← init0  @ ();
    gate x    ← H      @ x;
    gate y    ← init1  @ ();
    gate y    ← H      @ y;
    let (x,y) ← unbox Uf (x,,y);
    gate y    ← meas   @ y;
    gate ()   ← discard @ y;
    gate x    ← H      @ x;
    gate x    ← meas   @ x;
  output x.

```



Note that Deutsch’s algorithm consists mostly of applying basic gates to single qubit wires, along with one unboxing of the input circuit U_f . In the simplest case, our `gate` notation simply hides some higher order abstract syntax:

Notation "'gate' p2 ← g @ p ; c2" := (gate g p (fun p2 ⇒ c2))

We also have special cases of the `gate` notation, firstly for gates with the unit output,

Notation "'gate' () ← g @ p ; c2" := (gate g p (fun _ ⇒ c2))

and secondly for gates with multiple outputs

Notation "'gate' (p1 , p2) ← g @ p ; c2" :=
 (gate g p (fun x ⇒ let (p1,p2) := wproj x in c2))

The `wproj` function here matches on a pattern of type $\text{Pat } (W_1 \otimes W_2)$ and returns its two constituent sub-patterns as `p1` and `p2`. Coq’s limited capacity for recursive notations doesn’t allow us to extend this notation to have arbitrary patterns on the left (since `(p1,p2)` cannot be made independent of the rest of the notation), instead we provide additional notations for patterns of three and four sub-patterns.

Similarly to `gate`, we write `box p ⇒ C` for `box (fun p ⇒ C)` and `let p ← c1 ; c2` for `compose c1 (fun p ⇒ c2)`. These notations also have variants for `()` and pairs, implemented in the same manner as `gate`. The perceptive reader will notice that when we unbox U_f we provide the pattern `(p1,,p2)`. This notation is defined using Coq’s recursive notations⁷ as

⁶We will leave the explanation of this algorithm for Section 7.1.3, in which we verify its correctness. For this chapter, the reader only needs to know the circuit we are implementing.

⁷The Coq reference manual calls these *recursive patterns* (Coq Development Team, 2018). We try to avoid using “patterns” to refer to anything but wire patterns, hence “recursive notations”.

Notation "`(x ,, y ,, .. ,, z)`" := `(pair .. (pair x y) .. z)`

and can be nested. We use the (admittedly ugly) double comma notation in order to save the single comma notation for `QWIRE`'s functional syntax.

We also define a `lift_wire` function that checks if a wire is a `Bit` or `Qubit` and either directly calls `lift` on it or else measures and then lifts it. This is built into our notation for `lift`, along with variants that allow the lift notation to be applied on patterns of wires (implicitly measuring and lifting them one at a time).

Functional Syntax We can now introduce our "functional" syntax for quantum circuit, again via the example of Deutsch's algorithm:

```
Definition deutsch (U_f : Square_Box (Qubit ⊗ Qubit)) : Box One Bit :=
  box () =>
    let x      ← H $ init_0 $ ();
    let y      ← H $ init_1 $ ();
    let (x,y) ← U_f $ (x,y);
    let ()     ← discard $ meas $ y;
    meas $ _H $ x.
```

The reader will notice a few changes:

1. All explicit uses of `gate` applications have been replaced by `let`.
2. Using a boxed circuit `U_f` looks no different than applying a basic gate like `H` or `meas`.
3. We can chain circuits: `H $ init_0 $ ()` applies `init_0` to `()` and then immediately applies `H` to the initialized qubit.
4. There is no explicit call to `output`.

These work through two simple coercions: We use `output` to coerce patterns to circuits and `boxed_gate` to coerce gates to boxed circuits. We then define `$` as a notation for the following function:

```
Definition apply_box {w1 w2} (b : Box w1 w2) (c : Circuit w1) :
  Circuit w2 := let x ← c; unbox b x.
```

When we write `init_0 $ ()`, then, we are really calling `apply_box` with the arguments `boxed_gate init_0` and `output ()`. This function returns a circuit producing a qubit, which can then be provided as the second argument to `apply_box (boxed_gate H)`, producing a superimposed qubit.

How then do we interpret `U_f $ (x,y)`? It is natural to assume that this is simply notation for `apply_box U_f (output (pair x y))`. However, things are slightly more complicated than that. We would like to be able to write `let (x,y) ← (H,H) $ (x,y)` or even `let (x,y) ← (H $ x, y)`. This requires `(_,_)` to take a pair of circuits with outputs `W1` and `W2` and return a `Circuit W1 ⊗ W2`. This notation calls the function `pair_circ`

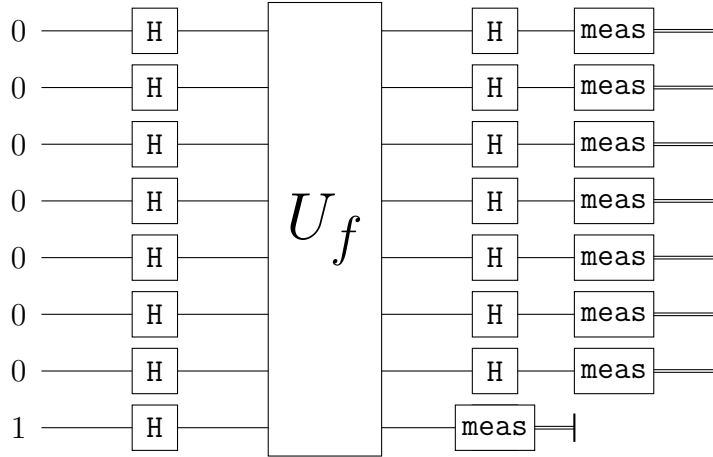


Figure 5.3: The Deutsch-Jozsa algorithm on 8 qubits (including the target).

which does exactly that. This notation is also defined as a recursive notation, allowing us to write (H,X,Z,q) and to nest circuit pairs inside other circuit pairs.

All of the functions we use here are fairly lightweight. Still, it is impossible to write a circuit in normal form using these notations (and we would like our normal forms to be readable when stepping through proofs). Hence, we maintain the imperative syntax for representing circuits, and the second presentation of `deutsch` efficiently reduces to the first.

Qwire in Action Just to give the reader a better sense for the flavor of `QWIRE`, we will conclude with two more interesting programs.

The first is the Deutsch-Jozsa algorithm (Deutsch and Jozsa, 1992), a generalization of Deutsch’s algorithm to an arbitrary number of qubits. Our implementation uses the notations `g #n` for `inParMany g n` (that is, apply n copies of g in parallel) and `()` for a pattern containing only units:

```

Definition Deutsch_Jozsa (n : ℕ)
  (U_f : Box ((n ⊗ Qubit) ⊗ Qubit) ((n ⊗ Qubit) ⊗ Qubit)) :
  Box One (n ⊗ Bit) :=
  box () =>
    let qs    ← H #n $ init_0 #n $ ();
    let q     ← H $ init_1 $ ();
    let (qs,q) ← U_f $ (qs,q);
    let ()    ← discard $ meas $ q;
    meas #n $ _H #n $ qs.

```

We also provide a coin tossing program that uses dynamic lifting (Figure 5.4). First, we will define a simple coin toss.

```

Definition coin_flip : Box One Bit :=

```

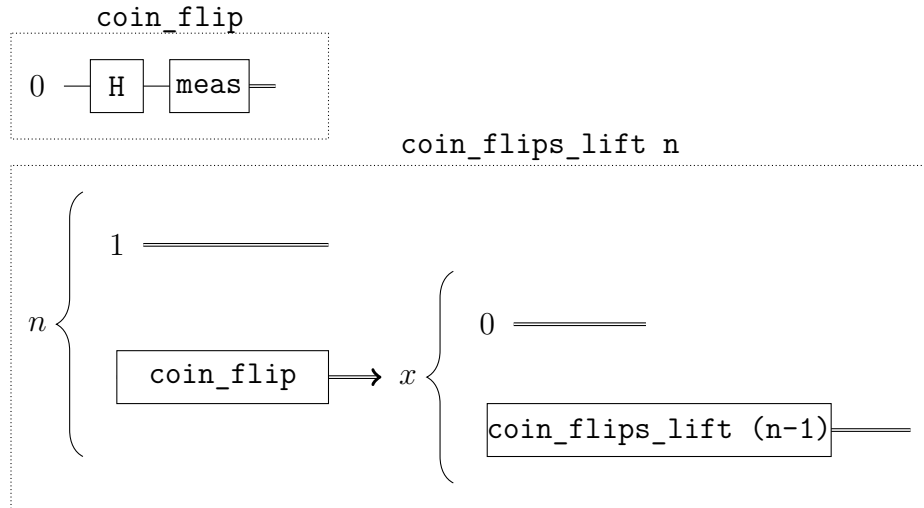


Figure 5.4: A basic coin flip circuit and a multiple-coin flipping circuit using dynamic lifting. The braces denote branching on a classical term (nats or bools).

```
box_ () => meas $ _H $ init_0 $ () .
```

Now we define a program that tosses up to n coins, stopping and returning 0 if a tails is shown, and returning 1 if all the tosses come up heads:

```
Fixpoint coin_flips_lift (n : ℕ) : Box One Bit :=
  box_ () =>
  match n with
  | 0 => new_1 $ ()
  | S n' => lift_ x <- coin_flip $ ();
           if x then coin_flips_lift n' $ ()
           else new_0 $ ()
  end.
```

This concludes our demonstration of \mathcal{Q} WIRE circuits and our chapter on \mathcal{Q} WIRE in practice.

Chapter 6

Verifying Qwire

Before we begin verifying quantum circuits in QWIRE, it's important to verify QWIRE itself. What do we mean by “verifying QWIRE”? QWIRE has notions like unitaries, density matrices, mixed states, superoperators, and even matrices themselves that aren't enforced at the type level. One can write the following matrix definition in Coq:

```
Definition bad_matrix : Matrix 4 4 := fun x y => 6.
```

This clearly doesn't correspond to a valid 4×4 matrix: It has infinitely many non-zero elements. Similarly, `Density n` is just a notation for an $n \times n$ matrix, and a `Superoperator`, which should be a mapping between proper density matrices, is simply a function from some `Density m` to `Density n`. Instead, the properties we care about are enforced by *extrinsic predicates* like `WF_Matrix`, `WF_Unitary`, `Pure_State`, `Mixed_State`, and `WF_Superoperator` that guarantee that our terms have the desired properties. In the following section, we will discuss each of these properties and how we show that our matrices possess them and our functions preserve them.

We are also concerned with broader questions about the correctness of QWIRE. One such question is whether circuit composition corresponds to function composition, as it should. We will deal with these questions in the following section.

6.1 Predicates and Preservation

6.1.1 Well-Formed Matrices

As we noted in Chapter 5, QWIRE matrices are simply functions from pairs of natural numbers (corresponding to row-column coordinates) to complex numbers. We say that such a matrix is *well-formed* when it is 0 outside the specified bounds. Recall that we capture this in the following predicate:

```
Definition WF_Matrix (m n: ℕ) (A : Matrix m n) : ℙ :=  
  ∀ x y, x ≥ m ∨ y ≥ n → A x y = 0.
```

Beyond proving that the matrices we frequently use, such as unitary operators and qubit matrices, are well-formed, we want to show that the various operations we perform on matrices preserve well-formedness. Our matrix library proves this property about all of our functions from matrices to matrices, including $+$, $*$, \times , \otimes , † , and \dagger .

It is interesting to note which theorems about matrices require well-formedness and which do not. This tells us whether these are theorems about functions that just happen to apply to matrices as a special case, or whether they are specific to matrices. Naturally, the left and right additive identities on matrices apply to all functions: If we lift $+$ to operate on functions, then $\forall f, (\lambda x.0)+f = f$. By similar reasoning, matrix addition is commutative and distributive even for ill-formed matrices. The transpose and adjoint operators can be thought of as simply swapping their arguments (and negating the complex component in one case) and thus are trivially involutive.

Operations like the trace, matrix multiplication, and Kronecker product are more difficult to analyze, given that they *use* the matrix's dimensions in their definitions. Still, it is interesting to note that, of the core properties of matrix arithmetic, only three required well-formedness in their proofs:

Lemma `Mmult_1_l`: $\forall m\ n\ (A : \text{Matrix } m\ n), \text{WF_Matrix } m\ n\ A \rightarrow 'I_m \times A = A$.

Lemma `Mmult_1_r`: $\forall m\ n\ (A : \text{Matrix } m\ n), \text{WF_Matrix } m\ n\ A \rightarrow A \times 'I_n = A$.

Lemma `Kron_1_l` : $\forall m\ n\ (A : \text{Matrix } m\ n), \text{WF_Matrix } m\ n\ A \rightarrow 'I_1 \otimes A = A$.

`Kron_1_r`, by contrast, simply states that $\forall m\ n\ (A : \text{Matrix } m\ n), A \otimes 'I_1 = A$. This is because the dimensions of the right-side matrix appear in the definition of the Kronecker product, so $(A \otimes 'I_1) x\ y$ becomes $A (x/1) (y/1) * 'I_1 (x \bmod 1) (y \bmod 1)$. Since $x \bmod 1$ is always 0 and $'I_1\ 0\ 0 = 1$, this easily simplifies to $A\ x\ y$.

Obviously, well-formedness is important for all of our unitary operators and density matrices. Therefore, our predicates `WF_Unitary U` and `Mixed_State ρ` themselves assert that U and ρ are well-formed matrices.

6.1.2 Unitarity

For every gate introduced in Chapter 5, we have a corresponding unitary matrix. Our `WF_Unitary` predicate is quite simple:

Definition `WF_Unitary {n: \mathbb{N} } (U : Matrix n n): $\mathbb{P} :=$`
`WF_Matrix n n U \wedge U † \times U = Id n.`

We can easily prove that every concrete unitary operator satisfies this predicate and that the control and adjoint operations preserve it. For most of our operators, with the exception of the phase shift by arbitrary real numbers, we can further prove that they are their own adjoints and, hence, inverses.

We can take things a step further. Not only do we want to know that our basic unitary matrices are in fact unitary, but we also want to know that when we apply a unitary gate to a large quantum state, this transformation is unitary. In the case of applying a single-qubit unitary U , this amounts to verifying that $I_m \otimes U \otimes I_n$ is a unitary matrix for any m and n . In the case of controlled unitaries, this involves proving

that the rather involved procedure in Section 5.5 produces unitary matrices. We have verified this procedure, which will allow us to show that our circuits correspond to valid superoperators in the next section.

6.1.3 Pure and Mixed States

As discussed in our introduction to quantum computing (Chapter 2), we have two types of quantum states: *pure states* and *mixed states*. Since mixed states are simply distributions over pure states, it makes sense to begin with the definition of pure states.

Formally, a pure state in density matrix form is an $n \times n$ matrix ρ satisfying the following properties (Hall (2013), Section 19.3):

1. It is self-adjoint (or *Hermitian*),
2. It is positive semidefinite.
3. $\text{trace}(\rho) = 1$.
4. $\rho^2 = \rho$.

The fourth item distinguishes pure states from mixed states.

The first, third, and fourth properties were covered in the introduction and are fairly easy to check. The fourth property will hold whenever $\text{trace}(\rho^2) = 1$, which simplifies checking it even further.

Positive semidefiniteness gives us some trouble. We call a Hermitian matrix ρ *positive semidefinite* if, for every appropriately sized vector v , $v^\dagger \rho v \geq 0$, treating $v^\dagger \rho v$ as a scalar. This is a difficult property to check since it quantifies over all vectors v . There exist a number of equivalent formulations, the most popular being that the eigenvalues of the matrix are all positive, but none are easy to check.

Instead, we return to our definition of pure states from Chapter 2, in which we noted that a pure state $|\phi\rangle$ in vector form becomes $|\phi\rangle\langle\phi|$ in density matrix form. In Coq:

```

Definition Pure_State {n} (ρ : Density n) : ℙ :=
  ∃ (ϕ : Matrix n 1), Pure_State_Vector ϕ ∧ ρ = ϕ × ϕ†.

```

What is a pure state as a vector? It is simply any complex vector $|\phi\rangle$ whose inner product is 1. Equivalently, $\langle\phi|\phi\rangle = I_1$. In our formalization, we add that the matrix must be well-formed:

```

Definition Pure_State_Vector {n} (ϕ : Matrix n 1): ℙ :=
  WF_Matrix n 1 ϕ ∧ ϕ† × ϕ = 'I_1.

```

Note that it is not trivial to prove that an arbitrary matrix is a `Pure_State` as we have defined it: It requires us to provide the corresponding pure state vector as a

witness. However, it has proven far more usable in practice than any of the definitions given above.

A mixed state is generally defined as a sum $\sum_i p_i \rho_i$, where each ρ_i is a pure state, every p_i is in $[0, 1]$, and $\sum_i p_i = 1$. We express this using the following inductive predicate:

```

Inductive Mixed_State {n} : (Matrix n n) → ℙ :=
| Pure_S : ∀ ρ, Pure_State ρ → Mixed_State ρ
| Mix_S : ∀ (p : ℝ) ρ₁ ρ₂, 0 < p < 1 → Mixed_State ρ₁ → Mixed_State ρ₂ →
Mixed_State (p * ρ₁ + (1-p) * ρ₂).

```

Note that this definition makes mixed states into a superset of pure states, as did our use of square brackets in the equation above. This is a fairly common practice and useful for our purposes: We will often care that a matrix corresponds to a valid quantum state, but we will almost never care that it is *impure*. If we wished to express that a state was mixed in the strict sense, we could simply write `Mixed_State ρ ∧ ¬ Pure_State ρ`.

We separately prove important properties of mixed states: for instance, that they all correspond to well-formed matrices and have traces that are equal to one. Most importantly, we show that they are preserved by unitary transformations and other operations. This brings us to the notion of a *superoperator*.

6.1.4 Superoperator Correctness

A *superoperator* is a function that takes mixed states to mixed states. Since our type of superoperator is simply a function from matrices to matrices, we introduce a predicate that says a superoperator truly preserves mixed states:

```

Definition WF_Superoperator {m n} (f : Superoperator m n) :=
(∀ ρ, Mixed_State ρ → Mixed_State (f ρ)).

```

One of the most relevant results for quantum computing is that every unitary operator preserves mixed states. We first show that this is true for pure states, and then we lift it to mixed states. We get the following result:

```

Lemma WF_Superoperator_unitary : ∀ {n} (U : Matrix n n),
WF_Unitary U → WF_Superoperator (super U).

```

where `super U ρ` is $U\rho U^\dagger$.

As we have noted in our section on the `WF_Unitary` predicate, we have shown that our denotation of unitary gates corresponds to a valid unitary transformation even when applied to a multiple-qubit system. Combined with the lemma above, this gets us much of the way towards verifying that our circuits are valid superoperators. First, however, we will present a simpler lemma: Applying a gate to the appropriate number of qubits is a superoperator:

```

Lemma denote_gate_correct : ∀ {W₁} {W₂} (g : Gate W₁ W₂),
WF_Superoperator (denote_gate true g).

```

This lemma deals only with the application of gates to the correct number of qubits, outside of the context of a larger circuit. However, it contains most of the reasoning about the different kinds of gates relevant to our main theorem of this section. In particular, it uses our `WF_Superoperator_unitary` lemma and proves similar results about measurement, initialization, and discard: Each operation preserves mixed states. However, it is limited by its assumption that gates are being applied to quantum states of exactly the same arity: for instance, measurement to a single qubit or *CNOT* to two qubits. Hence, we prove the following theorem, which cannot use `denote_gate_correct` directly but generalizes its result to larger quantum states.

Theorem 9. *Every well-typed static circuit corresponds to a valid superoperator.*

Theorem `denote_static_circuit_correct` : $\forall W (\Gamma_0 \Gamma : \text{Ctx}) (c : \text{Circuit } W),$
`Static_Circuit c` \rightarrow
 $\Gamma \vdash c : \text{Circ}$ \rightarrow
`WF_Superoperator` (`denote_circuit c` $\Gamma_0 \Gamma$).

A *static circuit* is any circuit that doesn’t do dynamic lifting; that is, it normalizes to a sequence of gate applications followed by an output. To understand why we limit this result to static circuits, recall the semantics of a circuit with lifting. Dynamic lifting branches on the value of some bit: It breaks our distribution over pure states into two sub-distributions over pure states, and then it applies distinct circuits to each sub-distribution. Our `WF_Superoperator` predicate only says that a function takes full distributions to full distributions over quantum states. By contrast, to reason about sub-distributions, we would require a predicate that asserts that our function takes sub-distributions of weight w to sub-distributions of weight w (again, over pure states). We could then prove that this property holds of arbitrary well-typed circuits, from which we could immediately derive that well-typed circuits correspond to valid Superoperators.

Aside from making us reason about arbitrary quantum states, proving `denote_static_circuit_correct` requires us to connect our `Types_Circuit` predicate to narrower restrictions on circuits. For instance, our correctness lemma for applying a unitary gate to an n -qubit quantum state requires that the specified wire indices should be less than n , which follows from well-typedness. The lemma does not, however, demand that all the indices be disjoint, as the linear type system does. Instead, if we say “apply `ctrl Z` to qubits 1 and 1”, the denotation function will treat this as simply, “apply `Z` to qubit 1”. We have similar requirements for applying initialization, measurement, or discard gates. The fact that ill-typed circuits can produce valid superoperators doesn’t bother us, however. We only concern ourselves with the denotation of well-typed circuits, and these all correspond to valid superoperators.

The measurement and initialization cases of `denote_static_circuit_correct` rely upon the *operator sum decomposition theorem* (Kitaev et al., 2002, Section 11.1) which says that

$$f(\rho) = \sum_m A_m \rho A_m^\dagger$$

is a valid superoperator if and only if

$$\sum_m A_m^\dagger A_m = I.$$

We have axiomatized this theorem in the Coq development due to the difficulty of connecting our definition of mixed states to the mathematical description given in the previous subsection, and thereby to the definitions used in proving the operator sum decomposition theorem. In principle, we could use this theorem to define *mixed states* but we feel that this definition would be less intuitive. Note that we can immediately derive that applying unitaries preserves mixed states from this axiom, however we prefer to prove that (and the initialization case) directly, without recourse to an axiom. We summarize the axioms used in the *QWIRE* development in Chapter 10.

6.2 Towards Compositionality

An important next step for *QWIRE* is proving that the composition lemma holds: The composition of two circuits $\llbracket p \leftarrow c; c' \rrbracket$ should be equal to $\llbracket c' \ p \rrbracket \circ \llbracket c \rrbracket$. We give a full statement of `denote_compose` below:

```
Fact denote_compose : ∀ W (c : Circuit W) (Γ : Ctx),
  Γ ⊢ c :Circ →
  ∀ W' (f : Pat W → Circuit W') (Γ₀ Γ₁ Γ₁' Γ₀1 : Ctx),
  (∀ Γ₂ Γ₂' (p2 : Pat w2) {pf2 : Γ₂' == Γ₂ • Γ},
    Γ₂ ⊢ p2 :Pat → Γ₂' ⊢ f p2 :Circ) →
  Γ₁' == Γ₁ • Γ →
  Γ₀1 == Γ₀ • Γ₁ →
  denote_circuit (compose c f) Γ₀ Γ₁' =
  compose_super
  (denote_circuit (f (add_fresh_pat w Γ₁)) Γ₀ (add_fresh_state W Γ₁))
  (denote_circuit c (Γ₀1) Γ).
```

Unfortunately, we do not yet have a formal proof of `denote_compose`. There are two reasons for this absence: Firstly, the compilation process to de Bruijn circuits is pretty complicated and it can be quite difficult to reason about the results of compilation in a general setting. A more foundational concern relates to our higher-order abstract syntax: Currently, it is possible to write a circuit that branches on the underlying representation of a variable, producing one circuit if `w` is represented by 0 and another if `w` is any other number. This makes it difficult to reason parametrically about programs in general and is discussed in depth by Despeyroux et al. (1995). One solution, following the Hybrid system (Felty and Momigliano, 2012), is to introduce a parametricity predicate like their `abstr`. (This system was recently used by Mahmoud and Felty (2018) to encode Proto-Quipper (Ross, 2015) and formalize its metatheory.) Another involves making the representation of variables themselves parametric, thereby preventing the programmer from matching on them.

If we assume the `denote_compose` lemma, we get a number of desirable corollaries, including a characterization of our circuit sequencing function:

Lemma `inSeq_correct` : $\forall W_1 W_2 W_2 (c' : \text{Box } W_2 W_2) (c : \text{Box } W_1 W_2),$
`Typed_Box c` \rightarrow `Typed_Box c'` \rightarrow
`denote_box (inSeq c c')` =
`compose_super (denote_box g) (denote_box f).`

Mathematically, for boxed circuits c and c' , $\llbracket c; c' \rrbracket = \llbracket c' \rrbracket \circ \llbracket c \rrbracket$. A similar (sketched but unproven) lemma for parallel composition says that $\llbracket c_1 \parallel c_2 \rrbracket (\rho_1 \otimes \rho_2) = (\llbracket c_1 \rrbracket \rho_1) \otimes (\llbracket c_2 \rrbracket \rho_2)$, provided that the dimensions line up:

Fact `inPar_correct` : $\forall W_1 W_1' W_2 W_2' (f : \text{Box } W_1 W_1') (g : \text{Box } W_2 W_2') (\text{safe} : \mathbb{B})$
 $(\rho_1 : \text{Square } \llbracket (2^{W_1}) \rrbracket) (\rho_2 : \text{Square } \llbracket (2^{W_2}) \rrbracket),$
`Typed_Box f` \rightarrow `Typed_Box g` \rightarrow
`WF_Matrix` $\llbracket (2^{W_1}) \rrbracket \llbracket (2^{W_1}) \rrbracket \rho_1 \rightarrow$
`WF_Matrix` $\llbracket (2^{W_2}) \rrbracket \llbracket (2^{W_2}) \rrbracket \rho_2 \rightarrow$
`denote_box safe (inPar f g) (\rho_1 \otimes \rho_2) =`
`(denote_box safe f \rho_1 \otimes denote_box true g \rho_2).`

Note that this statement is narrower than we would like it to be, in that it requires the input matrix to be the product of two non-entangled states. Really, we would like to write something of the form $\llbracket c_1 \parallel c_2 \rrbracket = \llbracket c_1 \rrbracket \otimes \llbracket c_2 \rrbracket$, for a tensor product that corresponds to the tensor product on unitary matrices. However, our representation of superoperators as simple functions from matrices to matrices limits us from introducing such an operator. This suggests implementing alternative representations of superoperators, such as the Choi or Stinespring representations (Watrous, 2018, Section 2.2), that have additional structure that we could exploit. These could also aid us in proving the operator-sum decomposition theorem of the previous section.

6.3 Future Work on Qwire’s Metatheory

We have additional ambitions for QWIRE’s metatheory. One core part of that involves typing our de Bruijn circuits. We have defined a notion of typed `DeBruijn_Circuits` in the QWIRE development. However, we haven’t shown that every well-typed HOAS circuit compiles to a well-typed de Bruijn (DB) circuit, nor have we shown that well-typed DB circuits correspond to valid superoperators. Instead, we leapfrogged typed de Bruijn circuits in the previous section by directly showing that well-typed HOAS circuits compile to valid superoperators.

If we can already prove that HOAS circuits correspond to valid superoperators, why do we want a notion of well-typed de Bruijn circuits? Mostly because DB circuits closely resemble the “quantum instruction set” programs written in languages like QUIL (Smith et al., 2016) and OpenQASM (Cross et al., 2017). As we will discuss in Section 11.2, Dong-Ho Lee’s existing compiler from QWIRE to OpenQASM goes

directly through our de Bruijn circuits. We would also like to convert OpenQASM and QUIL programs to QWIRE programs, allowing us to typecheck and verify circuits generated by a variety of quantum programming languages. Translating these programs to de Bruijn circuits would substantially simplify this process. However, this requires us to have a procedure for typechecking de Bruijn circuits and proofs that well-typed de Bruijn circuits correspond to valid superoperators. We leave this for future work.

Chapter 7

Verifying Quantum Programs

We can now move on to the other goal of our thesis: formal verification of quantum programs. This verification can take a number of forms, from low-level checks that a given circuit produces the right output matrix to high level specifications of program properties, generally as functions from matrices to matrices. In this section, we will begin with the simplest forms of verification and move on to more complicated forms.

7.1 Verifying Matrices

7.1.1 Tossing Coins

From a conceptual standpoint, the easiest type of circuit to verify is a *closed circuit*, a circuit with no input. In this case, we can fully specify the correctness of the circuit in terms of the density matrix corresponding to its output.

For example, consider a simple coin flip circuit:

Definition `coin_flip` : Box One Bit := $|0\rangle$ — H — meas —

`box () => meas $ H $ init_0 $ ()`.

This circuit should output $|1\rangle$ (corresponding to heads) with one-half probability and otherwise return $|0\rangle$. We can encode this property in the density matrix

$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

where the one-half in the top left corresponds to the probability of measuring $|0\rangle$.

The proof of this property is quite simple, consisting of simply unfolding definitions and computing, using our `matrix_denote`, `Msimp1` and `solve_matrix` tactics from Chapter 9.

7.1.2 Teleport

On occasion we can write similar proofs about open circuits: for instance, the teleport example from Chapter 2.

Here is the code for teleport

```

Definition teleport :=
  box q =>
    let (a,b) ← bell100 $ ();
    let (x,y) ← alice $ (q,a) ;
    bob $ (x,y,b).

```

and the lemma we desire to prove about it:

```

Lemma teleport_eq : teleport ≡ id_circ.

```

This says that the denotation of `teleport` is identical to the denotation of our identity circuit.

We could also write this out more explicitly as

```

Lemma teleport_eq': ∀ (ρ : Density 2),
  WF_Matrix ρ → [[teleport]] ρ = ρ.

```

That is, for any input matrix ρ corresponding to a one-qubit quantum state, teleporting ρ returns ρ .

Given that teleport takes an input qubit, it might seem difficult to use the technique above. In the one-qubit case, at least, this happens not to be true. We can rewrite our input matrix as simply

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

and treat this as a concrete matrix in our computation.

In practice, the only difficulty in proving `teleport_eq` lies in the slowness of our matrix multiplier and the fact that we are left to prove $2 * (2 * (\frac{1}{\sqrt{2}} * (\frac{1}{\sqrt{2}} * \rho_{x,y}) * \frac{1}{2})) = \rho_{x,y}$ (for each possible $x, y \in \{0, 1\}$) after our solver has run. The first emphasizes the difficulty of multiplying abstract matrices, though this could be helped by faster matrix multiplication algorithms. The second obstacle is due to the limitations of our automation techniques in dealing with square roots, though these solvers could potentially be strengthened. In any case, the proof is easily completed manually.

We can also verify the correctness of a teleportation circuit using dynamic lifting, which more accurately models the intended protocol. Here Bob receives two boolean values, which he uses to decide how to modify his qubit

```

Definition bob_distant (u v : ℤ) : Box Qubit Qubit :=
  box b =>
    let b ← (if v then X else id_circ) $ b;
    let b ← (if u then Z else id_circ) $ b;
    output b.

```

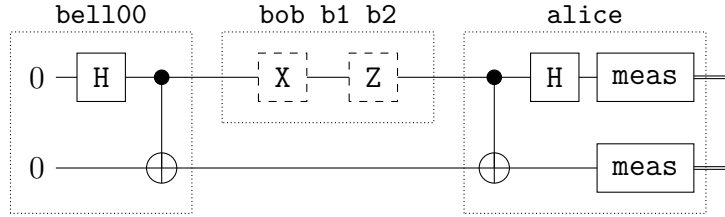


Figure 7.1: The superdense coding protocol with boolean inputs

and teleport provides those booleans via lifting:

```

Definition teleport_distant : Box Qubit Qubit :=
  box q =>
    let (a,b) ← bell100 $ ();
    let (x,y) ← alice $ (q,a) ;
    lift (u,v) ← (x,y) ;
    bob_distant u v $ b.

```

Gratifyingly, the statement and proof of `teleport_distant` are identical to those for `teleport`.

Lemma `teleport_distant_eq` : `teleport_distant` \equiv `id_circ`.

The *superdense coding* algorithm can be thought of as teleportation in reverse: Here Bob uses a qubit, along with an entangled pair, to transmit two bits of information to Alice. (The `bob` and `alice` circuits are identical to those in the teleportation example.) We show the version with boolean inputs (corresponding to `teleport_distant`) here:

```

Definition superdense_distant (b1 b2 :  $\mathbb{B}$ ) : Box One (Bit  $\otimes$  Bit) :=
  box_ () =>
    let_ (a,b) ← bell100 $ ();
    let_ q ← bob_distant b1 b2 $ b;
    alice $ (q,a).

```

The proof of superdense coding is easier than our teleportation proofs since the math is simpler, allowing us to complete the proof using our automation tactics:

Lemma `superdense_distant_eq` : \forall `b1 b2`,
`[[superdense_distant b1 b2]] I1 = bools_to_matrix [b1; b2]`.

Proof.

```

intros b1 b2.
specialize (WF_bools_to_matrix ([b1;b2])) as WF.
destruct b1, b2; matrix_denote; Msimpl; solve_matrix.

```

Qed.

Here `bools_to_matrix` converts the boolean inputs into their corresponding density matrix.

7.1.3 Deutsch’s Algorithm

Next, we verify an implementation of a classic algorithm from the the quantum computing literature, Deutsch’s algorithm (Deutsch, 1985; Cleve et al., 1998). Deutsch’s Problem presents the programmer with a function $f : \{0, 1\} \rightarrow \{0, 1\}$ and asks her to determine whether the function is constant or not. In the classical case, this obviously requires that we query the function on both 0 and 1. Deutsch’s algorithm, as modified by Cleve *et al.*, solves this on a quantum computer by using a single query. First, however, we must guarantee that f corresponds to a unitary transformation. We use a standard trick to transform f into a U_f , which is guaranteed to be unitary:

$$U_f(x \otimes y) = x \otimes (y \oplus f(x))$$

That is, U_f maintains the state of the input qubit and puts the result of $f(x)$ onto the second qubit in the form of $y \oplus f(x)$.

We can now recall Deutsch’s algorithm:

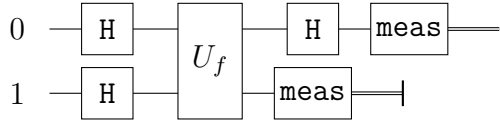
Definition deutsch U_f :

Box One Bit :=

box () \Rightarrow

```

let x      ← H $ init0 $ ();
let y      ← H $ init1 $ ();
let (x,y) ← Uf $ (x,y);
let ()     ← discard $ meas $ y;
meas $ H $ x.
```



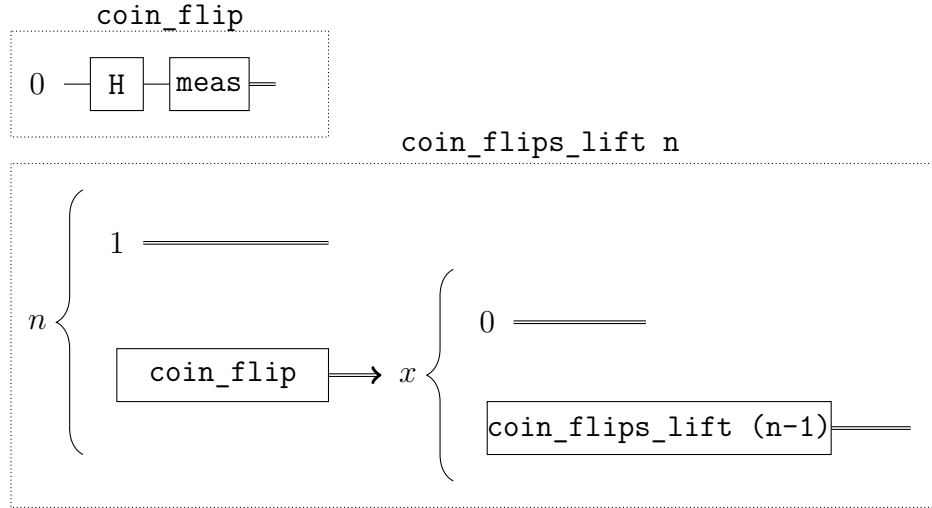
Here are the two statements we would like to prove about Deutsch’s algorithm:

Lemma deutsch_constant : $\forall f, \text{constant } f \rightarrow$
 $\llbracket \text{deutsch } (\text{fun_to_box } f) \rrbracket \text{ I1} = |0\rangle\langle 0|.$

Lemma deutsch_balanced : $\forall f, \text{balanced } f \rightarrow$
 $\llbracket \text{deutsch } (\text{fun_to_box } f) \rrbracket \text{ I1} = |1\rangle\langle 1|.$

We will first give an informal proof of the algorithm’s correctness. In the first case, if f is constant then either $f(x) = 0$ or $f(x) = 1$. This makes U_f the identity matrix, since $y \oplus 0 = y$, or $I_2 \otimes X$, since $y \oplus 1 = \neg y$. These sub-cases are trivial because our two qubits are never entangled, since $I_4 = I_2 \otimes I_2$, and $H(H|0\rangle) = |0\rangle$. In the balanced case, either $f(x) = x$ or $f(x) = 1 - x$. In the first of these, U_f is a *CNOT*, negating y whenever x is 1. In the second, U_f is a reverse-*CNOT* that negates y if x is 0. Either way, x and y will be entangled such that measuring y affects x , putting it into the desired $|0\rangle - |1\rangle$ or $|1\rangle - |0\rangle$ state. Applying a Hadamard yields $|1\rangle$ or $-|1\rangle$, and measuring strips off any minus sign.

In our Coq proof, we can simply do case analysis on f **true** and f **false**, immediately deriving a contradiction if f is not constant/balanced. Our `fun_to_box` constructs the corresponding boxed circuit for us, so we only have to reduce the circuit to its denotation and compute the result. From there, it only takes a bit of arithmetic



```

Fixpoint coin_flips_lift (n : ℕ) : Box One Bit :=
  box () =>
  match n with
  | 0    => new1 $ ()
  | S n' => lift x ← coin_flip $ ();
          if x then coin_flips_lift n' $ ()
          else new0 $ ()
  end.

```

Figure 7.2: Our lifted coin flips circuit.

reasoning to show that the qubit is in the desired basis state.

7.2 Matrix Families and Induction

7.2.1 Many Coins

The examples above might lead the reader to believe that number crunching will suffice to prove all of our algorithms correct or, worse, that *QWIRE* isn't up to the task of verifying families of circuits or more abstract circuits. Neither is true.

We first address families of circuits parameterized by natural numbers. Recall our `coin_flips_lift` circuit from Section 5.6:

```

Fixpoint coin_flips_lift (n : ℕ) : Box One Bit :=
  box () =>
  match n with
  | 0    => new1 $ ()
  | S n' => lift x ← coin_flip $ ();
          if x then coin_flips_lift n' $ ()

```

```

else new_0 $ ()
end.

```

This circuit flips a coin up to n times, stopping and returning tails ($|0\rangle$) if it ever lands tails. If all n tosses come up heads, it returns heads.

We would like to prove that that this circuit simulates a biased coin that returns heads with probability $\frac{1}{2^n}$. This is trivially encoded in the `biased_coin` matrix

$$\begin{pmatrix} 1 - \text{bias} & 0 \\ 0 & \text{bias} \end{pmatrix}$$

with `bias` set to $\frac{1}{2^n}$.

We now want to prove that the circuit's denotation matches its specification:

Lemma `flips_lift_correct` : $\forall n, \llbracket \text{coin_flips_lift } n \rrbracket 'I_1 = \text{biased_coin } (1/2^n)$.

Given that `coin_flips` has a simple recursive definition, it is only natural to prove its correctness using induction. The base case ($n = 0$) simply consists of proving that

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 - \frac{1}{2^0} & 0 \\ 0 & \frac{1}{2^0} \end{pmatrix}$$

which is simple arithmetic.

In the inductive case, the denotation of `coin_flips_lift (n+1)` reduces to the following formula, simplified for readability:

$$|0\rangle\langle 0| \times (\text{hadamard} \times (|0\rangle \times I_1 \times \langle 0|) \times \text{hadamard}) \times |0\rangle\langle 0| .+ \\ \llbracket \text{coin_flips_lift } n \rrbracket (\langle 1| (\text{hadamard} \times (|1\rangle \times I_1 \times \langle 1|) \times \text{hadamard}) \times |1\rangle)$$

Unfortunately, our inductive hypothesis fails us here: $\langle 1| (H \times (|1\rangle \times I_1 \times \langle 1|) \times H) \times |1\rangle$ is not the matrix (1) , in fact, it is $(\frac{1}{2})$. This forces us to prove a more general theorem:

Lemma `flips_lift_correct_gen` : $\forall (n:\mathbb{N}) (p:\mathbb{C}), \llbracket \text{coin_flips_lift } n \rrbracket (p .* I_1) = p .* \text{biased_coin } (1/2^n)$.

In this case, `coin_flips_lift (n+1)` reduces to

$$|0\rangle\langle 0| \times (\text{hadamard} \times (|0\rangle \times (p .* I_1) \times \langle 0|) \times \text{hadamard}) \times |0\rangle\langle 0| .+ \\ \llbracket \text{coin_flips_lift } n \rrbracket (\langle 1| (\text{hadamard} \times (|1\rangle \times (p .* I_1) \times \langle 1|) \times \text{hadamard}) \times |1\rangle)$$

and we can rewrite via our inductive hypothesis, obtaining

$$|0\rangle\langle 0| \times (\text{hadamard} \times (|0\rangle \times (p .* I_1) \times \langle 0|) \times \text{hadamard}) \times |0\rangle\langle 0| .+ \\ (p/2) .* \text{biased_coin } (1/2^n)$$

It is then easy to show that this simplifies to

$$p .* \begin{pmatrix} 1 - \frac{1}{2^{1+n}} & 0 \\ 0 & \frac{1}{2^{1+n}} \end{pmatrix}.$$

We can then apply this lemma to the special case where $p = 1$, giving us our desired result.

7.3 Algebraic Reasoning about Circuits

7.3.1 A Unitary and Its Adjoint

For many circuits, number crunching or even induction won't be enough.

Consider the following simple circuit, which composes an arbitrary unitary gate with its adjoint:

```

Definition unitary_adjoint {W} (U : Unitary W)
  : Box W W :=
  box p => adj U $ U $ p.

```



The function `adj` takes in a unitary gate and returns its adjoint:

```

Fixpoint adj {W} (U : Unitary W) : Unitary W :=
  match U with
  | R_ φ           => R_ (- φ)
  | ctrl U'        => ctrl (adj U')
  | bit_ctrl U'    => bit_ctrl (adj U')
  | U'             => U'
  end.

```

Note that most of our unitaries are their own adjoints. We could also define `adj` as a constructor for unitaries, as we did in earlier versions of `QWIRE`, but that would complicate proofs about our unitary gate set.

Clearly, we cannot prove that `unitary_adjoint` is the identity by simply multiplying matrices: U here can be any unitary gate on any number of wires. Instead, we have to reduce the goal to something of the form

$$\llbracket \text{adj } U \rrbracket \times \llbracket U \rrbracket \times \rho \times \llbracket \text{adj } U \rrbracket \times \llbracket U \rrbracket,$$

show that the denotation of $\llbracket \text{adj } U \rrbracket$ is $\llbracket U \rrbracket^\dagger$, and then use the fact (proved in Chapter 6) that the denotation of every unitary gate is a unitary matrix to replace $\llbracket U \rrbracket^\dagger \times \llbracket U \rrbracket$ with the identity matrix.

Hence, proving a simple fact about unitary gates requires careful management of the goal and the application of lemmas from across the `QWIRE` development.

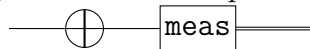
We have yet to use this technique to prove properties of complex circuits, but we suspect that the correctness proof for the quantum fourier transform (or QFT) will be similarly algebraic. Here is the statement of that theorem for the basis qubits, written out mathematically:

$$\llbracket \text{qft} \rrbracket n |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i xy}{2^n}} |y\rangle$$

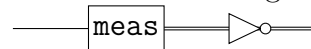
7.4 Equational Rewriting

One of the more powerful techniques for reasoning about circuit behavior and *optimizing* those circuits is equational reasoning. Using a set of basic circuit identities, we could rewrite \mathcal{Q} WIRE circuits, shrinking them and making them easier to run on a real-world quantum computer. One common identity (a unitary followed by its adjoint is equal to the identity) is given in Section 7.3.1. Here, we present some other useful transformations, drawn from Staton’s (2015) equational theory for quantum computation.

Rather than negate a qubit and then measure it, we can always measure the qubit and then negate it. We can represent this as the equivalence of the following circuits:



Definition `X_meas` :=
`box q => meas $ X $ q.`

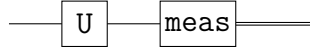


Definition `meas_NOT` :=
`box q => BNOT $ meas $ q.`

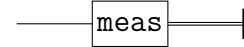
BNOT here is simply the classical (bit-valued) NOT gate.

We can prove this by crunching matrices, but it also proves to be straightforward algebraically: The denotation of `X` is equal to the denotation of `BNOT` and commutes with measurement. Many of the equalities in this section will be similarly easy to prove via computation or matrix rewriting.

The equality of the following two circuits is obvious in the classical setting but less intuitive in the presence of entanglement, where measurement may disturb multiple qubits:



Definition `U_meas_discard U` :=
`box q => discard $ meas $ U $ q.`



Definition `meas_discard` :=
`box q => discard $ meas $ q.`

This says that applying a unitary to a qubit and then measuring and discarding it is the same as simply measuring and then discarding the qubit. We prove the equality of these circuits computationally.

A number of additional equalities drawn from Staton’s theory are available in the file `Equations.v` in the Coq development.

Our rewriting system should also account for dynamic lifting and optimize around it when called for. The following dynamically lifted circuit with is equal to the identity circuit on one `Bit`:


```

Definition lift_new : Box Bit Bit :=
  box b =>
    lift x ← b;
    new x $ ().

```

Here, `new x` is `new1` when `x` is true and, otherwise, `new0`.

While we have a small library for rewriting circuits, an optimizing compiler is still a work-in-progress. Substantial work has gone into rewriting libraries for quantum circuits, most notably using the ZX-calculus (Coecke and Duncan, 2008; Backens, 2014) and the related Quantomatic tool (Kissinger, 2011). This tool faces two challenges, in that ZX graphs are more expressive than quantum circuits, and its rewrite rules are not directed, making it hard to write a terminating procedure for shrinking circuits. A recent paper by Fagan and Duncan (2018) makes some headway on these issues, but we have yet to address the second in the QWIRE setting. Fortunately, QWIRE circuits correspond precisely to valid quantum circuits, so we don't have to convert between representations; however, graphs may well be more amenable to rewriting than sequences of gate applications. This form of rewriting would be a significant step forward for automated circuit optimization and motivates the need for compositionality lemmas of the style proposed in Section 6.2).

Chapter 8

Reversibility

8.1 Ancillae and Assertions

Many quantum algorithms rely heavily on *quantum oracles*, classical programs executed inside quantum circuits. Toffoli (1980) proved that any classical, boolean-valued function $f(x)$ can be implemented as a unitary circuit f_u satisfying $f_u(x, z) = (x, z \oplus f(x))$. Toffoli's construction for quantum oracles is used in many quantum algorithms, such as the modular arithmetic of Shor's algorithm (1999). As a concrete example, Figure 8.1 shows quantum circuits that implement the boolean functions and (\wedge) and or (\vee).

Unfortunately, Toffoli's construction introduces significant overhead. Consider a circuit meant to compute the boolean formula $(a \vee b) \wedge (c \vee d)$. The circuit needs two additional scratch wires, or *ancillae*, to carry the outputs of $(a \vee b)$ and $(c \vee d)$, as seen in Figure 8.2. The annotation 0 at the start of a wire means that qubit is initialized in the state $|0\rangle$. When constructed in this naive way, the resulting circuit no longer corresponds to a unitary transformation and cannot be safely used in a larger quantum circuit.

The solution is to *uncompute* the intermediate values $a \vee b$ and $c \vee d$ and then discard them at the end of the quantum circuit (Figure 8.3). The annotation 0 at the end of a wire is an *assertion* that the qubit at that point is in the zero state, at which point we can safely discard it without affecting the remainder of the state. (If we measured and discarded a non-zero qubit, we would affect whatever qubits it was

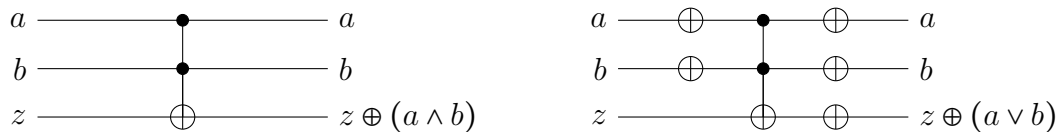


Figure 8.1: Quantum oracles implementing the boolean \wedge and \vee . The \oplus gates represent negation, and \bullet represents control.

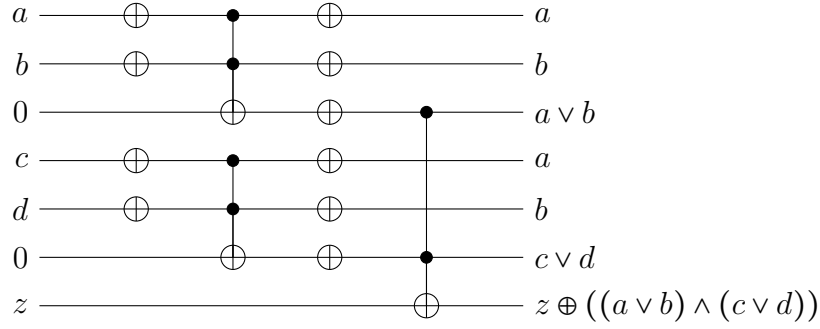


Figure 8.2: An non-unitary quantum oracle for $(a \vee b) \wedge (c \vee d)$

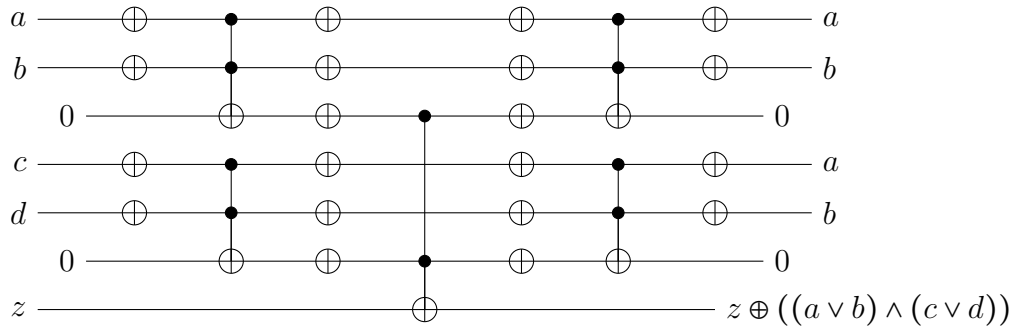


Figure 8.3: A unitary quantum oracle for $(a \vee b) \wedge (c \vee d)$ with ancillae

entangled with.)

How can we verify that such an assertion is actually true? We cannot dynamically check the assertion, since we can only access the value of a qubit by measuring it, thereby collapsing the qubit in question to a 0 or 1 state. However, we can statically reason that the qubit must be in the state $|0\rangle$ by analyzing the circuit semantics.

The claim that a qubit is in the 0 state is a *semantic* assertion about the behavior of the circuit. Unfortunately, this makes it hard to verify—computing the semantics of a quantum program is computationally intractable in the general case. Circuit programming languages often allow users to make such assertions but not to verify that they are true. For example, Quipper (Green et al., 2013a) allows programmers to make assertions about the state of ancillae, but these assertions are never checked. Likewise, in Q# (Svore et al., 2018) the assertion will be checked by a simulator but cannot be checked when a program is run on a quantum computer. Hence, when the qubit is reused, a common use for ancillae which Q# emphasizes, it may be in the wrong state. The QCL quantum circuit language (Ömer, 2005) provides a built-in method for creating reversible circuits from classical functions, but the programmer must trust this method to safely manage ancillae. In a step in the right direction, the REVERC compiler (Amy et al., 2017) for the (non-quantum) reversible computing

language *REVS* (Parent et al., 2017) provides a similar approach to compilation and verifies that it correctly uncomputes its ancilla. However, other assertions in *REVS* that a wire is correctly in the 0 state are ignored if they cannot be automatically verified.

In this chapter, we develop verification techniques for safely working with ancillae. Our approach allows the programmer to discard qubits that are in the state $|0\rangle$ or $|1\rangle$, provided that she first formally proves that the qubits are in the specified state. Inspired by the *REVERC* compiler (Amy et al., 2017), we also provide syntactic conditions that the programmer may satisfy to guarantee that her assertions are true. However, our quantum circuits do not need to match this syntactic specification: a programmer may instead manually prove that her circuit safely discards qubits using the denotational semantics of the language. This gives the programmer the flexibility to use ancillae where the proofs of such assertions are non-trivial.

This chapter makes the follow core contributions:

- We extend *QWIRE* with assertion-bearing ancillae.
- We give semantic conditions for the closely related properties of (a) when a circuit is reversible and (b) when a circuit contains only valid assertions about its ancillae.
- We provide syntactic conditions that guarantee the correctness of these assertions for common use-cases.
- We implement a compiler that transforms boolean expressions into reversible *QWIRE* circuits and prove its correctness.
- We show how this compilation can be used perform quantum arithmetic via a quantum adder.

We should note that the results of this chapter have not yet been fully verified in Coq. In particular, our section on syntactic guarantees for circuits (Section 8.3) describes a number of lemmas that have mostly been sketched out in Coq or proved based on admitted lemmas. Our section on oracles also admits the denotation of two functions that apply CNOT and Toffoli gates to wires based on their positions within the circuit (as in “wire number 7”) instead of referencing named patterns. More broadly, this chapter assumes the correctness of two important lemmas from Section 6.2: `compose_correct` and `inPar_correct`, which say that the denotation of circuits arranged in sequence and in parallel correspond to functional composition and the tensor product, respectively. The lemmas assumed in this chapter (excluding the lemmas of Section 8.3) are summarized in Table 8.1.

Assumption	Description
<code>denote_compose</code>	The composition of circuits is equal to the composition of their denotations (Section 6.2)
<code>inPar_correct</code>	Gives the denotation of circuits composed in parallel
<code>valid_ancillae_box_equal</code>	Our two notations of assertion validity are equivalent
<code>valid_ancillae_unbox</code>	Relates the validity of assertions on boxed and unboxed circuits
<code>[gate]_at_spec</code>	Describes the denotation of the <code>[gate]_at [indices]</code> circuit, which applies a gate to the given indices
<code>ancilla_free_[gate]_at</code>	<code>[gate]_at</code> has no ancillae
<code>ancilla_free_seq</code>	The composition of two ancilla free circuits has no ancillae
<code>strip_one_l_out_eq</code>	Converting a <code>Box W (One ⊗ W')</code> to a <code>Box W W'</code> preserves its denotation
<code>strip_one_r_out_eq</code>	The same for <code>Box W (W' ⊗ One)</code>
<code>valid_ancillae_box'_equiv</code>	Denotationally equivalent circuits must have the same validity
<code>valid_inSeq</code>	The composition of two valid circuits is valid
<code>HOAS_Equiv_inSeq'</code>	If $c_1 \equiv c'_1$ and $c_2 \equiv c'_2$ then $c_1;;c_2 \equiv c'_1;;c'_2$

Table 8.1: Assumptions underlying the reversible computing development. The first two are in `Composition.v`, the next two are in `Ancilla.v`, two instances of `[gate]_at_spec` are in `Oracles.v` and the remaining assumptions are in `Symmetric.v`.

8.2 *Safe and Unsafe Semantics*

As we saw in Chapters 4 and 5, *QWIRE*'s semantics is given in terms of *density matrices* ρ which represent *mixed states*—distributions over pure quantum states. These chapters left out an important feature of *QWIRE*, however: *QWIRE* actually has *two* semantics, which happen to coincide in the absence of ancillae. The *safe* semantics corresponds to an operational model that does not trust assertions, so an `assertx` gate first measures the input qubit before discarding the result. The *unsafe* semantics assumes that assertions are accurate, so an `assertx` gate simply discards its input qubit without measuring it. The two semantics coincide exactly when all assertions in a circuit are accurate, in which case we call the circuit *valid*.

We will briefly remind the reader of the denotation of *QWIRE* circuits (without padding) along with the safe denotation of `assertx`:

$$\begin{aligned} \text{denote_safe } U \rho &= \llbracket U \rrbracket \rho \llbracket U \rrbracket^\dagger \\ \text{denote_safe } \text{init}_0 \rho &= |0\rangle \rho \langle 0| \\ \text{denote_safe } \text{init}_1 \rho &= |1\rangle \rho \langle 1| \\ \text{denote_safe } \text{meas} \rho &= |0\rangle \langle 0| \rho |0\rangle \langle 0| + |1\rangle \langle 1| \rho |1\rangle \langle 1| \\ \text{denote_safe } \{\text{discard}, \text{assert}_0, \text{assert}_1\} \rho &= \langle 0| \rho |0\rangle + \langle 1| \rho |1\rangle \end{aligned}$$

Under the safe semantics, the assertions `assert0` and `assert1` are treated as a measurement followed by a discard. This is semantically the same as the denotation of `discard`, except that `discard` is guaranteed by the type system to only throw away a classically valued bit. This operation on qubits is safe even if the qubit is in a superposition of $|0\rangle$ and $|1\rangle$, due to the implicit measurement.

The *unsafe* semantics is the same as the safe semantics, except for `assert0` and `assert1`:

$$\begin{aligned} \text{denote_unsafe } \text{assert}_0 \rho &= \langle 0| \rho |0\rangle \\ \text{denote_unsafe } \text{assert}_1 \rho &= \langle 1| \rho |1\rangle \end{aligned}$$

It should be immediately clear why this is unsafe: if ρ isn't in the zero state (in the first case), then an assertion produces a density matrix with a trace less than 1. Operationally, this corresponds to the instruction “throw away this qubit in the zero state,” which is quantum-mechanically impossible in the general case. However, this semantics corresponds to the intended meaning of `assertx` when we know the assertion is true. It also ensures that the composition of `initx` with `assertx` is equivalent to the identity, which allows us to optimize away qubit initialization and discarding.

We can now define what it means for the ancilla assertions in a circuit to be valid.

Definition `valid_ancillae` $W (c : \text{Circuit } W) : \mathbb{P} :=$
`(denote c = denote_unsafe c).`

An equivalent definition states that the unsafe semantics preserves the trace of its input, which is always 1, and therefore maps it to a total probability distribution.

Definition `valid_ancillae'` $W (c : \text{Circuit } W) : \mathbb{P} :=$
 $\forall \rho, \text{Mixed_State } \rho \rightarrow \text{trace } (\text{denote_unsafe } c \rho) = 1.$

The second definition follows from the first because the safe semantics is trace preserving. The first follows from the second since `denote_unsafe c ρ` corresponds to a sub-distribution of `denote_safe c ρ`. If its trace is one then they must represent the same distribution.

These two definitions precisely characterize what it means for circuits to have always correct assertions. For brevity, we call such circuits *valid*. In the next section, we define syntactic conditions that are sufficient but not necessary for validity. Programmers will often write syntactically valid circuits like those produced by `compile`

function in Section 8.4), but when needed the semantic definition of validity is still available.

An important property related to the validity of a circuit is its *reversibility*. We say that c and c' are *equivalent*, written $c \equiv c'$, if both their safe and unsafe denotations are equal. (If c and c' are valid, this is equivalent to $\text{denote } c = \text{denote } c'$, but otherwise it is a stronger claim.) Reversibility says that a circuit has a left and right inverse:

Definition `reversible` $\{W_1 W_2\}$ $(c : \text{Box } W_1 W_2) : \mathbb{P} :=$
 $(\exists c', c' ;; c \equiv \text{id_circ}) \wedge (\exists c', c ;; c' \equiv \text{id_circ})$

In Section 8.4, the compiler produces circuits that are their own inverses:

Definition `self_inverse` $\{W\}$ $(c : \text{Box } W W) : \mathbb{P} := c ;; c \equiv \text{id_circ}.$

We can now show that in any reversible circuit all the ancilla assertions hold.

Lemma 8. *If c is reversible, then it is valid.*

Proof. Let c' be c 's inverse. By the second definition of validity, it suffices to show that the trace of $\text{denote_unsafe } c \rho$ is equal to 1 for every initial mixed state ρ . We know that the trace of $\text{denote_unsafe } \text{id_circ } \rho$ is 1; hence,

$$\begin{aligned} 1 &= \text{trace } (\text{denote_unsafe } (c ;; c') \rho) \\ &= \text{trace } (\text{denote_unsafe } c' (\text{denote_unsafe } c \rho)) \end{aligned}$$

Because the unsafe semantics is trace-non-increasing, it must be the case that the trace of $\text{denote_unsafe } c \rho$ is 1 as well. \square

8.3 Syntactically Valid Ancillae

Let c be a circuit made up only of classical gates: the initialization gates, the not gate X , the controlled-not gate $CNOT$, and the Toffoli gate T . Let c' be the result of reversing the order of the gates in c and swapping every initialization with an assertion of the corresponding boolean value. Then every assertion in $c ;; c'$, where semicolons denote sequencing, is valid.

Unfortunately, every circuit of this form is also equivalent to the identity circuit, so as a syntactic condition of validity, this is much too restrictive. In practice, the quantum oracles discussed in the introduction are mostly symmetric, but they introduce key pieces of asymmetry to compute meaningful results. In REVERC, this construction is called the *restricted inverse*; QCL (Ömer, 2005) and Quipper (Green et al., 2013a) take similar approaches.

Let c be a circuit with an equal number of input and output wires whose qubits can be broken up into two disjoint sets: the first n qubits are called the *source*, and the last t qubits are called the *target*. That is, $c : \text{Box } (n+t \otimes \text{Qubit}) (n+t \otimes \text{Qubit})$. The syntactic condition of *source symmetry* on circuits guarantees that c is the identity on

all source qubits. In addition, it guarantees that assertions are only made on source qubits with a corresponding initialization.

A classical gate *acts on the qubit i* if it affects the value of that qubit in an m -qubit system: **X** acts on its only argument, **CNOT** acts on its second argument (the target), and **Toffoli** acts on its third argument.

The property of *source symmetry* on circuits is defined inductively as follows:

- The identity circuit is source symmetric.
- If g is a classical gate and c is source symmetric, then $g ;; c ;; g$ is source symmetric.
- If g is a classical gate that acts on a qubit in the target and c is source symmetric, then both $g ;; c$ and $c ;; g$ are source symmetric.
- If c is source symmetric and i is in the source of c , then $\text{init_at } b \ i ;; c ;; \text{assert_at } b \ i$ is source symmetric.

The key property of a source symmetric circuit is that it does not affect the value of its source qubits. We say that a circuit c is a *no-op* at qubit i if, when initialized with a boolean b , the qubit is still equal to b after executing the circuit. We could define this as $\llbracket c \rrbracket (\rho_1 \otimes |b\rangle \langle b| \otimes \rho_2) = \rho'_1 \otimes |b\rangle \langle b| \otimes \rho'_2$ for some $\rho_1, \rho_2, \rho'_1, \rho'_2$, but this would require ρ_1 and ρ_2 (and ρ'_1 and ρ'_2) to be separable, which is an unnecessary restriction. Instead, we use the `valid_ancillae` predicate and say if we initialize an ancilla in state x at i , apply b , and then assert that $i = x$, our assertion will be valid:

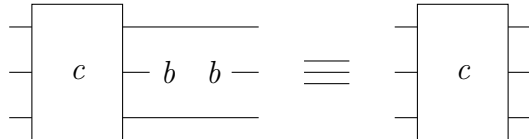
Definition `noop_on` $(m \ k : \mathbb{N}) (c : \text{Box } (\text{Qubits } (1 + m)) (\text{Qubits } (1+m))) : \mathbb{P}$
`:=`
 $\forall b, \text{valid_ancillae } (\text{init_at } b \ i ;; c ;; \text{assert_at } b \ i).$

We similarly define a predicate, `noop_on_source n`, that says that a given circuit is a no-op on each of its first n inputs.

These inductive definitions allow us to state a number of closely related lemmas about symmetric circuits:

Lemma 9. *If the classical gate g acts on the qubit k and $i \neq k$, then g is a no-op on i .*

Lemma 10. *Let c be a circuit such that $c ;; \text{assert_at } b \ i$ is a valid assertion. Then $c ;; \text{assert_at } b \ i ;; \text{init_at } b \ i \equiv c$.*



Lemma 11. *If c and c' are both no-ops on qubit i , then $c ;; c'$ is also a no-op on qubit i .*

Conjecture 1. *If c is source symmetric, then it is a no-op on its source.*

These lemmas have been admitted, rather than proven, in the Coq development (Symmetric.v). Conjecture 1 is labeled as a *conjecture* rather than a lemma, since we do not yet have a paper proof of the statement. It may be the case that we need to strengthen our definition of no-op for this conjecture to hold.

Since all ancillae in a source symmetric circuit occur on sources, we can prove from the statements above that source symmetric circuits are valid.

Theorem 10. *If c is source symmetric, then all its assertions are valid.*

Source symmetric circuits also satisfy a more general property: they are reversible.

The inverse of a source symmetric circuit is defined by induction on source symmetry:

- The inverse of the identity circuit is the identity;
- The inverse of $g ;; c ;; g$ is $g ;; c^{-1} ;; g$;
- The inverses of $c ;; g$ and $g ;; c$ are $g ;; c^{-1}$ and $c^{-1} ;; g$; and
- The inverse of $\text{init_at } b \ i ;; c ;; \text{assert_at } b \ i$ is $\text{init_at } b \ i ;; c^{-1} ;; \text{assert_at } b \ i$.

Clearly, the inverse of any source symmetric circuit is also source symmetric, and the inverse is involutive, meaning $(c^{-1})^{-1} = c$.

Theorem 11. *If c is source symmetric, then $c^{-1} ;; c$ is equivalent to the identity circuit.*

Proof. By induction on the proof of source symmetry. The only interesting case is the case for ancilla, showing

$$\text{init_at } b \ i ;; c^{-1} ;; \text{assert_at } b \ i ;; \text{init_at } b \ i ;; c ;; \text{assert_at } b \ i \equiv \text{id_circ.}$$

From Theorem 10 we know that the circuit $\text{init_at } b \ i ;; c^{-1} ;; \text{assert_at } b \ i$ is valid. Then, by Lemma 10, we know that $\text{init_at } b \ i ;; c^{-1} ;; \text{assert_at } b \ i ;; \text{init_at } b \ i$ is equivalent to $\text{init_at } b \ i ;; c^{-1}$. Thus the goal reduces to $\text{init_at } b \ i ;; c^{-1} ;; c ;; \text{assert_at } b \ i$. This is equivalent to the identity by the induction hypothesis plus the fact that $\text{init_at } b \ i ;; \text{assert_at } b \ i$ is the identity. \square

We can now say that any circuit followed by its inverse is valid. But this theorem is easily extensible. For instance, we can add the following to our inductive definition of symmetric, and the theorem will still hold:

- If c is source symmetric, and $c \equiv c'$, then c' is source symmetric.

This extension allows us to use existing (semantic) equivalences to satisfy our (syntactic) source symmetry predicate, which in turn proves the semantic property of validity. For example, because teleportation is semantically equivalent to the identity circuit, we know trivially that it is valid, even though it is not source symmetric. The Coq development provides many useful compiler optimizations in the file `Equations.v` that can now be used in establishing source symmetry.

8.4 Compiling Oracles

Now that we have syntactic guarantees for circuit validity, we can consider a compiler from boolean expressions to source-symmetric circuits, producing the quantum oracles described in the introduction to this chapter. The resulting circuits are all source symmetric, so it follows from the previous section that they are valid.

We begin with a small boolean expression language, borrowed from Amy et al. (2017), with variables, constants, negation (\neg), conjunction (\wedge), and exclusive-or (\oplus).

$$b ::= x \mid \text{true} \mid \text{false} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \oplus b_2$$

The *interpretation function* $\llbracket b \rrbracket_{\mathbf{f}}$ takes a boolean expression b and a valuation function $\mathbf{f} : \text{Var} \rightarrow \mathbb{B}$ and returns the value of the boolean expression with the variables assigned as in \mathbf{f} .

The compiler takes a boolean expression b and a *map* Γ from the variables of b to the wire indices¹. The resulting circuit `compile b Γ` has $|\Gamma|+1$ qubit-valued input and output wires, where $|\Gamma|$ is the number of variables in the scope of b .

The compiler uses `init_at`, `assert_at`, `X_at`, `CNOT_at`, and `Toffoli_at` circuits, each of which applies the corresponding gate to the given index in the list of n wires. We show the `compile` function below.

¹In the Coq development, these maps are represented by linear typing contexts.

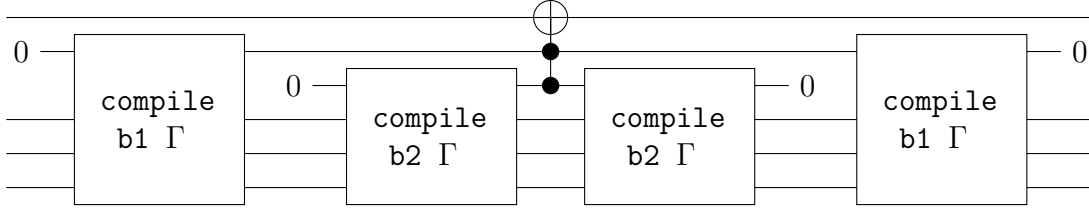


Figure 8.4: Compiling $b_1 \wedge b_2$ on 3 qubits. The top wire is the target.

```

Fixpoint compile (b : bexp) (Γ : Ctx) : Square_Box (S [(Γ)]) ⊗ Qubit) :=
  match b with
  | b_t      ⇒ TRUE || id_circ
  | b_f      ⇒ FALSE || id_circ
  | b_var v  ⇒ CNOT_at (1 + index v Γ) 0
  | b_not b  ⇒ init_at true 1           ;;
               id_circ || (compile b Γ) ;;
               CNOT_at 1 0             ;;
               id_circ || (compile b Γ) ;;
               assert_at true 1
  | b_and b1 b2 ⇒ init_at false 1      ;;
               id_circ || compile b1 Γ ;;
               init_at false 2        ;;
               id_circ || id_circ || compile b2 Γ ;;
               Toffoli_at 1 2 0       ;;
               id_circ || id_circ || compile b2 Γ ;;
               assert_at false 2      ;;
               id_circ || compile b1 Γ ;;
               assert_at false 1
  | b_xor b1 b2 ⇒ init_at false 1      ;;
               id_circ || compile b1 Γ ;;
               CNOT_at 1 0            ;;
               id_circ || compile b1 Γ ;;
               id_circ || compile b2 Γ ;;
               CNOT_at 1 0            ;;
               id_circ || compile b2 Γ ;;
               assert_at false 1
  end.

```

We make heavy use of sequencing (;) and parallel (||) operators in defining this circuit. The `TRUE` case outputs the exclusive-or of `true` with the target wire, which is equivalent to simply negating the target wire. Similarly, `FALSE` reduces to the identity. The variable case `b_var` applies a CNOT gate from the variable's associated wire to the target, thereby sharing its value.

The AND case (Figure 8.4) is more interesting. We first initialize a qubit in the 0 state and recursively compile the value of `b1` to it. We then do the same for `b2`. We

apply a Toffoli gate from $\mathbf{b1}$ and $\mathbf{b2}$, now occupying the 1 and 2 positions in our list, to the target qubit at 0. We then reapply the symmetric functions `compile b2 Γ` and `compile b1 Γ` to their respective wires, returning the ancillae to their original states and discarding them. We are left with the target wire \mathbf{z} holding the boolean value $b_z \oplus (b_1 \wedge b_2)$ and $|\Gamma|$ wires retaining their initial values.

Finally, we have the XOR case. Here we borrow a trick from REVERC (Amy et al., 2017) and allocate only a single ancilla instead of the two we used in the AND case. Instead of calculating $(b_1 \oplus b_2) \oplus t$, where t is the target, we calculate the equivalent $b_2 \oplus (b_1 \oplus t)$, taking advantage of the associativity and commutativity of \oplus . Hence, as soon as we've computed b_1 , we can apply a *CNOT* from b_1 to the target and immediately uncompute b_1 . This frees up our ancilla, which we then use as a target for `compile b2`.

Note that our entire `compile` circuit is source symmetric, and therefore our assertions are guaranteed to hold by Theorem 10.

We can now go about proving the correctness of this compilation.

Theorem `compile_correct` : $\forall (b : \text{bexp}) (\Gamma : \text{Ctx}) (f : \text{Var} \rightarrow \mathbb{B}) (z : \mathbb{B}),$
`vars b \subseteq domain $\Gamma \rightarrow$`
`[[compile b Γ]] (bool_to_matrix t \otimes basis_state Γ f) =`
`bool_to_matrix (z \oplus [[b]]f) \otimes basis_state Γ f.`

The function `basis_state` takes the wires referenced by Γ and the assignments of f and produces the corresponding basis state. This forms the input to the compiled boolean expression along with the target, a classical qubit in the $|0\rangle$ or $|1\rangle$ state. The statement of `compile`'s correctness says that when we apply `[[compile b Γ]]` to this basis state with an additional target qubit, we obtain the same matrix with the result of the boolean expression on the target. The proof follows by induction on the boolean expression.

8.5 Quantum Arithmetic in Qwire

In this section, we show how to use the compiler from the previous section to implement a quantum adder, which has applications in many quantum algorithms, including Shor's algorithm. A verified quantum adder is therefore an important step towards verifying a variety of quantum programs.

The input to an adder consists of two n -bit numbers represented as sequences of bits $x_{1:n}$ and $y_{1:n}$, as well as a carry-in bit c_{in} . The output consists of the sum $sum_{1:n}$ and the carry-out c_{out} .

To begin, consider a simple 1-bit adder that takes in three bits, c_{in} , x , and y , and computes their sum and carry-out values. The sum is equal to $x \oplus y \oplus c_{in}$, and the carry is $(c_{in} \wedge (x \oplus y)) \oplus (x \wedge y)$. The expressions can be compiled to 4- and 5-qubit circuits `adder_sum` and `adder_carry`, respectively, where the order of qubits is c_{out} , sum , y , x , and c_{in} .

```

Definition adder_sum : Box (4  $\otimes$  Qubit) (4  $\otimes$  Qubit) :=
  compile ((c_in  $\wedge$  (x  $\oplus$  y))  $\oplus$  (x  $\wedge$  y)) (list_of_Qubits 4).
Definition adder_carry : Box (5  $\otimes$  Qubit) (5  $\otimes$  Qubit) :=
  compile (x  $\oplus$  y  $\oplus$  c_in) (list_of_Qubits 5).
Definition adder_1 : Box (5  $\otimes$  Qubit) (5  $\otimes$  Qubit) :=
  adder_carry ;; (id_circ  $\parallel$  adder_sum).

```

Here, `adder_sum` computes the sum of its three input bits and `adder_carry` computes the carry, ignoring the result of `adder_sum`. Semantically, the adder should produce the appropriate boolean values; the operation `bools_to_matrix` converts a list of booleans to a density matrix.

```

Lemma adder_1_spec :  $\forall$  (cin x y sum cout :  $\mathbb{B}$ ),
  [[adder_1]] (bools_to_matrix [cout; sum; y; x; cin])
= (bools_to_matrix [ cout  $\oplus$  (c_in  $\wedge$  (x  $\oplus$  y))  $\oplus$  (x  $\wedge$  y));
    ; sum  $\oplus$  (x  $\oplus$  y  $\oplus$  c_in)
    ; y; x; cin]).

```

Next, we extend the 1-qubit adder to n qubits. The n -qubit adder contains two parts—`adder_left` and `adder_right`—defined recursively using padded `adder_1` and `adder_carry` circuits. The left part computes the sum and carry sequentially from the least significant bit, initializing an ancilla for the carry in each step. When it reaches the most significant bit, it computes the most significant bit of the sum and carry-out using the 1-qubit adder. The right part of the adder uncomputes the carries and discards the ancillae. The definitions of the circuits are shown below and illustrated in Figure 8.5.

```

Fixpoint adder_left (n :  $\mathbb{N}$ ) : Box ((1+3*n)  $\otimes$  Qubit) ((1+4*n)  $\otimes$  Qubit) :=
  match n with
  | S n'  $\Rightarrow$  (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (adder_left n')))) ;;
    (init_at false (4*n) 0) ;;
    (adder_1_pad (4*n'))
  end.
Fixpoint adder_right (n :  $\mathbb{N}$ ) : Box ((1+4*n)  $\otimes$  Qubit) ((1+3*n)  $\otimes$  Qubit) :=
  match n with
  | 0  $\Rightarrow$  id_circ
  | S n'  $\Rightarrow$  (adder_carry_pad (4*n')) ;;
    (assert_at false (4*n) 0) ;;
    (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (adder_right n'))))
  end.
Fixpoint adder_circ (n :  $\mathbb{N}$ ) : Box ((2+3*n)  $\otimes$  Qubit) ((2+3*n)  $\otimes$  Qubit) :=
  match n with
  | 0  $\Rightarrow$  id_circ
  | S n'  $\Rightarrow$  (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (adder_left n'))))));;
    (adder_1_pad (4*n')) ;;
    (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (id_circ  $\parallel$  (adder_right n'))))));;
  end.

```

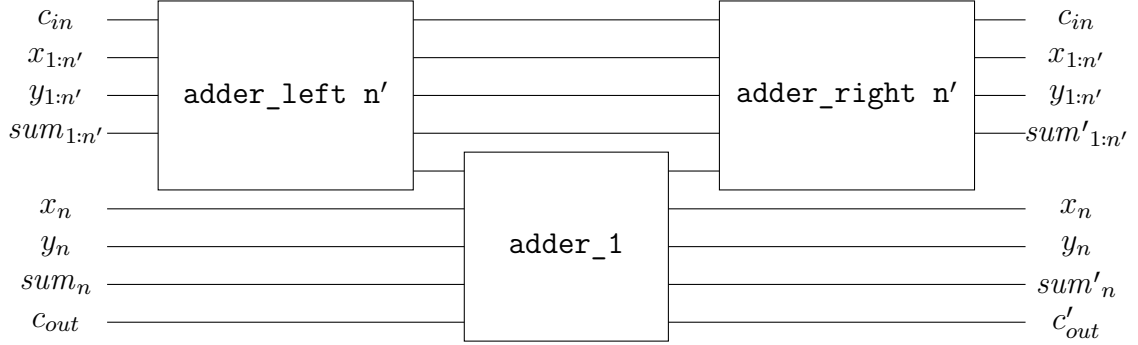


Figure 8.5: A quantum circuit for the n -adder where $n' = n - 1$. The n' ancillae created in `adder_left` are all terminated inside `adder_right`.

We can now prove the correctness of the n -qubit adder:

```

Lemma adder_circ_n_spec :  $\forall$  (n :  $\mathbb{N}$ ) (f : Var  $\rightarrow$   $\mathbb{B}$ ),
  let li := list_of_Qubits (2 + 3 * n) in
  [[adder_circ_n n]] (ctx_to_matrix li f)
  = (ctx_to_matrix li (compute_adder_n n f)).

```

Like `bools_to_matrix` above, `ctx_to_matrix` takes in a context and an assignment f of variables to booleans and constructs the corresponding density matrix. The function `compute_adder_n` likewise takes a function f that assigns values to each of the $3 * n + 2$ input variables and returns a boolean function f' representing the state of the same variables after addition (computed classically). The specification states that the n -bit adder circuit computes the state corresponding to the function `compute_adder_n` for any initial assignment.

Note that the lemma gives a correspondence between the denotation of the circuit and functional computation on the assignment. This can reduce the time required to verify more complex arithmetic circuits. A natural next step is to verify the correspondence between our functions on lists of booleans and Coq's binary representations of natural numbers, thereby grounding our results in the Coq standard library and allowing us to easily move between numerical representations.

8.6 Next Steps for Reversible Computation

As noted throughout this chapter, our investigation into reversible computing in `QWIRE` is a work in progress. The obvious next step for this line of work is to prove all the outstanding claims in this chapter, which mainly relate to syntactically guaranteeing circuit validity. But there are also some natural steps after that.

As the reader may have noticed in the previous section, our verified adder doesn't make heavy use of our `compile` function. Instead, compilation is only used for the base cases. This is unfortunate, since in principle we should be able to write a complete

adder in our boolean expression language and then compile that to a circuit. In order to accomplish this, we would need to add the following features to our `bexp` language:

1. Pairs would allow us to represent binary numbers, where `(true,(true,(false,true)))` could represent 1101 or 13.
2. Projection operators would allow us to extract values from pairs.
3. Let bindings would allow us to reuse sub-circuits for efficiency.

We could also make our `bexps` dependently typed, which would allow us to associate `bexps` with the number of wires entering and exiting the corresponding circuit. We could even include types for n -bit numbers that correspond to product types. And naturally, there is much more that we could do with the `bexp` language, including adding lambdas, branching, recursion, and other common programming language idioms.

We would also like to optimize our current compiler. Our `compile` function borrows a trick from REVERC (Amy et al., 2017), in that it doesn't use additional ancilla to compile exclusive-ors. However, there is a lot of optimization that could potentially be done, and, given the limitations of today's quantum computers, it is all worth doing.

Finally, a recent innovation in the area of quantum computing concerns so-called *dirty ancillae*. We call an ancilla "dirty" if it may be initialized in an arbitrary state, not only $|0\rangle$. Häner et al. (2016) show that these can take the place of our "clean" ancillae in many quantum circuits, and Q# (Svore et al., 2018) allows us to assert that a qubit has been returned to its initial state, whatever that state may be. Extending the work in this paper to verify that dirty ancillae are returned to their initial states would require substantial additional machinery, however we believe that the payoff in terms of programming and verification justifies the added effort.

Chapter 9

Automation

The entire *QWIRE* language makes heavy use of Coq automation tools, particularly its *LTac* tactic language and its hint databases. In this chapter, we will look at some of the more interesting automation in *QWIRE*, with a particular focus on typechecking and verification.

9.1 Typechecking Circuits

Typechecking *QWIRE* circuits can be a difficult task. Before we delve into the tactics, let's review the typing rules from Chapter 5:

```
Inductive Types_Circuit : 0Ctx → ∀ {w}, Circuit w → Set :=
| types_output : ∀ {Γ w} {p : Pat w}, Γ ⊢ p :Pat → Γ ⊢ output p :Circ
| types_gate : ∀ {Γ Γ1 Γ1' w1 w2 w} {f : Pat w2 → Circuit w}
               {p1 : Pat w1} {g : Gate w1 w2},
               Γ1 ⊢ p1 :Pat →
               (∀ Γ2 Γ2' (p2 : Pat w2) {pf2 : Γ2' == Γ2 • Γ},
                Γ2 ⊢ p2 :Pat → Γ2' ⊢ f p2 :Circ) →
               ∀ {pf1 : Γ1' == Γ1 • Γ},
               Γ1' ⊢ gate g p1 f :Circ
| types_lift : ∀ {Γ1 Γ2 Γ w } {p : Pat Bit} {f : ℤ → Circuit w},
               Γ1 ⊢ p :Pat →
               (∀ b, Γ2 ⊢ f b :Circ) →
               ∀ {pf : Γ == Γ1 • Γ2},
               Γ ⊢ lift p f :Circ.
```

```
Definition Typed_Box {W1 W2 : WType} (b : Box W1 W2) : Set :=
  ∀ Γ (p : Pat W1), Γ ⊢ p :Pat → Γ ⊢ unbox b p :Circ.
```

Now let's try typing a simple circuit, following the derivation in Figure 9.1. That simplified presentation follows the style of Chapter 4, where each Γ_i is simply the singleton context that types p_i .

Here is the corresponding *QWIRE* program in our imperative notation.

$$\frac{\Gamma_1, \Gamma_2 \vdash (p_1, p_2) \quad \frac{\Gamma_0, \Gamma_3, \Gamma_4 \vdash (p_0, p_3, p_4)}{\Gamma_0, \Gamma_3, \Gamma_4 \vdash \text{output } (p_0, p_3, p_4)}}{\Gamma_1, \Gamma_2, \Gamma_0 \vdash \text{gate } (p_3, p_4) \leftarrow \text{CNOT } (p_1, p_2); \text{output } (p_0, p_3, p_4)}$$

Figure 9.1: The typing derivation for a simple `QWIRE` program that applies a CNOT to the last two qubits in a 3-qubit system. We haven't expanded out the proofs for typing patterns since they're trivial.

```

Definition cnot12 : Square_Box (Qubit ⊗ Qubit ⊗ Qubit) :=
  box (p0,p1,p2) =>
    gate (p3,p4) ← CNOT @(p1,,p2);
    output (p0,p3,p4).

```

Note that we have to box the circuit to create a closed term. We have avoided shadowing any variables for ease of presentation.

Let's walk through a manual proof that `cnot12` is well-typed.¹ We can begin by unfolding the definition of `Typed_Box`, obtaining the following proof state:

```

∀ (Γ : OCtx) (p : Pat (Qubit ⊗ Qubit ⊗ Qubit)),
  Γ ⊢ p :Pat →
  Γ ⊢ unbox (box (p0,p1,p2) =>
    gate (p3,p4) ← CNOT @(p1,,p2);
    output (p0,p3,p4)) p :Circ

```

We now conveniently have an input pattern p , a typing context Γ , and a proof that Γ types p . We will introduce these into our context and repeatedly invert the typing judgment, obtaining the following proof state:

```

p0, p1, p2 : Pat Qubit
Γ2, Γ0, Γ1 : OCtx
V : is_valid (Γ0 ∪ Γ1 ∪ Γ2)
V0 : is_valid (Γ0 ∪ Γ1)
TP0 : Γ0 ⊢ p0 :Pat
TP1 : Γ1 ⊢ p1 :Pat
TP2 : Γ2 ⊢ p2 :Pat
====
(Γ0 ∪ Γ1 ∪ Γ2) ⊢ (gate_ (p3, p4)← CNOT @(p1,, p2);
  output (p0,, p3,, p4))) :Circ

```

This looks a lot more like our conclusion in Figure 9.1 (though we have cleaned it up slightly). It's time to apply a typing rule:

```

apply types_gate with (Γ := Γ0) (Γ1 := Γ1 ∪ Γ2); try solve_merge.

```

There are two worrisome things about this approach. Firstly, we had to explicitly tell Coq how to break $\Gamma_0 \cup \Gamma_1 \cup \Gamma_2$ into one context that would type (p_1, p_2) and a

¹This is `cnot12_WT_manual` in `Typechecking.v`.

second context that would type the rest of the circuit. We also had to call a tactic, `solve_merge`, to discharge an obligation of the form $\Gamma_1 \uplus \Gamma_2 == \Gamma_1 \bullet \Gamma_2$. In this case, the solution was easy: `split` gives us the goals $\Gamma_1 \uplus \Gamma_2 = \Gamma_1 \uplus \Gamma_2$, which is trivial, and `is_valid` ($\Gamma_1 \uplus \Gamma_2$), which follows from our hypothesis $\forall : \text{is_valid} (\Gamma_0 \uplus \Gamma_1 \uplus \Gamma_2)$. But this won't always be so easy, as we will see later.

We now have two goals, corresponding to the second row in Figure 4.2. In that derivation, the solution to the first $((\Gamma_1 \uplus \Gamma_2) \vdash (p1,,p2))$ was simple enough that we didn't bother writing it; in Coq, it requires another explicit application and call to `solve_merge`:

```
apply types_pair with ( $\Gamma_1 := \Gamma_1$ ) ( $\Gamma_2 := \Gamma_2$ ); try solve_merge.
```

The cases that follow from that actually are trivial for Coq to solve: They consist of applying our hypotheses TP1 and TP2.

Let's now take a look at our remaining goal, corresponding to the continuation:

```
 $\forall (\Gamma \Gamma' : \text{OCtx}) (p : \text{Pat} (\text{Qubit} \otimes \text{Qubit})),$   

 $\Gamma' == \Gamma \bullet \Gamma_0 \rightarrow \Gamma \vdash p : \text{Pat} \rightarrow$   

 $\Gamma' \vdash (\text{let } (p3,p4) \leftarrow p; (\text{output } (p0,, p3,, p4))) : \text{Circ}$ 
```

Once again, destructing the typing judgment gives us more useful patterns and contexts:

```
 $\Gamma' \vdash \text{output } (p0,, p3,, p4) : \text{Circ}$ 
```

with $\Gamma' = \Gamma_3 \uplus \Gamma_4 \uplus \Gamma_0$ and proofs that each p_i types p_i in the context.

We can now apply `types_output` and solve for the remaining patterns. The full proof is given in Figure 9.2.

Unfortunately, this proof isn't quite automatic. At several points, we stopped and manually specified contexts based on our knowledge of how the final proof should look. On the other hand, this process *should* be automatic because the typing derivation is *syntax directed*. As we see in Figure 9.1., we can fill in this derivation by blindly applying the appropriate rules from bottom to top, leaving the typing contexts out. The only challenge, then, is in finding the correct contexts. We can obtain these from the leaves of the typing derivation, then percolate the typing contexts downwards.

In order to mechanize this strategy, we use Coq's `evars`, or existential variables. An `evar` acts like a placeholder whose value we can fill in later.² Using `evars`, we can leave some context unspecified until we determine their values at the leaves of the typing derivation.

We can now present a "more automatic" version of our typing derivation in Figure 9.3.

Let's walk through this proof. The first change from the previous proof is that our call to `intros` doesn't specify any names for our variables, instead letting Coq name

²Naturally, this is subject to restrictions. An `evar` can only be unified with a value that was in scope at the time it was created. Otherwise, we could use them to prove false statements, like $\exists x, \forall y, x = y$, by replacing x with an `evar`, introducing y , and then unifying x with y .

```

Lemma cnot12_WT_manual : Typed_Box cnot12.
Proof.
  unfold Typed_Box, cnot12.
  intros  $\Gamma$  p TP. simpl.
  dependent destruction TP.
  dependent destruction TP1.
  (* Give contexts and patterns the correct names *)
  rename  $\Gamma_0$  into  $\Gamma$ ,  $\Gamma_1$  into  $\Gamma_0$ . rename  $\Gamma$  into  $\Gamma_1$ .
  rename p3 into p1.
  rename TP1_1 into TP0, TP1_2 into TP1.
  (* Apply gate typing rule *)
  apply @types_gate with ( $\Gamma := \Gamma_0$ ) ( $\Gamma_1 := \Gamma_1 \uplus \Gamma_2$ ); try solve_merge.
- (* types (p1,p2) *)
  apply types_pair with ( $\Gamma_1 := \Gamma_1$ ) ( $\Gamma_2 := \Gamma_2$ ); try solve_merge.
  + apply TP1. (* types p1 *)
  + apply TP2. (* types p2 *)
- (* types `output (p0, p3, p4)` *)
  intros  $\Gamma$   $\Gamma'$  p M TP.
  dependent destruction TP.
  apply (@types_output _ _ _ _ eq_refl).
  (* types (p0, p3, p4) *)
  apply types_pair with ( $\Gamma_1 := \Gamma_0 \uplus \Gamma_3$ ) ( $\Gamma_2 := \Gamma_4$ ); try solve_merge.
  + (* types (p0, p3) *)
    apply types_pair with ( $\Gamma_1 := \Gamma_0$ ) ( $\Gamma_2 := \Gamma_3$ ); try solve_merge.
    * apply TP0. (* types p0 *)
    * apply TP3. (* types p3 *)
  + apply TP4. (* types p4 *)
Qed.

```

Figure 9.2: Manually typechecking cnot12.

```

Lemma cnot12_WT_evars : Typed_Box cnot12.
Proof.
  unfold Typed_Box, cnot12.
  intros; simpl.
  invert_patterns.
  eapply types_gate.
  Focus 1.
    eapply @types_pair. (* types (p1, p2) *)
      4: eauto. (* types p2 *)
      3: eauto. (* types p1 *)
      2: monoid. (* unifies ? $\Gamma = \Gamma_1 \uplus \Gamma_2$  *)
      1: validate. (* solves is_valid ( $\Gamma_1 \uplus \Gamma_2$ ) *)
  Focus 2. (* 3 *)
    split. (* _ == _ • _ *)
      2: monoid. (* unifies  $\Gamma_0 \uplus \Gamma_1 \uplus \Gamma_2 = \Gamma_1 \uplus \Gamma_2 \uplus ?\Gamma$  *)
      1: validate. (* solves `is_valid ( $\Gamma_0 \uplus \Gamma_1 \uplus \Gamma_2$ )` *)
  Focus 1. (* 2 *)
    intros; simpl.
    invert_patterns.
    eapply @types_output.
    Focus 1.
      monoid.
    Focus 1. (* 2 *)
      destruct_merges; subst.
      eapply @types_pair.
      Focus 4.
        eauto. (* types p4 *)
      Focus 3.
        eapply @types_pair. (* types (p0,p3) *)
          4: eauto. (* types p3 *)
          3: eauto. (* types p0 *)
          2: monoid. (* unifies ? $\Gamma = \Gamma_0 \uplus \Gamma_3$  *)
          1: validate. (* solves `is_valid ( $\Gamma_1 \uplus \Gamma_2$ )` *)
      Focus 2.
        monoid. (* unifies  $\Gamma_3 \uplus \Gamma_4 \uplus \Gamma_0 = \Gamma_0 \uplus \Gamma_3 \uplus \Gamma_4$  *)
      Focus 1.
        validate. (* solves `is_valid ( $\Gamma_1 \uplus \Gamma_2$ )` *)
Qed.

```

Figure 9.3: Typechecking cnot12 using evars.

them itself. This is because we don't want to refer to any specific terms or hypotheses in our proof, since it should be fully automatable. The next line calls `invert_patterns`. This corresponds to the `dependent destructions` in our manual proof, breaking down complex patterns into bits, qubits, and units, and simultaneously breaking down the contexts that type them. We can now apply our first typing rule.

Where previously we called `apply types_gate` and specified values for Γ_1 and Γ_2 , here we simply call `eapply types_gate`. This applies the same rule, but instead of using our specified values for Γ_1 and Γ_2 , it leaves them as `evars` to be filled in later. We are left with the following three goals to prove (we have replaced Coq's generated names with our own, for readability):

```
subgoal 1 (ID 1463) is:
  ?Γ1 ⊢ (p1,,p2) :Pat
```

```
subgoal 2 (ID 1464) is:
  ∀ (Γ Γ' : 0Ctx) (p : Pat (Qubit ⊗ Qubit)),
  Γ' == Γ • ?Γ2 → Γ ⊢ p :Pat →
  Γ' ⊢ (let p3 p4 ← p; (output (p0,, p3,, p4))) :Circ
```

```
subgoal 3 (ID 1465) is:
  (Γ0 ∪ Γ1 ∪ Γ2) == ?Γ1 • ?Γ2
```

Here, the structure of our proof changes. We can no longer simply solve our goals one-by-one, nor can we immediately discharge goals of the form $\Gamma == \Gamma_1 \bullet \Gamma_2$ with a quick `solve_merge`. Our second goal cannot be solved without finding a value for $? \Gamma_2$, whereas our third goal needs to find a value for $? \Gamma_1$ *or* $? \Gamma_2$ to perform unification. Hence, we are forced to address the goals in the following order: first, third, and then second. We use `Focus n` or `n:` to address a specific goal whose index may be decremented during this process.

We won't dwell on subgoal 1: Its structure completely parallels the structure of the broader proof. Its outcome is to type `(p1,,p2)` with the contexts Γ_1 and Γ_2 , thereby telling Coq that $? \Gamma_1$ is $\Gamma_1 \cup \Gamma_2$, which percolates to the other subgoals.

We can now proceed to subgoal 3. It asks us to prove that $(\Gamma_0 \cup \Gamma_1 \cup \Gamma_2) == (\Gamma_1 \cup \Gamma_2) \bullet ? \Gamma_2$, which we can split into its components: `is_valid` $(\Gamma_0 \cup \Gamma_1 \cup \Gamma_2)$ and $\Gamma_0 \cup \Gamma_1 \cup \Gamma_2 = \Gamma_1 \cup \Gamma_2 \cup ? \Gamma_2$. We start with the second case: Our `monoid` can solve any valid merge equality that involves a single `evar`, and here it quickly recognizes that $? \Gamma_2$ must be Γ_0 . We then call our `validate` tactic, which checks that $\Gamma_0 \cup \Gamma_1 \cup \Gamma_2$ is a valid context. We will discuss the mechanics of `monoid` and `validate` shortly. Note that in this context we could have called `monoid` and `validate` in either order since the `is_valid` subgoal did not involve any `evars`. However, it frequently will, so we take the general practice of calling `monoid`, which can instantiate an `evar` first.

Now only subgoal 2 remains, and asks us to typecheck the continuation:

```
∀ (Γ Γ' : 0Ctx) (p : Pat (Qubit ⊗ Qubit)),
  Γ' == Γ • Γ0 → Γ ⊢ p :Pat →
```

$\Gamma' \vdash (\text{let } p3 \ p4 \leftarrow p; (\text{output } (p0, p3, p4))) : \text{Circ}$

We do this in exactly the same way that we typechecked the broader circuit. The only addition is a call to `destruct_merges`, which replaces any $\Gamma = \Gamma_1 \bullet \Gamma_2$ in the hypotheses with $\Gamma = \Gamma_1 \uplus \Gamma_2$ and `is_valid` Γ , followed by `subst`, which replaces all instances of Γ with $\Gamma_1 \uplus \Gamma_2$.

We can now present the core of our `typecheck` tactic:

1. Unfold `Typed_Box` and the target circuit
2. `intros` hypotheses
3. `invert_patterns`
4. `destruct_merges`
5. `econstructor` (which calls `eapply` on the relevant constructor)
 - (a) `typecheck` any goal of the form $? \Gamma \vdash p : \text{Pat}$ then `eauto`
 - (b) Use `monoid` on any goal of the form $\Gamma_0 \uplus \Gamma_1 = \Gamma_2 \uplus \Gamma_3$ with one `evar` or less
 - (c) Use `validate` on any goal of the form `is_valid exp` without `evars`
 - (d) Call `typecheck` on what remains

Note that step (a) must precede (b), which in turn must precede (c) and (d). We need to typecheck patterns to infer the values of any `evars` and then use `monoid` to instantiate whatever `evars` remain. We can now present our final proof that `cnot12` is well-typed:

Lemma `cnot23_WT` : `Typed_Box cnot12`.

Proof.

`type_check`.

Qed.

Our `type_check` tactic has some additional functionality not captured by the algorithm above. Since our circuits are highly modular and often included in larger circuits, we generally want to reuse proofs of typing judgments wherever possible. In particular, if we compose two circuits together, we should not compute the result in order to type them; rather, we should use our `compose_typing` lemma (corresponding to `TYPECIRCGATE` in Figure 4.2) and then typecheck the component circuits separately. Moreover, if these circuits already have typing proofs, we should apply those directly. In order to facilitate this practice, we maintain a Coq `Hints` database, called `typed_db`. Whenever a new circuit is typechecked, we can add its proof to `typed_db`. Our typechecking tactics always calls out to `typed_db` before attempting to typecheck a circuit, speeding up the typechecking process. The typing proofs for higher-order

functions on circuits, like `inPar` and `inSeq` (which compose circuit in parallel and in sequence), are also added to the database.

We next discuss `monoid` and `validate`, which lie at the heart of our typechecking procedure.

9.1.1 Monoid

Partial Commutative Monoids The structure that underlies the `monoid` tactic is called a *partial commutative monoid*, or PCM for short. Partiality refers to the fact that some operations may lead to an invalid result, called \perp . To be precise, our structure is

$$\{A, \top, \perp, \circ\}$$

for some set A , obeying the following properties:

$$\begin{aligned} a \circ \top &= a \\ a \circ \perp &= \perp \\ (a \circ b) \circ c &= a \circ (b \circ c) \\ a \circ b &= b \circ a \end{aligned}$$

In our case, \circ corresponds to our merge operation \uplus , \top to the empty context \square (which can be safely merged with any other context, yielding that context), and \perp to `Invalid`, which results from an attempt to merge two overlapping contexts. A corresponds to the set of singleton contexts, which we can merge together to form arbitrary typing contexts. We use this structure to type our circuits, via two tactics: The `monoid` tactic shows the equality of two expressions, and the `validate` tactic demonstrates disjointness.

The theory of partial commutative monoids (and the special case of *nilpotent commutative monoids*, where $a \circ a = \perp$ if $a \neq \top$) is explored in more detail in the Linear Typing Contexts library³ by Jennifer Paykin and this author. Unfortunately, a thorough exposition of this development is not yet available, though it is discussed in similar detail to here in Paykin’s thesis (2018). Here we will only focus on the applications of this structure to typechecking linear circuits.

A PCM Solver Our `monoid` tactic is based upon the tactic of the same name in Adam Chlipala’s “Certified Programming with Dependent Types” (Chlipala, 2013) for solving monoidal equalities using reflection. A similar approach is taken by Coq’s built-in tactics `ring` and `field` for solving ring and field equalities. First, an Ltac reifies an expression as a list of base variables, ignoring identity elements \top , flattening out the associativity of \circ , and collapsing the expression to \perp if \perp occurs anywhere in the expression:

³<https://github.com/inQWIRE/LinearTypingContexts>

```

Ltac reify a :=
  match a with
  |  $\top$   $\Rightarrow$  constr:(Some [])
  |  $\perp$   $\Rightarrow$  constr:(None)
  | ?a1  $\circ$  ?a2  $\Rightarrow$  let e1 := reify a1 in
                      let e2 := reify a2 in
                      match e1, e2 with
                      | Some ls1, Some ls2  $\Rightarrow$  constr:(Some (ls1 ++ ls2))
                      | _, _  $\Rightarrow$  constr:(None)
                      end
  | _  $\Rightarrow$  constr:(Some [a])
end.

```

A goal $a1 = a2$ can then be exchanged with one of the form `from_list ls1 = from_list ls2`, where `lsi` is the result of calling `reify` on `ai`. We can then check if `ls1` and `ls2` are permutations of one another and, if so, apply a lemma to solve the goal.

Our extensions to `monoid` add commutativity (by looking for permutations rather than simple equality) and also allow us to unify expressions that contain a single `evvar`. We limit this to one `evvar`, since multiple `evvars` admit multiple solutions, just like an additive equation of the form $a + x = b + y$ admits infinite possible values for x and y . (If x and y were on the same side of the equals sign and constrained to be natural numbers, there would be a finite number of solutions, and the same is true for our `OCtxs`. Unfortunately for our purposes, most finite numbers are not 1.)

Conveniently, our typechecking algorithm above always instantiates all but one `evvar` before calling `monoid`.

9.1.2 Validate

Our `validate` tactic solve goals of the form `is_valid Γ` , where Γ may be the merger of many contexts, thereby showing that Γ is not `Invalid` and can type a circuit. The `validate` tactic makes use of a simple observation: $\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3$ is valid if and only if $\Gamma_1 \uplus \Gamma_2$, $\Gamma_2 \uplus \Gamma_3$, and $\Gamma_1 \uplus \Gamma_3$ are valid. Hence, checking validity amounts to checking pairwise disjointness.

The `validate` tactics first rewrites in the premises to find the largest expressions such that $\Gamma_1 \uplus \dots \uplus \Gamma_n$ is valid. For instance, if it finds a hypothesis $H : \Gamma = \Gamma_1 \uplus \Gamma_2$, and Γ appears in some larger merge expression, it will replace all instances of Γ with $\Gamma_1 \uplus \Gamma_2$ and clear H . It then breaks down our large merge expressions into pairwise claims of validity. Once Coq's hypothesis context is saturated with expressions of validity, we similarly replace the goal with pairwise claims of validity. If we can unify these goals with our existing hypotheses, we've proven validity. If not, it's impossible to show validity since no more information about our abstract contexts can be gathered.

This concludes our discussion of `QWIRE`'s linear typechecker.

9.2 Arithmetic

As noted in Section 5.4, we use the Coquelicot (Boldo et al., 2015) extension of the Coq reals to complex numbers to populate our matrices. This is mainly to make use of Coq’s built-in tactics for solving real number equalities, particularly the linear real arithmetic tactic `lra`. We define an equivalent function for complex numbers called `clra`⁴ that breaks a complex equality $c = c'$ into its real components, `fst c = fst c'` and `snd c = snd c'`, and then solves each using `lra`:

```
Ltac clra := eapply c_proj_eq; simpl; lra.
```

Here `c_proj_eq` says that if the first and second projections are equal, so are the pairs that make up the complex numbers.

Note that `clra`, like `lra`, is what the Mathematical Components library (Mahboubi et al., 2016) calls a *terminating tactic*: It either solves the goal or fails without making progress. We will also need tactics that simplify the goal state without solving it. The most straightforward of these is `Csimpl`, which rewrites using the basic additive and multiplicative identities for 0 and 1 and replaces c^* (our notation for the complex conjugate) with c when the imaginary part is 0. This lightweight technique proves particularly valuable for dealing with the products of sparse matrices, since their elements contain many sub-expressions of the form $A \times y * 0$ and $0^* * B \times y$.

Coq has little automation support for two kinds of formulae that appear frequently in our development: equations involving $\sqrt{2}$ and equations involving natural number powers of 2. The first appear whenever we apply the Hadamard matrix, which, you may recall, has the form

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

Our `group_radicals` tactic groups together all of the $\sqrt{2}$ terms in our formula and simplifies what it can. This often results in a substantially more manageable goal state.

The other expression, 2^n for natural numbers 2 and n , is even more common in our development. To understand why, recall that an n -qubit system corresponds to a density matrix with dimensions $2^n \times 2^n$. When we take the Kronecker product of an $m \times n$ and $o \times p$ matrix, we get an $mo \times np$ matrix as the result. Hence, in the case of m - and n -qubit systems, we should obtain a matrix of height $2^m * 2^n$ or 2^{m+n} . Since multiplying A and B requires that the second dimension of A should match the first dimension of B , we need to reduce these dimensions to a normal form. Our `unify_pows_two` does exactly this through a series of rewriting rules.

Both `clra` and `Csimpl` are simple, unextendible tactics, so we also provide the user with `C_db`, a hints database of complex number equalities. `C_db` is a *rewriting* database, meaning that it is used to simplify the goal rather than to solve it. The

⁴Technically, this should be `lca` for “linear complex arithmetic,” but `clra` makes the connection to `lra` clear, as will our `mlra` tactic for matrices

idiom

`autorewrite with C_db`; `clra` is pretty common in the `QWIRE` development.

9.3 Linear Algebra

9.3.1 Matrix Properties

Automation is critical for handling matrices in Coq. Before we delve into our main tactics (`reduce_matrices` and `solve_matrices`) for handling equalities on matrices, we will look at some of the minor matrix tactics that play an important role in the `QWIRE` development.

Well-formedness As noted in Section 5.4, our matrices are simply functions of the form $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{C}$ with additional type information about their dimensions. We will often need to check a well-formedness condition that guarantees that a given matrix has all zeroes outside of its specified dimensions. We provide two approaches for solving this kind of problem. The `show_wf` tactic is designed for concrete matrices: It unfolds the definition of `WF_Matrix` and then destructs $x > m \vee y > n$. Consider the case where $x > m$. The tactic replaces x with $m + (x - m)$ and simplifies the goal. In the case of simple matrices that pattern match on x then y , this will immediately match the wild card case—the zero. In the $y > n$ case, we may have to destruct x up to m times first, but ultimately we will obtain $0 = 0$. In order to use `show_wf` on slightly more complex matrices involving sums, products or transposes, we call `cbv` to unfold these definitions and solve the final equality with `clra`.

Once our basic set of matrices is defined, we will generally create new matrices by composing existing ones. In the development, we show that all of our matrix operations preserve well-formedness, and we add those proofs to the database `wf_db`. We then use `auto with wf_db` to prove that matrix expressions are themselves well-formed.

Pure and Mixed States For quantum computation, it is often important to know that we are working with pure or mixed states (discussed in Section 6.1.3). We provide tactics `show_pure` and `show_mixed` to establish that a matrix possesses the desired property, with `show_mixed` including `show_pure` as a subroutine (since all pure states are mixed states). These mostly exploit existing lemmas about applying operations to pure states, though they often need to first show that dimensions line up. This can be accomplished using the `dim_solve` and `unify_dim_solve` tactics.

Lightweight Equalities For simple matrix equalities, we provide the tactic `mlra`, for *matrix linear real arithmetic*. The tactic begins by unfolding matrix definitions using the unfolding database `M_db`. It then applies `functional_extensionality` twice

and destructs the results using `destruct_m_eq`, allowing it to compare matrices at the element level. Hence, the goal

$$\begin{pmatrix} a_{0,0} & a_{0_1} \\ a_{1,0} & a_{1_1} \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0_1} \\ b_{1,0} & b_{1_1} \end{pmatrix}$$

becomes

$$\forall x, \forall y, \begin{pmatrix} a_{0,0} & a_{0_1} \\ a_{1,0} & a_{1_1} \end{pmatrix} (x, y) = \begin{pmatrix} b_{0,0} & b_{0_1} \\ b_{1,0} & b_{1_1} \end{pmatrix} (x, y)$$

which becomes the four goals

$$a_{0,0} = b_{0,0} \tag{9.3.1}$$

$$a_{0,1} = b_{0,1} \tag{9.3.2}$$

$$a_{1,0} = b_{1,0} \tag{9.3.3}$$

$$a_{1,1} = b_{1,1} \tag{9.3.4}$$

along with a small number of “ $0 = 0$ ” goals (due to the construction of our matrices), which are immediately discharged by `reflexivity`. Our `mlra` tactic then solves the four goals using our favorite tool for complex number equalities, `clra`.

`Msimp1` is the matrix counterpart to `Csimp1`. Unlike `Csimp1`, `Msimp1` has extensibility built in via the `M_db rewriting` database. (Coq allows us to use the same name for unfolding and rewriting databases.) `Msimp1` contains some additional code for rewriting expressions involving the Kronecker product or controlled matrices, since `autorewrite` can often struggle to unify these terms due to difficulty matching on the matrix dimensions. Otherwise, however, it behaves much like our other simplification tactics, in that it only uses simple equalities to rewrite a goal into a more manageable state.

9.3.2 Solving Matrix Equalities

So much for our lightweight tactics. Our powerful tactics for simplifying and solving matrix equalities are called `reduce_matrices` and `solve_matrix`, respectively. The two tactics rely on an overlapping set of sub-tactics, so we will treat the two of them together.

Let us begin by taking a bird’s-eye view of the problem of simplifying matrix expressions. Solving complicated matrix equalities is the rare area where computation by rewriting is substantially more efficient than simple evaluation, regardless of reduction strategy. To see why, consider the expression

$$A \times B \times C$$

where A has dimensions $m \times n$, and C has dimensions $o \times p$. Naively simplifying the two left-most matrices involves $m * o$ sums, each of vectors of length n , for a total

time of $m * o * n$. The resulting matrix has dimensions $m \times o$, so multiplying it by C takes an additional $m * p * o$ steps, for a total of $mon + mpo = mo(n + p)$. By contrast, if we started with the right-most matrices, it would take $np(m + o)$ time. Imagining that $m = o = 32$ and $n = p = 8$, the first approach takes 2^{14} steps, and the second takes 2^{12} . And these effects compound as we multiply more matrices. Hence, we take an approach where, if m or o is the smallest of the four numbers, we associate left; otherwise, we associate right. (We could compute $mo(n + p)$ vs. $np(m + o)$ in the tactic language, but this turns out to be a good and cheap proxy.) The tactic `assoc_least` reassociates at all levels of a complex matrix expression, though we will begin computing with the innermost matrices.

Once we've identified the matrices A and B that we want to multiply first, how do we multiply them together while ignoring the rest of the equation? We found that the easiest approach is simply to construct an appropriately-sized matrix E full of `evars` and then to replace $A \times B$ with E . This allows us to focus on the separate goal of computing E and to simplify $A \times B$ as much as possible before unifying it with E . This work is all done by the `reduce_aux` tactic:

```
Ltac reduce_aux M :=
  match M with
  | ?A .+ ?B      => compound A; reduce_aux A
  | ?A .+ ?B      => compound B; reduce_aux B
  | ?A × ?B       => compound A; reduce_aux A
  | ?A × ?B       => compound B; reduce_aux B
  | @Mmult ?m ?n ?o ?A ?B      => let E := evar_matrix m o in
                                   replace M with E;
                                   [| crunch_matrix ]
  | @Mplus ?m ?n ?A ?B         => let E := evar_matrix m n in
                                   replace M with E;
                                   [| crunch_matrix ]

  end.
```

The `compound A` tactic simply checks that A consists of the sum or matrix product of two matrices, in which case `reduce_aux` is recursively called on A . Otherwise, if `reduce_aux` is called on two irreducible matrices, it constructs a matrix full of `evars` using `evar_matrix`. Note that in this case, the match clause expands $?A \times ?B$ to `@Mmult ?m ?n ?o ?A ?B`. This gives us the dimensions of the matrices and, therefore, of the desired `evar` matrix. Finally, we call another tactic to unify $?A \times ?B$ with E .

The `crunch_matrix` tactic handles unification. It works much like `mlra`, except that when called as a subroutine of `reduce_matrices` and `solve_matrix`, it's guaranteed to work because we can unify any expression with an `evar`. Hence, it mostly has to make decisions regarding the degree of simplification required before unification. In practice, we only use `simpl` and `Csimpl` on the elements of the matrix, as heavy rewriting would dramatically slow down the tactic. The goal is solved by `try reflexivity`: The `try` is there so that `crunch_matrix` makes progress even if one side doesn't consist of `evars`.

This basically describes the full process of `reduce_matrices`. It looks for matrices

in the goal and calls `assoc_least` and `reduce_matrix_aux` on them. It keeps calling `reduce_matrix_aux` until no more matrices can be combined. Note that throughout this process, the relevant operations are matrix multiplication and addition. The Kronecker product and transpose are both easy enough to compute using our representation that $(A \otimes B)^\dagger$ is treated like an irreducible matrix in this process.

The `solve_matrix` tactic is essentially `reduce_matrices` with an extra step. Once it has reduced both sides of a matrix equality to atomic matrices, it makes a final call to `solve_matrix`. This tactic will rarely discharge the $m * n$ goals produced, as it only does basic simplifications and `try reflexivity`. Our `solve_matrix` will then attempt to unify the complex number equalities by rewriting with `C_db` and calling `try clra`. If `clra` fails to discharge all of the goals, they will be left for the user to complete (or, frequently, to revise their theorem).

9.4 Denoting Circuits

The last bit of relevant automation relates to the denotation of circuits. As we saw in Section 5.5, transforming circuits to super-operators isn't a simple computation, and Coq needs some help to get there. We provide two tactics for transforming circuits to their denotations, corresponding to the two representations of quantum states from Chapter 2.

The first tactic `matrix_denote` works pretty much as you might expect. It repeatedly unfolds terms from the database `den_db` and simplifies the goal. This leaves the user with a fairly complicated matrix expression, which she may either call `solve_matrix` on or attempt to make tractable in some other way.

However, as we saw in Section 2.4, there is a far more economical representation for circuits in terms of vectors, provided that the circuits don't contain measurement. Instead of providing an alternative denotation for measurement-free circuits, we take advantage of the fact that the density matrix representation of a pure state $|\phi\rangle$ is simply $|\phi\rangle\langle\phi|$. Hence, we can *fold* the matrix denotation of a circuit into the vector denotation by means of a few lemmas. The `vector_denote` tactic does exactly that, unfolding only what needs to be unfolded to produce a goal of shape `super U (|\phi\rangle\langle\phi|) = |\psi\rangle\langle\psi|`, where `U` may correspond to a sequence of unitary operations, and then applying a lemma to reduce this to `U |\phi\rangle = |\psi\rangle`. Such a goal can be efficiently discharged by `solve_matrix`, which will associate all the matrices to the right, so we're always multiplying by an $2^n \times 1$ matrix for some n .

9.5 Tactics Reference

We conclude with a table of our tactics, excluding some minor sub-tactics, along with brief descriptions for easy reference. We also include a list of databases for use in unfolding, rewriting, and solving goals.

Tactic	Effect
General Use Tactics	
<code>bdestruct t</code>	Destructs a boolean relation on natural numbers <code>t</code> and inserts the corresponding propositional relation into the context (from Appel (2018))
<code>bdestructΩ t</code>	Calls <code>bdestruct t</code> then tries to solve the goal with <code>omega</code>
<code>simpl_rewrite H</code>	Simplifies <code>H</code> and uses it to rewrite in the goal
<code>simpl_rewrite' H</code>	Like <code>simpl_rewrite H</code> but for rewriting right-to-left
<code>unify_pows_two</code>	Simplifies natural number expressions involving powers of two
Real and Complex Number Arithmetic	
<code>clra</code>	Linear real arithmetic (<code>lra</code>) extended to complex numbers
<code>nonzero</code>	Solves goals of the form $c \neq 0$ (for complex <code>c</code>)
<code>group_radicals</code>	Simplifies real and complex expressions involving square roots of two
<code>Rsimp1</code>	Simplifies real expressions
<code>Csimp1</code>	Simplifies complex expressions
<code>Rsolve</code>	Terminating tactic calling <code>Rsimp1</code> and <code>group_radicals</code>
<code>Csolve</code>	Terminating tactic calling <code>Rsolvea</code>
Matrix Tactics	
<code>prep_matrix_equality</code>	Applies <code>functional_extensionality</code> twice to show matrix equality
<code>show_wf</code>	Manually shows that a matrix is well-formed
<code>show_pure</code>	Shows that a matrix corresponds to a pure state
<code>show_mixed</code>	Shows that a matrix corresponds to a pure state
<code>dim_solve</code>	Shows that the dimensions of two matrices are equal
<code>unify_dim_solve</code>	Solves equations of the form $A \otimes B = A \otimes B$ where the implicit dimensions are different
<code>Msimp1</code>	Uses matrix equalities to simplify matrix expressions
<code>destruct_m_eq</code>	Repeatedly destructs natural numbers in match statements
<code>mlra</code>	Uses <code>destruct_m_eq</code> to replace a matrix equality with $m * n$ complex number equalities; attempts to solve these using <code>clra</code>

Tactic	Effect
<code>evar_matrix m n</code>	Creates an $m \times n$ matrix of evars
<code>assoc_least</code>	Reassociates matrices so that the smallest dimensions are multiplied first
<code>crunch_matrix</code>	Attempts to unify all the elements in two matrices
<code>reduce_matrix</code>	Does a single reduction of $A \times B$ or $A + B$, where A and B are irreducible matrices in the goal
<code>reduce_matrices</code>	Reduces all of the matrices in the goal
<code>solve_matrix</code>	Solves matrix equalities using <code>assoc_least</code> , <code>reduce_matrices</code> and <code>crunch_matrix</code>
Typechecking	
<code>invert_patterns</code>	Reduces all patterns in the hypotheses to <code>bit v</code> , <code>qubit v</code> or <code>unit</code>
<code>monoid</code>	Solves monoidal equalities with at most one evar
<code>validate</code>	Shows that a context is valid
<code>solve_merge</code>	Solves goals of the form $\Gamma == \Gamma_1 \cdot \Gamma_2$ using <code>monoid</code> and <code>validate</code>
<code>simple_typing</code>	Typechecks a circuit using only existing lemmas
<code>type_check_once</code>	A single iteration of the typechecking tactic
<code>type_check</code>	Typechecks all of the goals until they are solved
Circuit Denotation	
<code>matrix_denote</code>	Turns a circuit without into its vector denotation via unfolding and simplification
<code>vector_denote</code>	Turns a unitary circuit without measurement into its vector denotation
<code>case_safe</code>	Replaces the denotation of an ancilla-free circuit with its “safe” denotation
<code>case_unsafe</code>	Replaces the denotation of an ancilla-free circuit with its “safe” denotation
<code>rewrite_inPar</code>	Rewrites using the admitted fact <code>inPar_correct</code>
<code>compose_denotations</code>	Rewrites using the admitted fact <code>denote_compose</code>

Table 9.1: A summary of *QWIRE* tactics

Database	Description
<code>R_db</code>	Real number rewriting

<code>C_db</code>	Complex number rewriting
<code>C_db_light</code>	Lightweight complex rewriting
<code>Cdist_db</code>	Normalizing complex expressions
<code>M_db</code>	Matrix rewriting
<code>proof_db</code>	Circuit denotation rewriting
<code>monad_db</code>	Monad unfolding
<code>M_db</code>	Matrix unfolding
<code>den_db</code>	Circuit denotation unfolding
<code>vector_den_db</code>	Circuit denotation unfolding (vector representation)
<code>wf_db</code>	Matrix well-formedness proofs
<code>typed_db</code>	Typing proofs

Table 9.2: A summary of \mathcal{Q} WIRE databases

Chapter 10

Details: *Q*wire Within

The concept of *literate programming* (Knuth, 1984; Ramsey, 1994) has gained wide currency among the formal verification community. A literate program consists of a program and its exposition interleaved in a single file, allowing for clearer code and readable documentation. The popular “Software Foundations” series (Pierce et al., 2018; Appel, 2018) and “Certified Programming with Dependent Types” (Chlipala, 2013) are both literate programs written in Coq, typeset using Coq’s literate programming utility, Coqdoc. One could easily argue that this dissertation ought to be a literate program: It describes a tool, *Q*WIRE, that is written entirely inside the Coq proof assistant. Another argument would suggest that this is not a dissertation: It is a pointer to a dissertation hosted on Github. Constrained as we are by formatting requirements, we have provided a text that conveys the *ideas* of *Q*WIRE. However, this is not enough.

In this chapter, we focus on the details of *Q*WIRE. We describe the development itself in terms of its structure, the files that make up the structure, and the definitions and lemmas that give it substance. We also discuss the assumptions that underlie the *Q*WIRE development.

10.1 An outline of *Q*wire

The complete *Q*WIRE development consists of

- 25 Coq files,
- 20242 lines of code,
- 557 definitions,
- 786 theorems and lemmas, and
- 106 custom tactics.

File	LOC	Definitions	Fixpoints	Inductives	Ltacs
Prelim.v	397	3	8	0	7
Monad.v	409	28	4	0	4
Monoid.v	641	1	8	1	21
Complex.v	677	16	1	0	7
Matrix.v	1484	25	5	0	23
Quantum.v	1261	41	5	1	1
Contexts.v	1639	14	14	12	10
HOASCircuits.v	113	3	1	3	0
TypeChecking.v	415	3	2	0	6
DBCircuits.v	503	14	11	3	0
Denotation.v	4380	35	13	4	3
HOASLib.v	302	20	6	0	0
SemanticLib.v	110	0	0	0	0
HOASExamples.v	510	37	7	0	0
Composition.v	200	0	0	0	0
Ancilla.v	416	4	0	3	2
Symmetric.v	1536	25	3	2	1
Oracles.v	1376	1	18	2	14
Deutsch.v	232	2	0	0	3
Equations.v	458	20	0	0	0
HOASProofs.v	772	19	1	0	2
Arithmetic.v	1677	27	13	0	1
QASM.v	499	17	10	10	0
QASMPrinter.v	142	10	9	0	0
QASMEExamples.v	93	12	0	0	1
Totals	20242	377	139	41	106

Table 10.1: A brief summary of the \mathcal{Q} WIRE Coq development.

In definitions, we include code beginning with `Definition`, `Fixpoint`, or `Inductive`. By theorems and lemmas, we only mean Coq propositions that begin with `Theorem` or `Lemma` and end with `Qed` or `Defined`. We will delve into detail about our theorems and lemmas, as well as other kinds of statements, in the next section. We provide a fuller account of the QWIRE development in Table 10.1 and Table 10.2.

We can break down the twenty-four Coq files as follows¹:

- Preliminaries
 1. `Prelim.v` : A variety of general purpose definitions and tactics
 2. `Monad.v` : An implementation of some basic monads
 3. `Monoid.v` : A typeclass and solver for commutative monoids, modified from `LinearTypingContexts`
- Underlying mathematical libraries (Chapter 5)
 4. `Complex.v` : Complex number library, modified from `Coquelicot`
 5. `Matrix.v` : Matrix library
 6. `Quantum.v` : Defines unitary matrices and quantum operations
- Implementation of QWIRE (Chapter 5)
 7. `Contexts.v` : Defines wire types and typing contexts
 8. `HOASCircuits.v` : Defines QWIRE circuits using higher-order abstract syntax
 9. `TypeChecking.v` : Circuit notations and tactics for proving well-typedness
 10. `DBCircuits.v` : Compiling HOAS to De Bruijn style circuits
 11. `Denotation.v` : Defines the denotational semantics of QWIRE circuits and proves its (quantum mechanical) validity
 12. `HOASLib.v` : A library of basic circuits used in QWIRE programming
 13. `SemanticLib.v` : Proves the semantic properties of HOASLib circuits
 14. `HOASExamples.v` : Additional examples of HOAS circuits
- Verification of QWIRE circuits (Chapter 7)
 15. `HOASProofs.v` : General proofs about quantum circuits
 16. `Equations.v` : Equalities on small circuits
 17. `Deutsch.v` : Alternative approaches to verifying Deutsch’s algorithm

¹A modified version of this text also appears in the Github repository at <https://github.com/inQWIRE/QWIRE>

- Compositionality (Section 6.2)
 1. `Composition.v` : Defines the compositionality facts used in Chapter 8
- Reversible Circuits (Chapter 8)
 2. `Ancilla.v` : Defines the correctness of circuits using ancilla assertions
 3. `Symmetric.v` : Syntactic conditions for guaranteeing the validity of assertions
 4. `Oracles.v` : Compilation of boolean expressions to QWIRE circuits
 5. `Arithmetic.v` : Verification of a quantum adder
- Compilation to QASM (Section 11.2)
 6. `QASM.v` : Compilation from QWIRE to QASM
 7. `QASMPrinter.v` : A printer for compiled circuits, for execution on a quantum computer/simulator
 8. `QASMEexamples.v` : Examples of circuit compilation

We have annotated the divisions with the chapters that focus on them, where appropriate, though this characterization is fairly broad. `Typechecking.v`, for instance, is covered in substantial detail in Chapter 9, as is `Monoid.v`. We have also excluded some experimental files, including Dong-Ho Lee’s files for generating well-typed circuits, `Generator.v` and `DBGenerator.v` (see Section 11.2). We have included Lee’s compiler to QASM, though it is not our own work, which will also appear in Section 11.2.

10.2 Matters of Trust

The trusted computing base for QWIRE is quite small. Besides for the core of Coq itself, we include a few axioms from the Coq standard library. Our matrices are really functions which requires us to use the functional extensionality axiom from Coq’s standard library to prove that two matrices are equal:

```
Axiom functional_extensionality_dep :  $\forall$  {A} {B : A  $\rightarrow$  Type},
   $\forall$  (f g :  $\forall$  x : A, B x),
  ( $\forall$  x, f x = g x)  $\rightarrow$  f = g.
```

Furthermore, as noted in Section 5.4.1, Coq’s real numbers are axiomatized: 0 and 1 are simply given as parameters and the rules about addition, multiplication and other operations are given in terms of axioms. Fortunately, these axioms are very well studied and known to be consistent.

The QWIRE development itself adds three additional axioms. The first two are well established facts about linear algebra. `Minv_flip` asserts that the right inverse

File	Lemmas	Theorems	Facts	Propositions	Examples
Prelim.v	32	0	0	0	0
Monad.v	3	0	0	1	0
Monoid.v	18	0	0	0	10
Complex.v	84	0	0	0	0
Matrix.v	91	5	0	1	1
Quantum.v	84	0	0	2	2
Contexts.v	72	0	0	0	0
HOASCircuits.v	2	0	0	0	0
TypeChecking.v	4	0	0	0	1
DBCircuits.v	17	0	0	1	0
Denotation.v	113	2	2	19	0
HOASLib.v	22	0	0	0	0
SemanticLib.v	14	0	0	0	0
HOASExamples.v	38	0	0	4	1
Composition.v	1	1	2	0	0
Ancilla.v	8	0	2	2	0
Symmetric.v	45	3	18	1	0
Oracles.v	35	1	2	3	1
Deutsch.v	5	0	0	0	1
Equations.v	26	0	0	3	0
HOASProofs.v	22	0	0	8	2
Arithmetic.v	38	0	0	1	32
QASM.v	0	0	0	0	1
QASMPrinter.v	0	0	0	0	0
QASMEExamples.v	0	0	0	0	2
Totals	774	12	26	46	54

Table 10.2: A summary of the different kinds of statements in the \mathcal{Q} WIRE development.

of a square matrix is also its left inverse and `kron_assoc` says that the Kronecker product is associative. Proving these lemmas requires a substantial amount of linear algebra that we haven't yet formalized. Similarly, we have axiomatized the operator sum decomposition theorem,

```
Axiom operator_sum_decomposition :  $\forall$  {m n} (l : list (Matrix m n)),
  outer_sum l = 'I_n  $\leftrightarrow$  WF_Superoperator (operator_sum l).
```

whose proof requires more concepts from linear algebra than we were able to provide. See Section 6.1.4 for details.

Beyond these, we have a number statements beginning with `Fact`, to emphasize that these were admitted. All of these are used exclusively in the files on reversibility, plus `Deutsch.v` which depends on compositionality axioms and isn't discussed in this dissertation. We refer the reader to Table 8.1 for an synopsis of the admitted statements.

Finally, we include a number of conjectures, which begin with `Proposition`², and correspond to aborted claims that we would like to prove in the future.

We give a summary of the claims in `QWIRE` in Table 10.2. A complete list of theorems, lemmas, and facts in the `QWIRE` development is provided in Appendix C.

²`Conjecture` is unfortunately used by Coq as a synonym for `Axiom`.

Chapter 11

Open Wires and Loose Ends

The arguably grandiose title of this dissertation is “Formally Verified Quantum Programming”. Despite substantial recent interest, this area of research is in its infancy. In Chapter 1, we made an bold claim: Quantum programming demands formal verification, and therefore quantum programming languages must have well-defined semantics and support for verification. This thesis, along with the broader *QWIRE* project and the work that inspired it, only begins to tackle this issue. In this chapter, we will discuss the issues *QWIRE* still needs to address in order to serve both as a real-world quantum programming language and a robust, user-friendly verification tool.

This chapter will also seek to address a more interesting question, and hopefully one that excites the reader. What else can *QWIRE* do? To be precise, what challenges does quantum computing face, in both the short term and the long term, and how can *QWIRE* help address those challenges? In this thesis, we have depicted quantum computing as an exciting new technology that lies somewhere along the horizon. But as John Preskill pointed out in a recent survey (Preskill, 2018), we should be looking at multiple horizons, the first of which (Noisy Intermediate-Scale Quantum Computing, or NISQ) is years, not decades, away. We will try to address how *QWIRE* and formal verification can help us run the quantum programs of tomorrow, not just those of the far future.

11.1 Computing in *Qwire*

In the “future work” section of our 2017 QPL paper (Rand et al., 2017), we promised to move *QWIRE* to a more efficient backend, specifically referencing *QWIRE*’s linear algebra library. Aside from the linear algebra, we wanted to move away from our Coqelicot-based (Boldo et al., 2015) representation of complex numbers, which *QWIRE* still uses. Here we address the limitations of these libraries and the challenges of migrating away from them.

The Coqelicot complex number library, which we have modified and included in the *QWIRE* development, is based on the Coq standard library’s real numbers. Co-

quelicot’s \mathbb{C} is simply a pair of Coq \mathbb{R} s. Coq’s real numbers, in turn, are *axiomatized*: $0, 1, +, -, *, /$ and $<$ are simply declared as Coq parameters; they have no computational content. Likewise, the associativity, commutativity, and distributivity of $+$ and $*$ are declared as axioms, as are the remaining field laws.

From a computational standpoint, this is awful. It is never possible to reduce real number expressions using call-by-value or call-by-name reduction rules, because no definitions exist to reduce. In Coq’s defense, this limitation is inherent to the real numbers: $e + \pi$ cannot normalize to anything but $e + \pi$. The exact nature of our proofs prohibit using floating point numbers instead of reals, though once we account for error terms (see the next section), we may be able to loosen this restriction.

How can we improve QWIRE’s real number library to allow for computation? Potentially, we could restrict ourselves to the algebraic numbers. Right now, only one of our unitary gates, the $R_\theta = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$ gate, allows us to multiply our states by non-algebraic numbers, and we typically instantiate θ with π/n for an integer n , yielding $\sqrt[n]{-1}$. We could then compute by grouping like terms. It’s not clear how difficult this would be to implement, and we are not aware of a library that does this.

The other bottleneck is our matrix library. which is quite slow. In fact, we have trouble accounting for quite how slow it is, struggling to multiply even 16×16 dimensional matrices. In our QPL paper, we suggested adopting the Mathematical Components library’s matrices (Mahboubi et al., 2016) for proofs, while reflecting into the Coq Effective Algebra library (Cano et al., 2016) for efficient computation. This ultimately raised several obstacles.

The Mathematical Components library, written in the SSReflect proof language and used in the formally verified proofs of the Four Color Theorem (Gonthier, 2008) and Feit-Thompson Theorem (Gonthier et al., 2013), uses dependently typed matrices. On the face of it, this is an excellent use-case for dependent types. Matrix multiplication is only defined on two matrices of dimensions $m \times n$ and $n \times o$, respectively, and dependent types can enforce this property. Unfortunately, the Kronecker product, which is integral to quantum computation, multiplies the dimensions of its argument matrices. As soon as we start dealing with matrices of dimensions 2^m and 2^n , where neither m nor n is concrete, convincing the Coq typechecker that we are indeed multiplying compatible matrices becomes exceedingly difficult. This motivated our decision to use *phantom types* (Leijen and Meijer, 1999), lightweight types meant to guide development that are not enforced by the typechecker, in our matrices. We discussed this design choice in detail at the Fourth International Workshop on Coq for Programming Languages (Rand et al., 2018b).

CoqEAL has a number of advantages in terms of efficiently computing with large matrices, including an implementation of Strassen’s Algorithm (Strassen, 1969) for efficient multiplication, but a number of drawbacks as well. For one, it is not designed for proofs, but rather, for computation. That means we would have to convert some other matrix representation to CoqEAL matrices and only use them in the final stages of certain proofs involving concrete matrices. The current version of CoqEAL

also does not work with the Coq real numbers library and, hence, will not work with our complex numbers. These challenges could probably be overcome, but doing so would have limited payoff in the absence of the mathematical components library and computational complex numbers.

11.2 Connecting Qwire

So far in this thesis, we have gestured towards three efforts to extend QWIRE beyond the narrow world of the Coq proof assistant, all of which are currently being pursued by Dong-Ho Lee. Here we will flesh these out somewhat, noting that this author merely advised on these projects.

Lee’s main contribution to QWIRE is a compiler to the OpenQASM “quantum assembly” language (Cross et al., 2017) and, specifically, to the subset of OpenQASM that can run on IBM’s online quantum computer (IBM, 2017). The file QASM.v provides a Coq datatype for OpenQASM circuits and compiles QWIRE’s de Bruijn circuits to this representation. In turn, QASMPrinter.v prints these structures to a form that can be run on IBM’s quantum cloud. QASMEexamples.v provides a few examples of QWIRE circuits and their corresponding OpenQASM programs.

Lee’s ongoing work seeks to extract QWIRE to Haskell and OCaml for the sake of efficient simulation and random testing. The first aim is natural: We want to be able to efficiently simulate QWIRE circuits, thereby making use of the *run* semantics proposed in Chapter 4. More generally, a programming language ought to allow for execution of its programs, even if it was written for computers that do not yet exist. For the second aim, we have argued in this thesis that unit testing is not likely to work for quantum programs, and this is doubly true for random testing. However, random testing can assist in *proving circuit correctness* and other aspects of the formalization project. Inspired by the QuickChick Coq plugin (Lampropoulos, 2018; Lampropoulos and Pierce, 2018), we would like to use random testing to find small counterexamples before we try to prove an incorrect specification for a circuit or family of circuits. Towards that end, the Generator.v file seeks to generate well-typed de Bruijn circuits and extract them to OCaml for testing. Hopefully, this will ease the cumbersome verification process going forward.

11.3 The Future of Qwire

Research into QWIRE and research using QWIRE can extend into many domains, from investigating and implementing quantum abstractions, to testing quantum circuits in QWIRE, to developing an even richer metatheory and categorical semantics. In this section, however, we will focus on three near-term goals for QWIRE, which mostly pertain to near-term quantum computing.

	Depth	Gates	Clean Qubits	Total Qubits
Shor (1994)	$\Theta(nM(n))$	$\Theta(nM(n))$	$\Theta(n)$	$\Theta(n)$
Beckman et al. (1996)	$\Theta(n^3)$	$\Theta(n^3)$	$5n + 1$	$5n + 1$
Vedral et al. (1996)	$\Theta(n^3)$	$\Theta(n^3)$	$4n + 3$	$4n + 3$
Beauregard (2002)	$\Theta(n^3 \lg \frac{1}{\varepsilon})$	$\Theta(n^3 \lg \frac{n}{\varepsilon} \lg \frac{1}{\varepsilon})$	$2n + 3$	$2n + 3$
Takahashi et al. (2006)	$\Theta(n^3 \lg \frac{1}{\varepsilon})$	$\Theta(n^3 \lg \frac{n}{\varepsilon} \lg \frac{1}{\varepsilon})$	$2n + 2$	$2n + 2$
Zalka (2006)	$\Theta(n^3 \lg \frac{1}{\varepsilon})$	$\Theta(n^3 \lg \frac{n}{\varepsilon} \lg \frac{1}{\varepsilon})$	$1.5n + O(1)$	$1.5n + O(1)$
Häner et al. (2016)	$\Theta(n^3)$	$\Theta(n^3 \lg n)$	$2n + 2$	$2n + 2$
Gidney (2017)	$\Theta(n^3)$	$\Theta(n^3 \lg n)$	$n + 2$	$2n + 1$

Table 11.1: Space-efficient constructions of Shor’s algorithm over time. $M(n)$ is the classical time-complexity of multiplication and ε is the maximum error when synthesizing the circuit out of a fixed set of universal gates. Reproduced with permission from [Gidney \(2017\)](#).

11.3.1 Verified Optimization and Compilation

As we discussed in Chapter 7, verified optimization is a key next step for \mathcal{Q} WIRE. Before we delve into our optimization goals, it is worth discussing why this is an important problem.

While large corporations like Google and IBM seek to accelerate progress in quantum computing by building larger and more powerful quantum computers, a similar effort has gone into bringing down the cost of the quantum algorithms themselves. Among the major measures of cost are circuit depth, gate count, and the number of qubits required.

For instance, Figure 26 of [Gidney \(2017\)](#) (reproduced in Table 11.1) shows the improvement in Shor’s algorithm over the course of twenty-three years and eight papers. Shor’s initial approach (1994) only showed that we could factor numbers using $\Theta(n)$ qubits, where n is the length of the number to be factored. Subsequent work by [Beckman et al. \(1996\)](#), showed that the algorithm could be implemented using $5n$ qubits, and [Häner et al. \(2016\)](#) and [Gidney \(2017\)](#) reduced that to $2n$ qubits, with the latter showing that half of these could be *dirty* qubits in an unknown state. From [Beckman et al.](#) to [Gidney](#), the depth and gate requirements remained almost the same, increasing from $\Theta(n^3)$ to $\Theta(n^3 \log(n))$ in the latter case. Shor’s original algorithm is in principle more efficient on both metrics but impossible to implement using a reasonable number of qubits.

Other recent work has focused on efficiently approximating common quantum gates using a standard gate set ([Ross and Selinger, 2014](#); [Kliuchnikov et al., 2016](#)) and efficiently synthesizing various kinds of circuit ([Bocharov et al., 2015a,b](#)). [Saeedi and Markov \(2013\)](#) survey the literature on this topic.

Our goal is to use the circuit equalities discussed in Section 7.4, along with additional rules drawn from [Nam et al. \(2018\)](#), [Fagan and Duncan \(2018\)](#), and other sources in order to optimize circuits before executing them. In particular, we would

like to optimize these circuits subject to constraints. These constraints may take a variety of forms, from limiting the available gate set to restricting the interactivity of qubits. For instance, Kutin (2006) provides a version of Shor’s algorithm for a computer on which only adjacent qubits may interact with one another. This will require us to develop a means of specifying constraints in \mathcal{Q} WIRE and to prove that our optimizations not only preserve a circuit semantics, but also satisfy these constraints.

Once we have optimized our circuits, we would like to extend Dong-Ho Lee’s compiler to convert them to OpenQASM (Cross et al., 2017) or QUIL (Smith et al., 2016). The translation should also be bidirectional: We should be able to convert OpenQASM or QUIL programs to \mathcal{Q} WIRE to typecheck, optimize, and verify them. This would greatly increase the practical utility of \mathcal{Q} WIRE in the short term.

11.3.2 Error Awareness and Error Correction

The largest challenge facing quantum computing is the problem of errors: Errors creep into quantum computation not only when gates are applied to qubits, but even as time passes (through quantum decoherence). This gave rise to the study of quantum error correction (Preskill, 1998), which seeks to address the problems by using collections of qubits called *logical qubits* that are easily corrected. A variety of *threshold theorems* (Aharonov and Ben-Or, 1997; Kitaev, 1997; Gottesman, 1997) showed, under a variety of conditions, that arbitrarily large quantum computations could be performed, provided that the error rates on gates are kept below a certain threshold. Unfortunately, these error-correcting codes are expensive. Fowler et al. (2012) estimate that we need over 3000 physical qubits per logical qubit, which will prove unfeasible in the near term.

We have two goals for \mathcal{Q} WIRE in this area. The first is to extend \mathcal{Q} WIRE’s semantics to be *error-aware*. There are two main approaches to this problem: Extend the semantics to produce density matrices with error terms, or externally compute error terms for the entire circuit. This will help us quantify the errors in individual algorithms. We will be able to bound the probability of failure for each possible failure mode. This will allow us to identify which algorithms can be run without error correction and attempt to write and verify quantum algorithms for error prone quantum computers.

Once we have this model, we can start to write error correcting codes in \mathcal{Q} WIRE and verify that they work as specified. Specifically, we will be able to analyze the error rates of corrected circuits under a variety of assumptions and in different hardware models. This should lend some assurance to the error correcting codes currently proposed and perhaps help us come up with new error correction schemes.

11.3.3 Verified Algorithms and Cryptography

Of course, \mathcal{Q} WIRE shouldn’t be entirely focused on near term quantum computation. Many of the longer term possibilities for quantum computing are among the most

exciting. In principle, \mathcal{Q} WIRE should be able to verify a broad range of quantum programs, from Shor (1994) and Grover’s (1996) algorithms to quantum random walks (Aharonov et al., 2001; Childs et al., 2003) and quantum simulation of physical systems (Georgescu et al., 2014).

One of the most interesting problems for \mathcal{Q} WIRE is to verify quantum cryptographic protocols. Unruh’s Quantum Relational Hoare Logic (2018) seeks to form the foundation for a tool to verify quantum cryptographic protocols, modeled on the EasyCrypt tool (Barthe et al., 2011, 2012) for non-quantum cryptography. While such deductive systems are useful, they themselves often rest on weak foundations. We could attempt to prove the security of a cryptographic protocol by programming it in \mathcal{Q} WIRE and proving it directly or, alternatively, by developing a logic and proving it sound with respect to \mathcal{Q} WIRE’s semantics. This would provide a strong foundation for reasoning about quantum cryptographic systems.

In a field as young as quantum computing, we cannot predict what researchers will want to verify next year, let alone once we have scalable quantum computers to experiment with and program. But we do know that there is a great deal to verify. And we are confident that \mathcal{Q} WIRE and tools like it will play an increasingly important role as this field develops and gives us new protocols to verify and challenges to overcome.

Appendix A

Solutions to Exercises

Exercise 1. Show that H is its own inverse.

Solution. Let us first consider H applied to the basis states. We have already shown that $H(H|0\rangle) = |0\rangle$. The proof that $H(H|1\rangle) = |1\rangle$ follows similarly:

$$\begin{aligned} H(H|1\rangle) &= H\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) \\ &= \frac{1}{\sqrt{2}}H|0\rangle - \frac{1}{\sqrt{2}}H|1\rangle \\ &= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) - \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) \\ &= \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle - \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle \\ &= \frac{1}{2}|1\rangle + \frac{1}{2}|1\rangle \\ &= |1\rangle \end{aligned}$$

For any non-basis state $\alpha|0\rangle + \beta|1\rangle$, we have

$$H(H(\alpha|0\rangle + \beta|1\rangle)) = \alpha H(H|0\rangle) + \beta H(H|1\rangle) = \alpha|0\rangle + \beta|1\rangle$$

since unitaries distribute over superposition, $H(H|0\rangle) = |0\rangle$, and $H(H|1\rangle) = |1\rangle$. ■

What if we want to measure one qubit in a multiple qubit system? Let $\sum_i \alpha_i |i\rangle$ represent the part of the state in which the qubit to be measured is $|0\rangle$ and $\sum_j \beta_j |j\rangle$ represent the part in which the qubit is $|1\rangle$. Then the probability of measuring $|0\rangle$ is $\sum_i |\alpha_i|^2$, returning the state

$$\frac{1}{\sqrt{\sum_i |\alpha_i|^2}} \sum_i \alpha_i |i\rangle$$

and similarly for $|1\rangle$. The scaling factor on the left renormalizes the quantum state so that the squares of the amplitudes still add up to 1.

Exercise 2. Now try measuring the second qubit in both of these cases. Verify that the distribution of results is the same as if we had measured the whole system at once.

Solution. Let us begin by simplifying the expression for the case where we measured a $|0\rangle$:

$$\sqrt{\frac{3}{2}} \left(\frac{1}{3} |00\rangle + \frac{2+i}{3} |01\rangle \right) = \frac{1}{\sqrt{6}} |00\rangle + \frac{2+i}{\sqrt{6}} |01\rangle$$

The probability of measuring $|00\rangle$ here is $\left| \frac{1}{\sqrt{6}} \right|^2 = \frac{1}{6}$ and the probability of measuring $|01\rangle$ is $\left| \frac{2+i}{\sqrt{6}} \right|^2 = \frac{4+1}{6} = \frac{5}{6}$. We can scale these by the $\frac{2}{3}$ probability of having measured the first qubit as $|0\rangle$ to get total probabilities of $\frac{1}{9}$ and $\frac{5}{9}$, respectively.

In the case where we measured the first qubit as $|1\rangle$, we are guaranteed to measure $|11\rangle$ and so the total probability of measuring $|11\rangle$ is $\frac{1}{3}$.

Now, if we had initially measured the entire state we would have gotten

$$\text{meas} \left(\frac{1}{3} |00\rangle + \frac{2+i}{3} |01\rangle + \frac{1}{\sqrt{3}} |11\rangle \right) = \begin{cases} |00\rangle & \text{with probability } \left| \frac{1}{3} \right|^2 = \frac{1}{9} \\ |01\rangle & \text{with probability } \left| \frac{2+i}{3} \right|^2 = \frac{5}{9} \\ |11\rangle & \text{with probability } \left| \frac{1}{\sqrt{3}} \right|^2 = \frac{1}{3} \end{cases}$$

as expected. ■

Exercise 3. Verify that after measuring a qubit, the norm of the quantum state is still one.

Solution. The sum of squares after measuring $|i\rangle$ is

$$\left| \frac{1}{\sqrt{p_i}} \right|^2 \sum_i |\alpha_i|^2 = \frac{\sum_i |\alpha_i|^2}{\sum_i |\alpha_i|^2} = 1$$

.

Exercise 4. Write $(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$ as a vector, first by taking the Kronecker product directly and then by simplifying the expression and transforming it into vector notation. Confirm that both results are equal.

Solution. We can write the two components as the vectors $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ and $\begin{pmatrix} \gamma \\ \delta \end{pmatrix}$. Taking the

Kronecker product we get

$$\begin{pmatrix} \alpha \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \\ \beta \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}.$$

Alternatively, we can simplify the expression first using left and right distributivity:

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

and convert the result to matrix form:

$$\begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}$$

■

Exercise 5. Note that H , X , Z and $CNOT$ are all their own adjoints. Verify that these matrices are unitary.

Solution.

$$H: \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \right)^2 = \frac{1}{2} \begin{pmatrix} 1+1 & 1+-1 \\ 1+-1 & 1+--1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = I_2$$

$$X: \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 0+1 & 0+0 \\ 0+0 & 1+0 \end{pmatrix} = I_2$$

$$Z: \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}^2 = \begin{pmatrix} 1+0 & 0+0 \\ 0+0 & 0+--1 \end{pmatrix} = I_2$$

$$CNOT = \left(\frac{I_2 \mid 0}{0 \mid X} \right):$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 1+0+0+0 & 0+0+0+0 & 0+0+0+0 & 0+0+0+0 \\ 0+0+0+0 & 0+1+0+0 & 0+0+0+0 & 0+0+0+0 \\ 0+0+0+0 & 0+0+0+0 & 0+0+0+1 & 0+0+0+0 \\ 0+0+0+0 & 0+0+0+0 & 0+0+0+0 & 0+0+1+0 \end{pmatrix} = I_4$$

■

Exercise 6. Show that H , X , Z , and $CNOT$ have the behavior described in the previous section. That is, show that when applied to basis vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ they produce the claimed output.

Solution.

$$H: \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$X: \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$Z: \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

$CNOT$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

■

Exercise 7. Prove that for any pure state ρ in density matrix form, $\rho^2 = \rho$.

Solution. Since ρ represents a pure state, $\rho = |\phi\rangle\langle\phi|$ for some unit-vector $|\phi\rangle$.

Hence ρ^2 becomes $|\phi\rangle\langle\phi||\phi\rangle\langle\phi|$.

As we showed earlier, for any quantum state $|\psi\rangle$, $\langle\psi||\psi\rangle$ is the one element identity matrix (this is equivalent to the norm being 1). Hence, by multiplying the innermost matrices together, we obtain $|\phi\rangle I_1 \langle\phi| = |\phi\rangle\langle\phi| = \rho$.

■

Exercise 8. Show that for a pure state in density matrix form, the i^{th} element along the diagonal is the probability of measuring $|i\rangle$.

Solution. As we noted, a pure state in density matrix form can be written out as $|\phi\rangle\langle\phi|$. If $|\phi\rangle$ consists of the elements $\alpha_0, \alpha_1, \dots, \alpha_n$, then this produces the following matrix:

$$\begin{pmatrix} \alpha_0\overline{\alpha_0} & \alpha_0\overline{\alpha_1} & \dots & \alpha_0\overline{\alpha_n} \\ \alpha_1\overline{\alpha_0} & \alpha_1\overline{\alpha_1} & \dots & \alpha_1\overline{\alpha_n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_n\overline{\alpha_0} & \alpha_n\overline{\alpha_1} & \dots & \alpha_n\overline{\alpha_n} \end{pmatrix}$$

Thus, the k^{th} element along the diagonal will be $\alpha_k\overline{\alpha_k}$. It remains to show that every $\alpha_k\overline{\alpha_k} = |\alpha|^2$ (the probability of measuring $|k\rangle$).

Let us expand α_k out as $a + bi$.

$$(a + bi)(\overline{a + bi}) = (a + bi)(a - bi) = a^2 + b^2$$

. Similarly

$$|a + bi|^2 = \sqrt{a^2 + b^2}^2 = a^2 + b^2$$

■

Appendix B

Qwire in Theory: Proofs

B.1 Type safety and normalization

Theorem 5 (Preservation). *Suppose \longrightarrow_H satisfies preservation.*

1. *If $\vdash t:A$ and $t \longrightarrow t'$, then $\vdash t':A$.*
2. *If $;\mathcal{Q} \vdash C:W$ and $C \Longrightarrow C'$, then $;\mathcal{Q} \vdash C':W$.*

Proof.

1. If t steps via \longrightarrow_H then the result is immediate by the assumption that \longrightarrow_H satisfies preservation. Otherwise, suppose $t \longrightarrow_b t'$. It must be the case that $A = \mathbf{Box} W_1 W_2$ and $t = \mathbf{box} p \Rightarrow C$ where $\Omega \Rightarrow p:W_1$ and $;\Omega \vdash C:W_2$. If t steps via the structural rule with $C \Longrightarrow C'$, then $t' = \mathbf{box} p \Rightarrow C'$, and by the inductive hypothesis, $;\Omega \vdash C':W_2$ and so $\cdot \vdash \mathbf{box} p \Rightarrow C':\mathbf{Box} W_1 W_2$.

If t steps instead by an η rule, then $t' = \mathbf{box} p' \Rightarrow C \{p'/p\}$ where p' is concrete for W_1 . By Lemma 5 there is some \mathcal{Q} such that $\mathcal{Q} \Rightarrow p':W_1$, so by the substitution lemma (Lemma 4), we have $;\mathcal{Q} \vdash C \{p'/p\}:W_2$, and thus $\cdot \vdash t':\mathbf{Box}(W_1, W_2)$.

2. By induction on $C \Longrightarrow C'$.
 - (a) If $C = \mathbf{unbox} t p$ then we have

$$\cdot \vdash t:\mathbf{Box} W_1 W \quad \text{and} \quad \mathcal{Q} \Rightarrow p:W_1.$$

If C steps by a structural rule with $t \longrightarrow t'$, then by the inductive hypothesis we have $\cdot \vdash t':\mathbf{Box} W_1 W$, and so $;\mathcal{Q} \vdash \mathbf{unbox} t' p:W$. If it steps via the β rule, then $t = \mathbf{box} p' \Rightarrow N$, and so by inversion we know there is some $\mathcal{Q}' \Rightarrow p':W_1$ such that $;\mathcal{Q}' \vdash N:W_2$. By the substitution lemma (Lemma 4), we have that $;\mathcal{Q} \vdash N \{p/p'\}:W$ as expected.

(b) Suppose C is $p_2 \leftarrow \text{gate } g \ p_1; C_0$, where $\mathcal{Q} = \mathcal{Q}_1, \mathcal{Q}_0$ and

$$\mathcal{Q}_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad ; \Omega_2, \mathcal{Q}_0 \vdash C_0 : W.$$

If C steps via a structural rule on C_0 , the result is straightforward from the induction hypothesis. Otherwise, it steps via an η -expansion:

$$p_2 \leftarrow \text{gate } g \ p_1; C_0 \Longrightarrow p'_2 \leftarrow \text{gate } g \ p_1; C_0 \{p'_2/p_2\}$$

where $\mathcal{Q}_2 \Rightarrow p'_2 : W_1$. By Lemma 4 we know $;\mathcal{Q}_2, \mathcal{Q} \vdash C_0 \{p'_2/p_2\} : W$, and so $;\mathcal{Q}_1, \mathcal{Q} \vdash p'_2 \leftarrow \text{gate } g \ p_2; C_0 \{p'_2/p_2\} : W$.

(c) Finally, suppose $C = p \leftarrow C_1; C_2$, where $\mathcal{Q} = \mathcal{Q}_1, \mathcal{Q}_2$ and

$$;\mathcal{Q}_1 \vdash C_1 : W' \quad \Omega \Rightarrow p : W' \quad ; \Omega, \mathcal{Q}_2 \vdash C_2 : W$$

If C steps via a structural rule, the result is immediate. If it steps via a β -rule, then $C_1 = \text{output } p'$, and by inversion, $\mathcal{Q}_1 \Rightarrow p' : W$. By Lemma 4, we have $;\mathcal{Q}_1, \mathcal{Q}_2 \vdash C' \{p'/p\} : W'$.

If $C_1 = p_2 \leftarrow \text{gate } g \ p_1; C_0$ such that

$$p \leftarrow C_1; C_2 \Longrightarrow p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow C_0; C_2$$

by a commuting conversion, then by inversion we have $\mathcal{Q}_1 = \mathcal{Q}'_1, \mathcal{Q}_0$ where $g \in \mathcal{G}(W_1, W_2)$, $\mathcal{Q}'_1 \Rightarrow p_1 : W_1$, $\Omega'_2 \Rightarrow p_2 : W_2$, and $;\Omega'_2, \mathcal{Q}_0 \vdash C_0 : W'$. Then $;\Omega'_2, \mathcal{Q}_0, \mathcal{Q}_2 \vdash p \leftarrow C_0; C_2 : W$ and so

$$;\mathcal{Q}'_1, \mathcal{Q}_0, \mathcal{Q}_2 \vdash p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow C_0; C_2 : W.$$

If $C_1 = x \leftarrow \text{lift } p'; C_0$ such that

$$p \leftarrow C_1; C_2 \Longrightarrow x \leftarrow \text{lift } p'; p \leftarrow C_0; C_2$$

by a commuting conversion, then by inversion we have $\mathcal{Q}_1 = \mathcal{Q}_0, \mathcal{Q}'$ such that $\mathcal{Q}_0 \Rightarrow p' : W_0$ and $x : |W_0|; \mathcal{Q}' \vdash C_0 : W'$. In that case, $x : |W_0|; \mathcal{Q}', \mathcal{Q}_2 \vdash p \leftarrow C_0; C_2 : W$ and so $;\mathcal{Q}_0, \mathcal{Q}', \mathcal{Q}_2 \vdash x \leftarrow \text{lift } p'; p \leftarrow C_0; C_2 : W$.

□

Theorem 6 (Progress). *Suppose $\longrightarrow_{\text{H}}$ satisfies progress with respect to the values v^{H} .*

1. *If $\cdot \vdash t : A$ then either t is a value v^c or there is some t' such that $t \longrightarrow t'$.*
2. *If $;\mathcal{Q} \vdash C : W$ then either C is normal or there is some C' such that $C \Longrightarrow C'$.*

Proof.

1. By the progress hypothesis for $\longrightarrow_{\text{H}}$, either $t = v^{\text{H}}$ for some v^{H} or there exists some t' such that $t \longrightarrow_{\text{H}} t'$ (in which case $t \longrightarrow t'$ as well). In first case however, t is either a value in the original host language (v), or $t = \text{box } p \Rightarrow C$, where

$$\frac{\Omega \Rightarrow p : W_1 \quad ; \Omega \vdash C : W_2}{\cdot \vdash \text{box } p \Rightarrow C : \text{Box } W_1 \ W_2}$$

If p is not concrete for W_1 , then $\text{box } p \Rightarrow C$ can step via the η rule. If p is concrete, then by the inductive hypothesis, C is either normal already (in which case so is $\text{box } p \Rightarrow C$), or there is some C' such that $C \Longrightarrow C'$. In that case, $\text{box } p \Rightarrow C \longrightarrow_b \text{box } p \Rightarrow C'$.

2. By induction on the typing judgment of C .

(a) If the last rule in the derivation is

$$\frac{\cdot \vdash t : \text{Box } W_1 \ W_2 \quad \Omega \Rightarrow p : W_1}{; \Omega \vdash \text{unbox } t \ p : W_2}$$

then by the inductive hypothesis, either t can take a step to some t' , or t is a value of the form $\text{box } p' \Rightarrow N$. In the first case, $\text{unbox } t \ p \Longrightarrow \text{unbox } t' \ p$, and in the second case, $\text{unbox } t \ p \Longrightarrow N \{p'/p\}$.

(b) Next, suppose the last rule in the derivation is

$$\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad ; \Omega_2, \Omega \vdash C : W}{; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W}$$

If C is not concrete, then $p_2 \leftarrow \text{gate } g \ p_1 ; C$ can step via an η rule. Otherwise, C is either normal, in which case $p_2 \leftarrow \text{gate } g \ p_1 ; C$ is also normal, or C can take a step, in which case so can $p_2 \leftarrow \text{gate } g \ p_1 ; C$ by the structural rule.

(c) Suppose the circuit is

$$\frac{; \Omega_1 \vdash C : W \quad ; \Omega_0, \Omega_2 \vdash C' : W'}{; \Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W'}$$

By the inductive hypothesis, either C can take a step, in which case so can $p \leftarrow C ; C'$, or C is normal. The latter cases are straightforward:

$$\begin{aligned} p \leftarrow \text{output } p' ; C &\Longrightarrow C\{p'/p\} \\ p \leftarrow (p_2 \leftarrow \text{gate } g \ p_1 ; C_0) ; C &\Longrightarrow p_2 \leftarrow \text{gate } g \ p_1 ; C_0 ; p \leftarrow C_0 ; C \\ p \leftarrow (x \leftarrow \text{lift } p_0 ; C_0) ; C &\Longrightarrow x \leftarrow \text{lift } p_0 ; p \leftarrow C_0 ; C \end{aligned}$$

□

Theorem 7 (Normalization). *Suppose that $\longrightarrow_{\mathbb{H}}$ is strongly normalizing with respect to $v^{\mathbb{H}}$.*

1. *If $\cdot \vdash t : A$, there exists some value v^c such that $t \longrightarrow^* v^c$.*
2. *If $\cdot; \mathcal{Q} \vdash C : W$, there exists some normal circuit N such that $C \Longrightarrow^* N$.*

Proof. By induction on the number of constructors in the term and circuit.

1. By the normalization property for $\longrightarrow_{\mathbb{H}}$, there is some value v^c such that $t \longrightarrow_{\mathbb{H}}^* v^c$. This value v^c is either a regular host language value v , in which case we are done, or it is some uninterpreted boxed circuit $\mathbf{box} (p : W) \Rightarrow C$. If p is concrete with respect to W , then by the inductive hypothesis, there is some N such that $C \Longrightarrow^* N$, and so $\mathbf{box} p \Rightarrow C \longrightarrow^* \mathbf{box} p \Rightarrow N$.

If p is not concrete, then by an η -expansion, there is some p' that is concrete for W and $\mathbf{box} p \Rightarrow C \longrightarrow_b \mathbf{box} p' \Rightarrow C \{p'/p\}$. By induction we know that $C \{p'/p\}$ normalizes (since the number of constructors in $C \{p'/p\}$ is the same as the number in C), and thus so does $\mathbf{box} p \Rightarrow C$.

2. If C is an output or lifting circuit then it is already normal. If C is an unboxing operator of the form

$$\frac{\cdot \vdash t : \mathbf{Box} W_1 W_2 \quad \Omega \Rightarrow p : W_1}{\cdot; \Omega \vdash \mathbf{unbox} t p : W_2}$$

then by the inductive hypothesis, there is some $\mathbf{box} p' \Rightarrow N$ such that $t \longrightarrow^* \mathbf{box} p' \Rightarrow N$, so $\mathbf{unbox} t p \longrightarrow^* N \{p/p'\}$, which is also normal.

Next, consider a gate application:

$$\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad \cdot; \Omega_2, \Omega \vdash C : W}{\cdot; \Omega_1, \Omega \vdash p_2 \leftarrow \mathbf{gate} g p_1 ; C : W}$$

Again, if C is concrete, it normalizes by the inductive hypothesis; otherwise there is some $\mathcal{Q}_2 \Rightarrow p'_2 : W_2$ where $C \{p'_2/p_2\}$ normalizes to some N , in which case $p_2 \leftarrow \mathbf{gate} g p_1 ; C \Longrightarrow^* p'_2 \leftarrow \mathbf{gate} g p_1 ; N$.

Finally, consider a composition operator:

$$\frac{\cdot; \Omega_1 \vdash C : W \quad \cdot; \Omega_0, \Omega_2 \vdash C' : W'}{\cdot; \Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W'}$$

By the inductive hypothesis, there is some N such that $C \Longrightarrow^* N$. If $N = \mathbf{output} p'$, then $p \leftarrow C ; C' \Longrightarrow^* C' \{p'/p\}$, which normalizes by the inductive hypothesis for C' . If $N = p_2 \leftarrow \mathbf{gate} g p_1 ; C_0$, then $p \leftarrow C ; C'$ normalizes to some N' by the inductive hypothesis, and so

$$p \leftarrow C ; C' \Longrightarrow^* p_2 \leftarrow \mathbf{gate} g p_1 ; N'$$

Finally, if $N = x \leftarrow \text{lift } p'; C_0$, then

$$p \leftarrow C; C' \Longrightarrow x \leftarrow \text{lift } p'; p \leftarrow C_0; C',$$

which is immediately normal. □

B.2 Soundness of denotational semantics

Theorem 8 (Soundness). *If $\cdot; \mathcal{Q} \vdash C : W$ and $C \Longrightarrow C'$, then*

$$\llbracket \mathcal{Q} \vdash C : W \rrbracket = \llbracket \mathcal{Q} \vdash C' : W \rrbracket.$$

Proof. By induction on the typing judgment.

If C is

$$\frac{\cdot; \mathcal{Q}' \vdash C : W \quad \pi : \mathcal{Q} \equiv \mathcal{Q}'}{\cdot; \mathcal{Q} \vdash C : W} \text{ TYP_CLOSED_CONCRETE_PERMUTE}$$

and $C \Longrightarrow C'$, then by the inductive hypothesis,

$$\begin{aligned} \llbracket \mathcal{Q} \vdash C : W \rrbracket &= \llbracket \mathcal{Q}' \vdash C : W \rrbracket \circ [\pi]^* \\ &= \llbracket \mathcal{Q}' \vdash C' : W \rrbracket \circ [\pi]^* = \llbracket \mathcal{Q} \vdash C' : W \rrbracket \end{aligned}$$

If

$$\frac{\cdot \vdash t : \text{Box } W_1 \ W_2 \quad \mathcal{Q} \Rightarrow p : W_1}{\cdot; \mathcal{Q} \vdash \text{unbox } t \ p : W_2} \text{ TYP_CLOSED_CONCRETE_UNBOX}$$

and the circuit steps by a structural rule with $t \longrightarrow t'$, then, assuming HOST is strongly normalizing we have some $\text{box } p' \Rightarrow N$ such that $t, t' \longrightarrow^* \text{box } p' \Rightarrow N$. Then

$$\llbracket \mathcal{Q} \vdash \text{unbox } t \ p : W_2 \rrbracket = \llbracket \mathcal{Q} \vdash \text{unbox } t' \ p : W_2 \rrbracket = \llbracket \mathcal{Q}' \vdash N : W_2 \rrbracket$$

Suppose

$$\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad g \in \mathcal{G}(W_1, W_2) \quad \cdot; \Omega_2, \Omega \vdash C : W}{\cdot; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W}$$

If the circuit steps via a structural rule, the result is immediate. If it steps via an η rule to $p'_2 \leftarrow \text{gate } g \ p_1 ; C \{p'_2/p_2\}$, then the result follows from the fact that $\llbracket C \{p'_2/p_2\} \rrbracket = \llbracket C \rrbracket$ (Lemma 7).

Next, consider

$$\frac{;\mathcal{Q}_1 \vdash C_1 : W \quad \Omega_0 \Rightarrow p : W \quad ;\Omega_0, \mathcal{Q}_2 \vdash C_2 : W'}{;\mathcal{Q}_1, \mathcal{Q}_2 \vdash p \leftarrow C_1; C_2 : W'}$$

If the circuit steps via a structural rule, the result follows immediately. Otherwise, we know C_1 is normal, and the circuit stepped via a β or commuting conversion rule. We proceed by a further case analysis on the typing judgment of C_1 .

For a permutation rule $\pi : \mathcal{Q}_1 \equiv \mathcal{Q}'_1$, by induction we know that

$$\llbracket \mathcal{Q}'_1, \mathcal{Q}_2 \vdash p \leftarrow C_1; C_2 : W' \rrbracket = \llbracket \mathcal{Q}'_1, \mathcal{Q}_2 \vdash C' : W' \rrbracket$$

But then

$$\begin{aligned} & \llbracket \mathcal{Q}_1, \mathcal{Q}_2 \vdash p \leftarrow C_1; C_2 : W' \rrbracket \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \mathcal{Q}_1 \vdash C_1 : W \rrbracket \otimes \mathbf{I}^*) \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ ((\llbracket \mathcal{Q}'_1 \vdash C_1 : W \rrbracket \circ [\pi]^*) \otimes \mathbf{I}^*) \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \mathcal{Q}'_1 \vdash C_1 : W \rrbracket \otimes \mathbf{I}^*) \circ ([\pi] \otimes \mathbf{I})^* \\ &= \llbracket \mathcal{Q}'_1, \mathcal{Q}_2 \vdash p \leftarrow C_1; C_2 : W' \rrbracket \circ ([\pi] \otimes \mathbf{I})^* \\ &= \llbracket \mathcal{Q}_1, \mathcal{Q}_2 \vdash p \leftarrow C_1; C_2 : W' \rrbracket \end{aligned}$$

For $C_1 = \text{output } p'$ with $\mathcal{Q}_1 \Rightarrow p' : W$, where

$$p \leftarrow C_1; C_2 \Longrightarrow C_2 \{p'/p\},$$

we know

$$\begin{aligned} & \llbracket \mathcal{Q}_1, \mathcal{Q}_2 \vdash p \leftarrow \text{output } p'; C_2 : W' \rrbracket \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \mathcal{Q}_1 \vdash \text{output } p' : W \rrbracket \otimes \mathbf{I}^*) \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\mathbf{I}^* \otimes \mathbf{I}^*) \\ &= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket = \llbracket \mathcal{Q}_1, \mathcal{Q}_2 \vdash C_2 \{p'/p\} : W' \rrbracket \end{aligned}$$

by Lemma 7.

If C_1 is

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad \mathcal{Q}'_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad ;\Omega_2, \mathcal{Q}' \vdash C_0 : W}{;\mathcal{Q}'_1, \mathcal{Q}' \vdash p_2 \leftarrow \text{gate } g \ p_1; C_0 : W}$$

and steps via a commuting conversion

$$p \leftarrow C_1; C_2 \Longrightarrow p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow C_0; C_2$$

then

$$\begin{aligned}
& \llbracket \mathcal{Q}'_1, \mathcal{Q}', \mathcal{Q}_2 \vdash p \leftarrow (p_2 \leftarrow \text{gate } g \ p_1; C_0); C_2 : W' \rrbracket \\
&= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \mathcal{Q}'_1, \mathcal{Q}' \vdash p_2 \leftarrow \text{gate } g \ p_1; C_0 : W \rrbracket \otimes \mathbf{I}^*) \\
&= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ ((\llbracket \Omega_2, \mathcal{Q}' \vdash C_0 : W \rrbracket \circ (\llbracket g \rrbracket \otimes \mathbf{I}^*)) \otimes \mathbf{I}^*) \\
&= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \Omega_2, \mathcal{Q}' \vdash C_0 : W \rrbracket \otimes \mathbf{I}^*) \circ (\llbracket g \rrbracket \otimes \mathbf{I}^* \otimes \mathbf{I}^*) \\
&= \llbracket \Omega_2, \mathcal{Q}', \mathcal{Q}_2 \vdash p \leftarrow C_0; C_2 : W' \rrbracket \circ (\llbracket g \rrbracket \otimes \mathbf{I}^*) \\
&= \llbracket \mathcal{Q}'_1, \mathcal{Q}', \mathcal{Q}_2 \vdash p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow C_0; C_2 : W' \rrbracket
\end{aligned}$$

Finally, if C_1 is

$$\frac{\mathcal{Q}_0 \Rightarrow p_0 : W_0 \quad x : |W_0|; \mathcal{Q}' \vdash C_0 : W}{; \mathcal{Q}_0, \mathcal{Q}' \vdash x \leftarrow \text{lift } p_0; C_0 : W}$$

and steps via a commuting conversion

$$p \leftarrow C_1; C_2 \Longrightarrow x \leftarrow \text{lift } p_0; p \leftarrow C_0; C_2$$

then

$$\begin{aligned}
& \llbracket \mathcal{Q}_0, \mathcal{Q}', \mathcal{Q}_2 \vdash p \leftarrow (x \leftarrow \text{lift } p_0; C_0); C_2 : W' \rrbracket \\
&= \llbracket \Omega_0, \mathcal{Q}_2 \vdash C_2 : W' \rrbracket \circ (\llbracket \mathcal{Q}_0, \mathcal{Q}' \vdash x \leftarrow \text{lift } p_0; C_0 : W \rrbracket \otimes \mathbf{I}^*) \\
&= \llbracket C_2 \rrbracket \circ \left(\left(\sum_{\vdash \mathcal{Q}' W_0} \llbracket \mathcal{Q}' \vdash C_0 \{v/x\} : W \rrbracket \circ ([v : |W_0|]^\dagger \otimes \mathbf{I}^*) \right) \otimes \mathbf{I}^* \right) \\
&= \llbracket C_2 \rrbracket \circ \sum_{\vdash \mathcal{Q}' W_0} ((\llbracket \mathcal{Q}' \vdash C_0 \{v/x\} : W \rrbracket \circ ([v : |W_0|]^\dagger \otimes \mathbf{I}^*)) \otimes \mathbf{I}^*) \\
&= \llbracket C_2 \rrbracket \circ \sum_{\vdash \mathcal{Q}' W_0} (\llbracket C_0 \{v/x\} \rrbracket \otimes \mathbf{I}^*) \circ ([v : |W_0|]^\dagger \otimes \mathbf{I}^* \otimes \mathbf{I}^*) \\
&= \sum_{\vdash \mathcal{Q}' W_0} \llbracket C_2 \rrbracket \circ (\llbracket C_0 \{v/x\} \rrbracket \otimes \mathbf{I}^*) \circ ([v : |W_0|]^\dagger \otimes \mathbf{I}^*) \\
&= \sum_{\vdash \mathcal{Q}' W_0} \llbracket p \leftarrow C_0 \{v/x\}; C_2 \rrbracket \circ ([v : |W_0|]^\dagger \otimes \mathbf{I}^*) \\
&= \llbracket x \leftarrow \text{lift } p_0; p \leftarrow C_0; C_2 \rrbracket
\end{aligned}$$

□

Appendix C

Qwire Documentation

C.1 Prelim.v

Lemma `xorb_nb_b` : $\forall b, \neg b \oplus b = \text{true}$.

Lemma `xorb_b_nb` : $\forall b, b \oplus \neg b = \text{true}$.

Lemma `xorb_involutive_l` : $\forall b b', b \oplus (b \oplus b') = b'$.

Lemma `xorb_involutive_r` : $\forall b b', b \oplus b' \oplus b' = b$.

Lemma `andb_xorb_dist` : $\forall b b1 b2, b \ \&\& \ (b1 \oplus b2) = (b \ \&\& \ b1) \oplus (b \ \&\& \ b2)$.

Lemma `beq_reflect` : $\forall x y, \text{reflect } (x = y) (x =? y)$.

Lemma `blt_reflect` : $\forall x y, \text{reflect } (x < y) (x <? y)$.

Lemma `ble_reflect` : $\forall x y, \text{reflect } (x \leq y) (x \leq? y)$.

Lemma `if_dist` : $\forall (A B : \text{Type}) (b : \mathbb{B}) (f : A \rightarrow B) (x y : A),$
 $f (\text{if } b \text{ then } x \text{ else } y) = \text{if } b \text{ then } f x \text{ else } f y$.

Lemma `update_length` : $\forall A (l : \text{list } A) (a : A) (n : \mathbb{N}),$
 $\text{length } (\text{update_at } l \ n \ a) = \text{length } l$.

Lemma `nth_nil` : $\forall \{A\} x, ([] : \text{list } A) \ \! \! \ x = \text{None}$.

Lemma `repeat_combine` : $\forall A n1 n2 (a : A),$
 $\text{List.repeat } a \ n1 \ ++ \ \text{List.repeat } a \ n2 = \text{List.repeat } a \ (n1 + n2)$.

Lemma `rev_repeat` : $\forall A (a : A) n, \text{rev } (\text{repeat } a \ n) = \text{repeat } a \ n$.

Lemma firstn_repeat_le : $\forall A (a : A) m n, (m \leq n) \rightarrow$
 firstn m (repeat a n) = repeat a m.

Lemma firstn_repeat_ge : $\forall A (a : A) m n, (m \geq n) \rightarrow$
 firstn m (repeat a n) = repeat a n.

Lemma firstn_repeat : $\forall A (a : A) m n,$
 firstn m (repeat a n) = repeat a (min m n).

Lemma skipn_repeat : $\forall A (a : A) m n,$
 skipn m (repeat a n) = repeat a (n-m).

Lemma skipn_length : $\forall \{A\} (l : \text{list } A) n,$
 length (skipn n l) = (length l - n).

Lemma disjoint_nil_l : $\forall ls, \text{nil} \perp ls.$

Lemma disjoint_nil_r : $\forall ls, ls \perp \text{nil}.$

Lemma disjoint_cons : $\forall a ls1 ls2,$
 ((negb (inb a ls1)) && disjoint ls1 ls2 = disjoint ls1 (a :: ls2)).

Lemma disjoint_symm : $\forall ls1 ls2, \text{disjoint } ls1 \text{ } ls2 = \text{disjoint } ls2 \text{ } ls1.$

Lemma eqb_neq : $\forall x y, x \neq y \rightarrow x =? y = \text{false}.$

Lemma lookup_app : $\forall x ls1 ls2,$
 lookup x (ls1 ++ ls2) = if inb x ls1 then lookup x ls1
 else (lookup x ls2 + length ls1).

Lemma subset_app : $\forall ls1 ls2 ls, (ls1 ++ ls2) \subseteq ls \rightarrow ls1 \subseteq ls \wedge ls2 \subseteq ls.$

Lemma seq_app : $\forall \text{offset1 offset2 start},$
 seq start offset1 ++ seq (start + offset1) offset2
 = seq start (offset1 + offset2).

Lemma inb_fmap_S : $\forall ls x,$
 inb (S x) (fmap S ls) = inb x ls.

Lemma double_mult : $\forall (n : \mathbb{N}), (n + n = 2 * n).$

Lemma pow_two_succ_l : $\forall x, (2^x * 2 = 2 ^ (x + 1)).$

Lemma pow_two_succ_r : $\forall x, (2 * 2^x = 2 ^ (x + 1)).$

Lemma double_pow : $\forall (n : \mathbb{N}), (2^n + 2^n = 2^{(n+1)})$.

Lemma pow_components : $\forall (a b m n : \mathbb{N}), a = b \rightarrow m = n \rightarrow (a^m = b^n)$.

C.2 Monad.v

Lemma fmap_compose' {f} (F : Functor f) {Functor_Correct f} :
 $\forall \{A B C\} (g : A \rightarrow B) (h : B \rightarrow C) (a : f A),$
fmap h (fmap g a) = fmap (h \circ g) a.

Lemma bind_eq : $\forall \{A B m\} \text{ `}{Monad m\} (a a' : m A) (f f' : A \rightarrow m B),$
a = a' \rightarrow
($\forall x, f x = f' x$) \rightarrow
bind a f = bind a' f'.

Lemma fmap_app : $\forall \{A B\} (f : A \rightarrow B) ls1 ls2,$
fmap f (ls1 ++ ls2) = fmap f ls1 ++ fmap f ls2.

C.3 Monoid.v

Lemma M_unit_l : $\forall a, \top \circ a = a$.

Lemma M_comm_assoc : $\forall a b c, a \circ b \circ c = b \circ a \circ c$.

Lemma M_comm_assoc_r : $\forall a b c, a \circ (b \circ c) = b \circ (a \circ c)$.

Lemma M_absorb_l : $\forall a, \perp \circ a = \perp$.

Lemma translate_Some : $\forall \{X\} \text{ `}{Translate X A\} (x : A),$
 $\langle \text{Some } x \rangle = \langle x \rangle$.

Lemma flatten_correct' : $\forall (ls1 ls2 : \text{list } A),$
 $\langle ls1 \rangle \circ \langle ls2 \rangle = \langle ls1 ++ ls2 \rangle$.

Lemma option_list_correct : $\forall (o1 o2 : \text{option } (\text{list } A)),$
 $\langle o1 \rangle \circ \langle o2 \rangle = \langle \text{do } ls1 \leftarrow o1;$
 $\text{do } ls2 \leftarrow o2;$
 $\text{return_ } (ls1 ++ ls2) \rangle$.

Lemma flatten_correct : $\forall e, \langle e \rangle = \langle \text{flatten } e \rangle$.

Lemma `index_wrt_cons` : \forall `idx a values`,
`index_wrt (a :: values) (fmap S idx) = index_wrt values idx`.

Lemma `index_wrt_default` : \forall (`ls : list A`),
`index_wrt ls (nats_lt (length ls)) = ls`.

Lemma `split_list` : \forall `values ls1 ls2`,
`\langle index_wrt values (ls1 ++ ls2) \rangle =`
`\langle index_wrt values ls1 \rangle \circ \langle index_wrt values ls2 \rangle`.

Lemma `in_interp_nats` : \forall `i a values idx`,
`In i idx \rightarrow`
`index values i = Some a \rightarrow`
`In a (index_wrt values idx)`.

Lemma `in_index` : \forall `i a values`,
`\langle index values i \rangle = a \rightarrow a = \perp \vee In a values`.

Lemma `in_index_wrt` : \forall `a idx values`,
`In a (index_wrt values idx) \rightarrow`
`a = \perp \vee In a values`.

Lemma `interp_permutation` : \forall (`values : list A`) (`idx1 idx2 : list \mathbb{N}`),
`Permutation idx1 idx2 \rightarrow`
`\langle index_wrt values idx1 \rangle = \langle index_wrt values idx2 \rangle`.

Lemma `permutation_reflection` : \forall `ls1 ls2`,
`@permutation \mathbb{N} _ PeanoNat.Nat.eq_dec ls1 ls2 \rightarrow Permutation ls1 ls2`.

Lemma `meq_multiplicity` : \forall (`ls1 ls2 : list \mathbb{N}`),
`(\forall x, In x ls1 \vee In x ls2 \rightarrow
multiplicity (contents ls1) x = multiplicity (contents ls2) x) \rightarrow
meq (contents ls1) (contents ls2).`

Lemma `interp_0` : \forall (`ls : list A`),
`In \perp ls \rightarrow \langle ls \rangle = \perp` .

C.4 Matrix.v

Lemma `mat_equiv_refl` : \forall `m n (A : Matrix m n)`, `mat_equiv A A`.

Lemma `mat_equiv_eq` : \forall `{m n : \mathbb{N} }` (`A B : Matrix m n`),
`WF_Matrix m n A \rightarrow`

$\text{WF_Matrix } m \ n \ B \rightarrow$
 $\text{mat_equiv } A \ B \rightarrow$
 $A = B.$

Lemma $\text{WF_list2D_to_matrix} : \forall m \ n \ li,$
 $\text{length } li = m \rightarrow$
 $(\forall li', \text{In } li' \ li \rightarrow \text{length } li' = n) \rightarrow$
 $\text{WF_Matrix } m \ n \ (\text{list2D_to_matrix } li).$

Lemma $\text{M23eq} : \text{M23} = \text{M23}'.$

Lemma $\text{Csum_0} : \forall f \ n, (\forall x, f \ x = C0) \rightarrow \text{Csum } f \ n = 0.$

Lemma $\text{Csum_1} : \forall f \ n, (\forall x, f \ x = C_1) \rightarrow \text{Csum } f \ n = \text{INR } n.$

Lemma $\text{Csum_constant} : \forall c \ n, \text{Csum } (\text{fun } x \Rightarrow c) \ n = \text{INR } n * c.$

Lemma $\text{Csum_eq} : \forall f \ g \ n, f = g \rightarrow \text{Csum } f \ n = \text{Csum } g \ n.$

Lemma $\text{Csum_0_bounded} : \forall f \ n, (\forall x, (x < n) \rightarrow f \ x = C0) \rightarrow \text{Csum } f \ n = 0.$

Lemma $\text{Csum_eq_bounded} : \forall f \ g \ n, (\forall x, (x < n) \rightarrow f \ x = g \ x) \rightarrow$
 $\text{Csum } f \ n = \text{Csum } g \ n.$

Lemma $\text{Csum_plus} : \forall f \ g \ n,$
 $\text{Csum } (\text{fun } x \Rightarrow f \ x + g \ x) \ n = \text{Csum } f \ n + \text{Csum } g \ n.$

Lemma $\text{Csum_mult_l} : \forall c \ f \ n, c * \text{Csum } f \ n = \text{Csum } (\text{fun } x \Rightarrow c * f \ x) \ n.$

Lemma $\text{Csum_mult_r} : \forall c \ f \ n, \text{Csum } f \ n * c = \text{Csum } (\text{fun } x \Rightarrow f \ x * c) \ n.$

Lemma $\text{Csum_conj_distr} : \forall f \ n, (\text{Csum } f \ n)^* = \text{Csum } (\text{fun } x \Rightarrow (f \ x)^*) \ n.$

Lemma $\text{Csum_extend_r} : \forall n \ f, \text{Csum } f \ n + f \ n = \text{Csum } f \ (S \ n).$

Lemma $\text{Csum_extend_l} : \forall n \ f,$
 $f \ 0 + \text{Csum } (\text{fun } x \Rightarrow f \ (S \ x)) \ n = \text{Csum } f \ (S \ n).$

Lemma $\text{Csum_unique} : \forall k \ (f : \mathbb{N} \rightarrow \mathbb{C}) \ n,$
 $(\exists x, (x < n) \wedge f \ x = k \wedge (\forall x', x \neq x' \rightarrow f \ x' = 0)) \rightarrow$
 $\text{Csum } f \ n = k.$

Lemma $\text{Csum_sum} : \forall m \ n \ f, \text{Csum } f \ (m + n) =$
 $\text{Csum } f \ m + \text{Csum } (\text{fun } x \Rightarrow f \ (m + x)) \ n.$

Lemma Csum_product : $\forall m n f g, n \neq 0 \rightarrow$
 Csum f m * Csum g n =
 Csum ($\text{fun } x \Rightarrow f (x / n) * g (x \text{ mod } n)$) (m * n).

Lemma Csum_ge_0 : $\forall f n, (\forall x, 0 \leq \text{fst } (f x)) \rightarrow 0 \leq \text{fst } (\text{Csum } f n)$.

Lemma Csum_member_le : $\forall (f : \mathbb{N} \rightarrow \mathbb{C}) (n : \mathbb{N}), (\forall x, 0 \leq \text{fst } (f x)) \rightarrow$
 $(\forall x, (x < n) \rightarrow \text{fst } (f x) \leq \text{fst } (\text{Csum } f n))$.

Lemma WF_Zero : $\forall \{m n : \mathbb{N}\}, \text{WF_Matrix } m n (\text{Zero } m n)$.

Lemma WF_Id : $\forall \{n : \mathbb{N}\}, \text{WF_Matrix } n n (\text{Id } n)$.

Lemma WF_I1 : $\text{WF_Matrix } 1 1 \text{I1}$.

Lemma WF_scale : $\forall \{m n : \mathbb{N}\} (r : \mathbb{C}) (A : \text{Matrix } m n),$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } m n (\text{scale } r A)$.

Lemma WF_plus : $\forall \{m n\} (A B : \text{Matrix } m n),$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } m n B \rightarrow \text{WF_Matrix } m n (A .+ B)$.

Lemma WF_mult : $\forall \{m n o : \mathbb{N}\} (A : \text{Matrix } m n) (B : \text{Matrix } n o),$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } n o B \rightarrow \text{WF_Matrix } m o (A \times B)$.

Lemma WF_kron : $\forall \{m n o p q r : \mathbb{N}\} (A : \text{Matrix } m n) (B : \text{Matrix } o p),$
 $q = m * o \rightarrow r = n * p \rightarrow$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } o p B \rightarrow \text{WF_Matrix } q r (A \otimes B)$.

Lemma WF_transpose : $\forall \{m n : \mathbb{N}\} (A : \text{Matrix } m n),$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } n m A^\top$.

Lemma WF_adjoint : $\forall \{m n : \mathbb{N}\} (A : \text{Matrix } m n),$
 $\text{WF_Matrix } m n A \rightarrow \text{WF_Matrix } n m A^\dagger$.

Lemma WF_outer_product : $\forall \{n\} (v : \text{Matrix } n 1),$
 $\text{WF_Matrix } n 1 v \rightarrow$
 $\text{WF_Matrix } n n (\text{outer_product } v)$.

Lemma WF_big_kron : $\forall n m (l : \text{list } (\text{Matrix } m n)) (A : \text{Matrix } m n),$
 $(\forall i, \text{WF_Matrix } m n (\text{nth } i l A)) \rightarrow$
 $\text{WF_Matrix } (m^{(\text{length } l)}) (n^{(\text{length } l)}) (\otimes l)$.

Lemma WFO_Zero_1 : $\forall (n : \mathbb{N}) (A : \text{Matrix } 0 n),$
 $\text{WF_Matrix } _ _ A \rightarrow$
 $A = \text{Zero } 0 n$.

Lemma WFO_Zero_r : $\forall (n : \mathbb{N}) (A : \text{Matrix } n \ 0),$
WF_Matrix _ _ A \rightarrow
A = Zero n 0.

Lemma WFO_Zero : $\forall (A : \text{Matrix } 0 \ 0),$ WF_Matrix _ _ A \rightarrow A = Zero 0 0.

Lemma Id0_Zero : 'I_ 0 = Zero 0 0.

Lemma trace_plus_dist : $\forall (n : \mathbb{N}) (A \ B : \text{Square } n),$
trace (A .+ B) = (trace A + trace B).

Lemma trace_mult_dist : $\forall n \ p (A : \text{Square } n),$
trace (p .* A) = (p * trace A).

Lemma Mplus_0_l : $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$ Zero m n .+ A = A.

Lemma Mplus_0_r : $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$ A .+ Zero m n = A.

Lemma Mmult_0_l : $\forall (m \ n \ o : \mathbb{N}) (A : \text{Matrix } n \ o),$ Zero m n \times A = Zero m o.

Lemma Mmult_0_r : $\forall (m \ n \ o : \mathbb{N}) (A : \text{Matrix } m \ n),$ A \times Zero n o = Zero m o.

Lemma Mmult_1_l_gen: $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n) (x \ z \ k : \mathbb{N}),$
k \leq m \rightarrow
(k \leq x \rightarrow Csum (fun y : $\mathbb{N} \Rightarrow ((\text{Id } m) \ x \ y * A \ y \ z)) \ k = C0) \wedge$
(k > x \rightarrow Csum (fun y : $\mathbb{N} \Rightarrow ((\text{Id } m) \ x \ y * A \ y \ z)) \ k = A \ x \ z).$

Lemma Mmult_1_l_mat_eq : $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$ Id m \times A \equiv A.

Lemma Mmult_1_l: $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$
WF_Matrix m n A \rightarrow Id m \times A = A.

Lemma Mmult_1_r_gen: $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n) (x \ z \ k : \mathbb{N}),$
k \leq n \rightarrow
(k \leq z \rightarrow Csum (fun y : $\mathbb{N} \Rightarrow (A \ x \ y * (\text{Id } n) \ y \ z)) \ k = C0) \wedge$
(k > z \rightarrow Csum (fun y : $\mathbb{N} \Rightarrow (A \ x \ y * (\text{Id } n) \ y \ z)) \ k = A \ x \ z).$

Lemma Mmult_1_r_mat_eq : $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$ A \times Id n \equiv A.

Lemma Mmult_1_r: $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$
WF_Matrix m n A \rightarrow A \times Id n = A.

Lemma Mmult_inf_l : $\forall (m \ n : \mathbb{N}) (A : \text{Matrix } m \ n),$
WF_Matrix m n A \rightarrow ∞ I \times A = A.

Lemma `Mmult_inf_r` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n),$
 $\text{WF_Matrix } m\ n\ A \rightarrow A \times \infty I = A.$

Lemma `kron_0_l` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } o\ p),$
 $\text{Zero } m\ n \otimes A = \text{Zero } (m * o)\ (n * p).$

Lemma `kron_0_r` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } m\ n),$
 $A \otimes \text{Zero } o\ p = \text{Zero } (m * o)\ (n * p).$

Lemma `kron_1_r` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n), A \otimes \text{Id } 1 = A.$

Lemma `kron_1_l` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n),$
 $\text{WF_Matrix } m\ n\ A \rightarrow \text{Id } 1 \otimes A = A.$

Theorem `transpose_involutive` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n), (A^\top)^\top = A.$

Theorem `adjoint_involutive` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n), A^{\dagger\dagger} = A.$

Lemma `id_transpose_eq` : $\forall n, (\text{Id } n)^\top = (\text{Id } n).$

Lemma `zero_transpose_eq` : $\forall m\ n, (\text{Zero } m\ n)^\top = (\text{Zero } n\ m).$

Lemma `id_adjoint_eq` : $\forall n, (\text{Id } n)^\dagger = (\text{Id } n).$

Lemma `zero_adjoint_eq` : $\forall m\ n, (\text{Zero } m\ n)^\dagger = (\text{Zero } n\ m).$

Theorem `Mplus_comm` : $\forall (m\ n : \mathbb{N}) (A\ B : \text{Matrix } m\ n),$
 $A .+ B = B .+ A.$

Theorem `Mplus_assoc` : $\forall (m\ n : \mathbb{N}) (A\ B\ C : \text{Matrix } m\ n),$
 $A .+ B .+ C = A .+ (B .+ C).$

Theorem `Mmult_assoc` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } n\ o)$
 $(C : \text{Matrix } o\ p), A \times B \times C = A \times (B \times C).$

Lemma `Mmult_plus_distr_l` : $\forall (m\ n\ o : \mathbb{N}) (A : \text{Matrix } m\ n)$
 $(B\ C : \text{Matrix } n\ o), A \times (B .+ C) = A \times B .+ A \times C.$

Lemma `Mmult_plus_distr_r` : $\forall (m\ n\ o : \mathbb{N}) (A\ B : \text{Matrix } m\ n)$
 $(C : \text{Matrix } n\ o), (A .+ B) \times C = A \times C .+ B \times C.$

Lemma `kron_plus_distr_l` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } m\ n)$
 $(B\ C : \text{Matrix } o\ p), A \otimes (B .+ C) = A \otimes B .+ A \otimes C.$

Lemma `kron_plus_distr_r` : $\forall (m\ n\ o\ p : \mathbb{N}) (A\ B : \text{Matrix } m\ n)$
 $(C : \text{Matrix } o\ p), (A .+ B) \otimes C = A \otimes C .+ B \otimes C.$

Lemma `Mscale_0_l` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n), C_0 .* A = \text{Zero } m\ n.$

Lemma `Mscale_0_r` : $\forall (m\ n : \mathbb{N}) (c : C), c .* \text{Zero } m\ n = \text{Zero } m\ n.$

Lemma `Mscale_1_l` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n), C_1 .* A = A.$

Lemma `Mscale_1_r` : $\forall (n : \mathbb{N}) (c : C),$
 $c .* 'I_n = \text{fun } x\ y \Rightarrow \text{if } (x =? y) \ \&\& \ (x <? n) \ \text{then } c \ \text{else } C_0.$

Lemma `Mscale_mult_dist_l` : $\forall (m\ n\ o : \mathbb{N}) (x : C) (A : \text{Matrix } m\ n)$
 $(B : \text{Matrix } n\ o), ((x .* A) \times B) = x .* (A \times B).$

Lemma `Mscale_mult_dist_r` : $\forall (m\ n\ o : \mathbb{N}) (x : C) (A : \text{Matrix } m\ n)$
 $(B : \text{Matrix } n\ o), (A \times (x .* B)) = x .* (A \times B).$

Lemma `Mscale_kron_dist_l` : $\forall (m\ n\ o\ p : \mathbb{N}) (x : C) (A : \text{Matrix } m\ n)$
 $(B : \text{Matrix } o\ p), ((x .* A) \otimes B) = x .* (A \otimes B).$

Lemma `Mscale_kron_dist_r` : $\forall (m\ n\ o\ p : \mathbb{N}) (x : C) (A : \text{Matrix } m\ n)$
 $(B : \text{Matrix } o\ p), (A \otimes (x .* B)) = x .* (A \otimes B).$

Lemma `Minv_unique` : $\forall (n : \mathbb{N}) (A\ B\ C : \text{Square } n),$
 $\text{WF_Matrix } n\ n\ A \rightarrow \text{WF_Matrix } n\ n\ B \rightarrow \text{WF_Matrix } n\ n\ C \rightarrow$
 $\text{Minv } A\ B \rightarrow \text{Minv } A\ C \rightarrow B = C.$

Lemma `Minv_symm` : $\forall (n : \mathbb{N}) (A\ B : \text{Square } n), \text{Minv } A\ B \rightarrow \text{Minv } B\ A.$

Lemma `Minv_left` : $\forall (n : \mathbb{N}) (A\ B : \text{Square } n),$
 $A \times B = \text{Id } n \rightarrow \text{Minv } A\ B.$

Lemma `Minv_right` : $\forall (n : \mathbb{N}) (A\ B : \text{Square } n),$
 $B \times A = \text{Id } n \rightarrow \text{Minv } A\ B.$

Lemma `kron_mixed_product` : $\forall (m\ n\ o\ p\ q\ r : \mathbb{N})$
 $(A : \text{Matrix } m\ n) (B : \text{Matrix } p\ q) (C : \text{Matrix } n\ o) (D : \text{Matrix } q\ r),$
 $(A \otimes B) \times (C \otimes D) = (A \times C) \otimes (B \times D).$

Lemma `Mplus_tranpose` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } m\ n),$
 $(A .+ B)^T = A^T .+ B^T.$

Lemma `Mmult_tranpose` : $\forall (m\ n\ o : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } n\ o),$
 $(A \times B)^T = B^T \times A^T.$

Lemma `kron_transpose` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } o\ p),$
 $(A \otimes B)^T = A^T \otimes B^T.$

Lemma `Mplus_adjoint` : $\forall (m\ n : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } m\ n),$
 $(A .+ B)^\dagger = A^\dagger .+ B^\dagger.$

Lemma `Mmult_adjoint` : $\forall (m\ n\ o : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } n\ o),$
 $(A \times B)^\dagger = B^\dagger \times A^\dagger.$

Lemma `kron_adjoint` : $\forall (m\ n\ o\ p : \mathbb{N}) (A : \text{Matrix } m\ n) (B : \text{Matrix } o\ p),$
 $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger.$

Lemma `id_kron` : $\forall (m\ n : \mathbb{N}), \text{Id } m \otimes \text{Id } n = \text{Id } (m * n).$

Lemma `outer_product_eq` : $\forall m (\phi\ \psi : \text{Matrix } m\ 1), \phi = \psi \rightarrow$
`outer_product` $\phi = \text{outer_product } \psi.$

Lemma `outer_product_kron` : $\forall m\ n (\phi : \text{Matrix } m\ 1) (\psi : \text{Matrix } n\ 1),$
`outer_product` $\phi \otimes \text{outer_product } \psi = \text{outer_product } (\phi \otimes \psi).$

Lemma `divmod_eq` : $\forall x\ y\ n\ z,$
`fst` $(\text{Nat.divmod } x\ y\ n\ z) = (n + \text{fst } (\text{Nat.divmod } x\ y\ 0\ z)).$

Lemma `divmod_S` : $\forall x\ y\ n\ z,$
`fst` $(\text{Nat.divmod } x\ y\ (S\ n)\ z) = (S\ n + \text{fst } (\text{Nat.divmod } x\ y\ 0\ z)).$

Lemma `divmod_0q0` : $\forall x\ q, \text{fst } (\text{Nat.divmod } x\ 0\ q\ 0) = (x + q).$

Lemma `divmod_0` : $\forall x, \text{fst } (\text{Nat.divmod } x\ 0\ 0\ 0) = x.$

Lemma `kron_shadow` : `@kron = kron'.`

Lemma `Mmult_shadow` : `@Mmult = Mmult'.`

C.5 Quantum.v

Lemma `bool_to_matrix_eq` : $\forall b, \text{bool_to_matrix } b = \text{bool_to_matrix}'\ b.$

Lemma `bool_to_ket_matrix_eq` : $\forall b,$
`outer_product` $(\text{bool_to_ket } b) = \text{bool_to_matrix } b.$

Lemma `hadamard_1` : `hadamard_k 1 = hadamard.`

Lemma `cnot_eq` : `cnot = control σ_x` .
Lemma `MmultX1` : `$\sigma_x \times |1\rangle = |0\rangle$` .
Lemma `Mmult1X` : `$\langle 1| \times \sigma_x = \langle 0|$` .
Lemma `MmultX0` : `$\sigma_x \times |0\rangle = |1\rangle$` .
Lemma `Mmult0X` : `$\langle 0| \times \sigma_x = \langle 1|$` .
Lemma `swap_swap` : `swap \times swap = Id 4`.
Lemma `swap_swap_r` : `$\forall n A, \text{WF_Matrix } n \ 4 \ A \rightarrow$
 $A \times \text{swap} \times \text{swap} = A$` .
Lemma `WF_bra0` : `WF_Matrix 1 2 $\langle 0|$` .
Lemma `WF_bra1` : `WF_Matrix 1 2 $\langle 1|$` .
Lemma `WF_ket0` : `WF_Matrix 2 1 $|0\rangle$` .
Lemma `WF_ket1` : `WF_Matrix 2 1 $|1\rangle$` .
Lemma `WF_braket0` : `WF_Matrix 2 2 $|0\rangle\langle 0|$` .
Lemma `WF_braket1` : `WF_Matrix 2 2 $|1\rangle\langle 1|$` .
Lemma `WF_bool_to_ket` : `$\forall b, \text{WF_Matrix } 2 \ 1 \ (\text{bool_to_ket } b)$` .
Lemma `WF_bool_to_matrix` : `$\forall b, \text{WF_Matrix } 2 \ 2 \ (\text{bool_to_matrix } b)$` .
Lemma `WF_bool_to_matrix'` : `$\forall b, \text{WF_Matrix } 2 \ 2 \ (\text{bool_to_matrix}' b)$` .
Lemma `WF_bools_to_matrix` : `$\forall l,$
 $\text{WF_Matrix } (2^{(\text{length } l)}) \ (2^{(\text{length } l)}) \ (\text{bools_to_matrix } l)$` .
Lemma `WF_hadamard` : `WF_Matrix 2 2 hadamard`.
Lemma `WF_sx` : `WF_Matrix 2 2 σ_x` .
Lemma `WF_sy` : `WF_Matrix 2 2 σ_y` .
Lemma `WF_sz` : `WF_Matrix 2 2 σ_z` .

Lemma WF_cnot : WF_Matrix 4 4 cnot.
Lemma WF_swap : WF_Matrix 4 4 swap.
Lemma WF_phase : $\forall \varphi$, WF_Matrix 2 2 (phase_shift φ).
Lemma WF_control : $\forall (n\ m : \mathbb{N}) (U : \text{Matrix } n\ n)$,
 $(m = 2 * n) \rightarrow \text{WF_Matrix } n\ n\ U \rightarrow \text{WF_Matrix } m\ m\ (\text{control } U)$.
Lemma H_unitary : WF_Unitary hadamard.
Lemma σ_x _unitary : WF_Unitary σ_x .
Lemma σ_y _unitary : WF_Unitary σ_y .
Lemma σ_z _unitary : WF_Unitary σ_z .
Lemma phase_unitary : $\forall \varphi$, @WF_Unitary 2 (phase_shift φ).
Lemma control_unitary : $\forall n (A : \text{Matrix } n\ n)$,
WF_Unitary A \rightarrow WF_Unitary (control A).
Lemma transpose_unitary : $\forall n (A : \text{Matrix } n\ n)$,
WF_Unitary A \rightarrow WF_Unitary (A^\dagger).
Lemma cnot_unitary : WF_Unitary cnot.
Lemma id_unitary : $\forall n$, WF_Unitary (Id n).
Lemma swap_unitary : WF_Unitary swap.
Lemma kron_unitary : $\forall \{m\ n\} (A : \text{Matrix } m\ m) (B : \text{Matrix } n\ n)$,
WF_Unitary A \rightarrow WF_Unitary B \rightarrow WF_Unitary (A \otimes B).
Lemma Mmult_unitary : $\forall (n : \mathbb{N}) (A : \text{Square } n) (B : \text{Square } n)$,
WF_Unitary A \rightarrow WF_Unitary B \rightarrow WF_Unitary (A \times B).
Lemma hadamard_sa : hadamard † = hadamard.
Lemma σ_x _sa : $\sigma_x^\dagger = \sigma_x$.
Lemma σ_y _sa : $\sigma_y^\dagger = \sigma_y$.
Lemma σ_z _sa : $\sigma_z^\dagger = \sigma_z$.

Lemma cnot_sa : $\text{cnot}^\dagger = \text{cnot}$.

Lemma swap_sa : $\text{swap}^\dagger = \text{swap}$.

Lemma control_adjoint : $\forall n (U : \text{Square } n), (\text{control } U)^\dagger = \text{control } (U^\dagger)$.

Lemma control_sa : $\forall (n : \mathbb{N}) (A : \text{Square } n), A^\dagger = A \rightarrow (\text{control } A)^\dagger = (\text{control } A)$.

Lemma phase_adjoint : $\forall \varphi, (\text{phase_shift } \varphi)^\dagger = \text{phase_shift } (-\varphi)$.

Lemma braket0_sa : $|0\rangle\langle 0|^\dagger = |0\rangle\langle 0|$.

Lemma braket1_sa : $|1\rangle\langle 1|^\dagger = |1\rangle\langle 1|$.

Lemma braket0_psd : $\text{positive_semidefinite } |0\rangle\langle 0|$.

Lemma braket1_psd : $\text{positive_semidefinite } |1\rangle\langle 1|$.

Lemma HO_psd : $\text{positive_semidefinite } (\text{hadamard} \times |0\rangle\langle 0| \times \text{hadamard})$.

Lemma WF_Pure : $\forall \{n\} (\rho : \text{Density } n), \text{Pure_State } \rho \rightarrow \text{WF_Matrix } n \ n \ \rho$.

Lemma WF_Mixed : $\forall \{n\} (\rho : \text{Density } n), \text{Mixed_State } \rho \rightarrow \text{WF_Matrix } n \ n \ \rho$.

Lemma pure0 : $\text{Pure_State } |0\rangle\langle 0|$.

Lemma pure1 : $\text{Pure_State } |1\rangle\langle 1|$.

Lemma pure_id1 : $\text{Pure_State } ('I_1)$.

Lemma pure_dim1 : $\forall (\rho : \text{Square } 1), \text{Pure_State } \rho \rightarrow \rho = 'I_1$.

Lemma pure_state_kron : $\forall m \ n (\rho : \text{Square } m) (\phi : \text{Square } n), \text{Pure_State } \rho \rightarrow \text{Pure_State } \phi \rightarrow \text{Pure_State } (\rho \otimes \phi)$.

Lemma mixed_state_kron : $\forall m \ n (\rho : \text{Square } m) (\phi : \text{Square } n), \text{Mixed_State } \rho \rightarrow \text{Mixed_State } \phi \rightarrow \text{Mixed_State } (\rho \otimes \phi)$.

Lemma pure_state_trace_1 : $\forall \{n\} (\rho : \text{Density } n), \text{Pure_State } \rho \rightarrow \text{trace } \rho = 1$.

Lemma mixed_state_trace_1 : $\forall \{n\} (\rho : \text{Density } n), \text{Mixed_State } \rho \rightarrow \text{trace } \rho = 1$.

Lemma mixed_state_diag_in01 : $\forall \{n\} (\rho : \text{Density } n) \ i, \text{Mixed_State } \rho \rightarrow 0 \leq \text{fst } (\rho \ i \ i) \leq 1$.

Lemma `mixed_state_diag_real` : $\forall \{n\} (\rho : \text{Density } n) i , \text{Mixed_State } \rho \rightarrow \text{snd } (\rho \ i \ i) = 0.$

Lemma `mixed_dim1` : $\forall (\rho : \text{Square } 1), \text{Mixed_State } \rho \rightarrow \rho = 'I_1.$

Lemma `super_I` : $\forall n \rho,$
 $\text{WF_Matrix } n \ n \ \rho \rightarrow$
 $\text{super } ('I_n) \ \rho = \rho.$

Lemma `WF_super` : $\forall m \ n \ (U : \text{Matrix } m \ n) (\rho : \text{Square } n),$
 $\text{WF_Matrix } m \ n \ U \rightarrow \text{WF_Matrix } n \ n \ \rho \rightarrow \text{WF_Matrix } m \ m \ (\text{super } U \ \rho).$

Lemma `super_outer_product` : $\forall m (\phi : \text{Matrix } m \ 1) (U : \text{Matrix } m \ m),$
 $\text{super } U \ (\text{outer_product } \phi) = \text{outer_product } (U \times \phi).$

Lemma `WF_compose_super` : $\forall m \ n \ p (g : \text{Superoperator } n \ p)$
 $(f : \text{Superoperator } m \ n) (\rho : \text{Square } m),$
 $\text{WF_Matrix } m \ m \ \rho \rightarrow$
 $(\forall A, \text{WF_Matrix } m \ m \ A \rightarrow \text{WF_Matrix } n \ n \ (f \ A)) \rightarrow$
 $(\forall A, \text{WF_Matrix } n \ n \ A \rightarrow \text{WF_Matrix } p \ p \ (g \ A)) \rightarrow$
 $\text{WF_Matrix } p \ p \ (\text{compose_super } g \ f \ \rho).$

Lemma `compose_super_correct` : $\forall \{m \ n \ p\}$
 $(g : \text{Superoperator } n \ p) (f : \text{Superoperator } m \ n),$
 $\text{WF_Superoperator } g \rightarrow$
 $\text{WF_Superoperator } f \rightarrow$
 $\text{WF_Superoperator } (\text{compose_super } g \ f).$

Lemma `sum_super_correct` : $\forall m \ n (f \ g : \text{Superoperator } m \ n),$
 $\text{WF_Superoperator } f \rightarrow \text{WF_Superoperator } g \rightarrow$
 $\text{WF_Superoperator } (\text{sum_super } f \ g).$

Lemma `pure_unitary` : $\forall \{n\} (U \ \rho : \text{Matrix } n \ n),$
 $\text{WF_Unitary } U \rightarrow \text{Pure_State } \rho \rightarrow \text{Pure_State } (\text{super } U \ \rho).$

Lemma `mixed_unitary` : $\forall \{n\} (U \ \rho : \text{Matrix } n \ n),$
 $\text{WF_Unitary } U \rightarrow \text{Mixed_State } \rho \rightarrow \text{Mixed_State } (\text{super } U \ \rho).$

Lemma `super_unitary_correct` : $\forall \{n\} (U : \text{Matrix } n \ n),$
 $\text{WF_Unitary } U \rightarrow \text{WF_Superoperator } (\text{super } U).$

Lemma `compose_super_assoc` : $\forall \{m \ n \ p \ q\}$
 $(f : \text{Superoperator } m \ n) (g : \text{Superoperator } n \ p) (h : \text{Superoperator } p \ q),$
 $\text{compose_super } (\text{compose_super } f \ g) \ h = \text{compose_super } f \ (\text{compose_super } g \ h).$

Lemma WF_Superoperator_compose : $\forall m n p (s : \text{Superoperator } n p)$
 $(s' : \text{Superoperator } m n), \text{WF_Superoperator } s \rightarrow \text{WF_Superoperator } s' \rightarrow$
 $\text{WF_Superoperator } (\text{compose_super } s s').$

Lemma swap_spec : $\forall (q q' : \text{Matrix } 2\ 1), \text{WF_Matrix } 2\ 1\ q \rightarrow$
 $\text{WF_Matrix } 2\ 1\ q' \rightarrow \text{swap} \times (q \otimes q') = q' \otimes q.$

Lemma swap_two_base : $\text{swap_two } 2\ 1\ 0 = \text{swap}.$

Lemma swap_second_two : $\text{swap_two } 3\ 1\ 2 = \text{Id } 2 \otimes \text{swap}.$

Lemma swap_0_2 :
 $\text{swap_two } 3\ 0\ 2 = ('I_2 \otimes \text{swap}) \times (\text{swap} \otimes 'I_2) \times ('I_2 \otimes \text{swap}).$

C.6 Contexts.v

Lemma size_ntensor : $\forall n W, \text{size_wtype } (n \otimes W) = (n * \text{size_wtype } W).$

Lemma ctx_octx : $\forall \Gamma \Gamma', \text{Valid } \Gamma = \text{Valid } \Gamma' \leftrightarrow \Gamma = \Gamma'.$

Lemma size_ctx_size_octx : $\forall (\Gamma : \text{Ctx}), \text{size_ctx } \Gamma = \text{size_octx } (\text{Valid } \Gamma).$

Lemma size_ctx_app : $\forall (\Gamma_1 \Gamma_2 : \text{Ctx}),$
 $\text{size_ctx } (\Gamma_1 ++ \Gamma_2) = (\text{size_ctx } \Gamma_1 + \text{size_ctx } \Gamma_2).$

Lemma Singleton_size : $\forall x w \Gamma, \text{SingletonCtx } x\ w\ \Gamma \rightarrow \text{size_ctx } \Gamma = 1.$

Lemma singleton_singleton : $\forall x W,$
 $\text{SingletonCtx } x\ W\ (\text{singleton } x\ W).$

Lemma singleton_equiv : $\forall x W \Gamma,$
 $\text{SingletonCtx } x\ W\ \Gamma \rightarrow \Gamma = \text{singleton } x\ W.$

Lemma singleton_size : $\forall x w, \text{size_ctx } (\text{singleton } x\ w) = 1.$

Lemma merge_shadow : $\text{merge} = \text{fun } \Gamma_1 \Gamma_2 \Rightarrow$
 $\text{match } \Gamma_1 \text{ with}$
 $| \text{Invalid} \Rightarrow \text{Invalid}$
 $| \text{Valid } \Gamma_1' \Rightarrow \text{match } \Gamma_2 \text{ with}$
 $| \text{Invalid} \Rightarrow \text{Invalid}$
 $| \text{Valid } \Gamma_2' \Rightarrow \text{merge}' \Gamma_1' \Gamma_2'$
 end
 $\text{end}.$

Lemma merge_merge' : $\forall (\Gamma_1 \Gamma_2 : \text{Ctx}), \Gamma_1 \uplus \Gamma_2 = (\text{merge}' \Gamma_1 \Gamma_2)$.

Lemma merge_cancel_l : $\forall \Gamma \Gamma_1 \Gamma_2, \Gamma_1 = \Gamma_2 \rightarrow \Gamma \uplus \Gamma_1 = \Gamma \uplus \Gamma_2$.

Lemma merge_cancel_r : $\forall \Gamma \Gamma_1 \Gamma_2, \Gamma_1 = \Gamma_2 \rightarrow \Gamma_1 \uplus \Gamma = \Gamma_2 \uplus \Gamma$.

Lemma merge_I_l : $\forall \Gamma, \text{Invalid} \uplus \Gamma = \text{Invalid}$.

Lemma merge_I_r : $\forall \Gamma, \Gamma \uplus \text{Invalid} = \text{Invalid}$.

Lemma merge_valid : $\forall (\Gamma_1 \Gamma_2 : \text{OCtx}) (\Gamma : \text{Ctx}),$
 $\Gamma_1 \uplus \Gamma_2 = \text{Valid } \Gamma \rightarrow$
 $(\exists \Gamma_1', \Gamma_1 = \text{Valid } \Gamma_1') \wedge (\exists \Gamma_2', \Gamma_2 = \text{Valid } \Gamma_2')$.

Lemma merge_valid : $\forall (\Gamma_1 \Gamma_2 : \text{OCtx}) (\Gamma : \text{Ctx}),$
 $\Gamma_1 \uplus \Gamma_2 = \text{Valid } \Gamma \rightarrow$
 $\{\Gamma_1' : \text{Ctx} \ \& \ \Gamma_1 = \text{Valid } \Gamma_1'\} * \{\Gamma_2' : \text{Ctx} \ \& \ \Gamma_2 = \text{Valid } \Gamma_2'\}$.

Lemma merge_invalid_iff : $\forall (o1 \ o2 : \text{option } \text{WType}) (\Gamma_1 \ \Gamma_2 : \text{Ctx}),$
 $\text{Valid } (o1 :: \Gamma_1) \uplus \text{Valid } (o2 :: \Gamma_2) = \text{Invalid} \leftrightarrow$
 $\text{merge_wire } o1 \ o2 = \text{Invalid} \vee \Gamma_1 \uplus \Gamma_2 = \text{Invalid}$.

Lemma merge_nil_l : $\forall \Gamma, \emptyset \uplus \Gamma = \Gamma$.

Lemma merge_nil_r : $\forall \Gamma, \Gamma \uplus \emptyset = \Gamma$.

Lemma merge_comm : $\forall \Gamma_1 \ \Gamma_2, \Gamma_1 \uplus \Gamma_2 = \Gamma_2 \uplus \Gamma_1$.

Lemma merge_assoc : $\forall \Gamma_1 \ \Gamma_2 \ \Gamma_3, \Gamma_1 \uplus (\Gamma_2 \uplus \Gamma_3) = \Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3$.

Lemma cons_distr_merge : $\forall \Gamma_1 \ \Gamma_2,$
 $\text{cons_o } \text{None } (\Gamma_1 \uplus \Gamma_2) = \text{cons_o } \text{None } \Gamma_1 \uplus \text{cons_o } \text{None } \Gamma_2$.

Lemma merge_nil_inversion' : $\forall (\Gamma_1 \ \Gamma_2 : \text{Ctx}),$
 $\Gamma_1 \uplus \Gamma_2 = \emptyset \rightarrow (\Gamma_1 = []) * (\Gamma_2 = [])$.

Lemma merge_nil_inversion : $\forall (\Gamma_1 \ \Gamma_2 : \text{OCtx}),$
 $\Gamma_1 \uplus \Gamma_2 = \emptyset \rightarrow (\Gamma_1 = \emptyset) * (\Gamma_2 = \emptyset)$.

Lemma ctx_cons_inversion : $\forall (\Gamma \ \Gamma_1 \ \Gamma_2 : \text{Ctx}) \ o \ o1 \ o2,$
 $\text{Valid } (o1 :: \Gamma_1) \uplus \text{Valid } (o2 :: \Gamma_2) = \text{Valid } (o :: \Gamma) \rightarrow$
 $(\Gamma_1 \uplus \Gamma_2 = \text{Valid } \Gamma) * (\text{merge_wire } o1 \ o2 = \text{Valid } [o])$.

Lemma merge_singleton_append : $\forall W (\Gamma : \text{Ctx}),$
 $\Gamma \uplus (\text{singleton } (\text{length } \Gamma) \ W) = \text{Valid } (\Gamma ++ [\text{Some } W])$.

Lemma merge_offset : $\forall (n : \mathbb{N}) (\Gamma_1 \Gamma_2 \Gamma : \text{Ctx}),$
Valid $\Gamma = \Gamma_1 \uplus \Gamma_2 \rightarrow$
Valid (repeat None n ++ Γ_1) \uplus Valid (repeat None n ++ Γ_2) =
Valid (repeat None n ++ Γ).

Lemma valid_valid : $\forall \Gamma, \text{is_valid (Valid } \Gamma).$

Lemma valid_empty : $\text{is_valid } \emptyset.$

Lemma not_valid : $\text{not (is_valid Invalid)}.$

Lemma valid_l : $\forall \Gamma_1 \Gamma_2, \text{is_valid } (\Gamma_1 \uplus \Gamma_2) \rightarrow \text{is_valid } \Gamma_1.$

Lemma valid_r : $\forall \Gamma_1 \Gamma_2, \text{is_valid } (\Gamma_1 \uplus \Gamma_2) \rightarrow \text{is_valid } \Gamma_2.$

Lemma valid_cons : $\forall (o1 o2 : \text{option WType}) (\Gamma_1 \Gamma_2 : \text{Ctx}),$
is_valid (Valid (o1 :: Γ_1) \uplus Valid (o2 :: Γ_2)) \leftrightarrow
(is_valid (merge_wire o1 o2) \wedge is_valid ($\Gamma_1 \uplus \Gamma_2$)).

Lemma valid_join : $\forall \Gamma_1 \Gamma_2 \Gamma_3,$
is_valid ($\Gamma_1 \uplus \Gamma_2$) \rightarrow
is_valid ($\Gamma_1 \uplus \Gamma_3$) \rightarrow is_valid ($\Gamma_2 \uplus \Gamma_3$) \rightarrow is_valid ($\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3$).

Lemma valid_split : $\forall \Gamma_1 \Gamma_2 \Gamma_3, \text{is_valid } (\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3) \rightarrow$
is_valid ($\Gamma_1 \uplus \Gamma_2$) \wedge is_valid ($\Gamma_1 \uplus \Gamma_3$) \wedge is_valid ($\Gamma_2 \uplus \Gamma_3$).

Lemma size_octx_merge : $\forall (\Gamma_1 \Gamma_2 : \text{OCtx}), \text{is_valid } (\Gamma_1 \uplus \Gamma_2) \rightarrow$
size_octx ($\Gamma_1 \uplus \Gamma_2$) = (size_octx Γ_1 + size_octx Γ_2).

Lemma merge_o_ind_fun : $\forall o1 o2 o,$
merge_o o1 o2 o \rightarrow merge_wire o1 o2 = Valid [o].

Lemma merge_ind_fun : $\forall \Gamma_1 \Gamma_2 \Gamma,$
merge_ind $\Gamma_1 \Gamma_2 \Gamma \rightarrow$
 $\Gamma == \Gamma_1 \bullet \Gamma_2.$

Lemma merge_o_fun_ind : $\forall o1 o2 o,$
merge_wire o1 o2 = Valid [o] \rightarrow
merge_o o1 o2 o.

Lemma merge_fun_ind : $\forall \Gamma_1 \Gamma_2 \Gamma,$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
merge_ind $\Gamma_1 \Gamma_2 \Gamma.$

Lemma merge_intersection : $\forall \Gamma_1 \Gamma_2 \Gamma_3 \Gamma_4,$
 is_valid $(\Gamma_1 \uplus \Gamma_2) \rightarrow (\Gamma_1 \uplus \Gamma_2) = (\Gamma_3 \uplus \Gamma_4) \rightarrow$
 $\{ \Gamma_{13} : \text{OCtx} \ \& \ \{ \Gamma_{14} : \text{OCtx} \ \& \ \{ \Gamma_{23} : \text{OCtx} \ \& \ \{ \Gamma_{24} : \text{OCtx} \ \&$
 $\Gamma_1 == \Gamma_{13} \bullet \Gamma_{14} \wedge \Gamma_2 == \Gamma_{23} \bullet \Gamma_{24} \wedge \Gamma_3 == \Gamma_{13} \bullet \Gamma_{23} \wedge \Gamma_4 == \Gamma_{14} \bullet \Gamma_{24}$
 $\} \} \} \}.$

Lemma disjoint_nil_r : $\forall \Gamma, \text{Disjoint } \Gamma \ \emptyset.$

Lemma disjoint_valid : $\forall \Gamma_1 \Gamma_2,$
 Disjoint $\Gamma_1 \Gamma_2 \rightarrow \text{is_valid } \Gamma_1 \rightarrow \text{is_valid } \Gamma_2 \rightarrow$
 is_valid $(\Gamma_1 \uplus \Gamma_2).$

Lemma disjoint_merge : $\forall \Gamma \Gamma_1 \Gamma_2,$
 Disjoint $\Gamma \Gamma_1 \rightarrow \text{Disjoint } \Gamma \Gamma_2 \rightarrow \text{Disjoint } \Gamma (\Gamma_1 \uplus \Gamma_2).$

Lemma disjoint_split : $\forall \Gamma_1 \Gamma_2 \Gamma,$
 is_valid $\Gamma_1 \rightarrow \text{is_valid } \Gamma_2 \rightarrow$
 Disjoint $\Gamma_1 \Gamma_2 \rightarrow \text{Disjoint } (\Gamma_1 \uplus \Gamma_2) \Gamma \rightarrow$
 Disjoint $\Gamma_1 \Gamma \wedge \text{Disjoint } \Gamma_2 \Gamma.$

Lemma index_invalid : $\forall i, \text{index Invalid } i = \text{None}.$

Lemma index_empty : $\forall i, \text{index } \emptyset \ i = \text{None}.$

Lemma singleton_index : $\forall x \ w \ \Gamma, \text{SingletonCtx } x \ w \ \Gamma \rightarrow$
 index $\Gamma \ x = \text{Some } w.$

Lemma empty_ctx_size : $\forall \Gamma, \text{empty_ctx } \Gamma \rightarrow \text{size_ctx } \Gamma = 0.$

Lemma eq_dec_empty_ctx : $\forall \Gamma, \{\text{empty_ctx } \Gamma\} + \{\neg \text{empty_ctx } \Gamma\}.$

Lemma merge_empty : $\forall (\Gamma \ \Gamma_1 \ \Gamma_2 : \text{Ctx}),$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \text{empty_ctx } \Gamma \rightarrow$
 $\text{empty_ctx } \Gamma_1 \wedge \text{empty_ctx } \Gamma_2.$

Lemma trim_otrim : $\forall (\Gamma : \text{Ctx}), \text{Valid } (\text{trim } \Gamma) = \text{otrim } \Gamma.$

Lemma size_ctx_trim : $\forall \Gamma, \text{size_ctx } (\text{trim } \Gamma) = \text{size_ctx } \Gamma.$

Lemma size_octx_trim : $\forall \Gamma, \text{size_octx } (\text{trim } \Gamma) = \text{size_octx } \Gamma.$

Lemma index_trim : $\forall \Gamma \ i,$
 index $(\text{trim } \Gamma) \ i = \text{index } \Gamma \ i.$

Lemma trim_empty : $\forall \Gamma, \text{empty_ctx } \Gamma \rightarrow \text{trim } \Gamma = [].$

Lemma trim_non_empty : $\forall \Gamma, \neg \text{empty_ctx } \Gamma \rightarrow \text{trim } \Gamma \neq []$.

Lemma trim_cons_non_empty : $\forall o \Gamma, \neg \text{empty_ctx } \Gamma \rightarrow \text{trim } (o :: \Gamma) = o :: \text{trim } \Gamma$.

Lemma trim_valid : $\forall (\Gamma : \text{OCtx}), \text{is_valid } \Gamma \leftrightarrow \text{is_valid } (\text{otrim } \Gamma)$.

Lemma trim_merge_dist : $\forall \Gamma_1 \Gamma_2, \text{otrim } \Gamma_1 \uplus \text{otrim } \Gamma_2 = \text{otrim } (\Gamma_1 \uplus \Gamma_2)$.

Lemma trim_merge : $\forall \Gamma \Gamma_1 \Gamma_2, \Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \text{otrim } \Gamma == \text{otrim } \Gamma_1 \bullet \text{otrim } \Gamma_2$.

Lemma merge_dec $\Gamma_1 \Gamma_2$: $\text{is_valid } (\Gamma_1 \uplus \Gamma_2) + \{\Gamma_1 \uplus \Gamma_2 = \text{Invalid}\}$.

Lemma pat_ctx_valid : $\forall \Gamma W (p : \text{Pat } W), \Gamma \vdash p : \text{Pat} \rightarrow \text{is_valid } \Gamma$.

Lemma octx_wtype_size : $\forall w (p : \text{Pat } w) (\Gamma : \text{OCtx}), \Gamma \vdash p : \text{Pat} \rightarrow \text{size_wtype } w = \text{size_octx } \Gamma$.

Lemma ctx_wtype_size : $\forall w (p : \text{Pat } w) (\Gamma : \text{Ctx}), \Gamma \vdash p : \text{Pat} \rightarrow \text{size_wtype } w = \text{size_ctx } \Gamma$.

Lemma size_wtype_length : $\forall \{w\} (p : \text{Pat } w), \text{length } (\text{pat_to_list } p) = \text{size_wtype } w$.

Lemma typed_pat_merge_valid : $\forall W \Gamma \Gamma' (p : \text{Pat } W), (p, \Gamma') = \text{mk_typed_pat } W \Gamma \rightarrow \text{is_valid } (\Gamma \uplus \Gamma')$.

Lemma typed_bit_typed : $\forall \Gamma p \Gamma', (p, \Gamma') = \text{mk_typed_bit } \Gamma \rightarrow \Gamma' \vdash p : \text{Pat}$.

Lemma typed_qubit_typed : $\forall \Gamma p \Gamma', (p, \Gamma') = \text{mk_typed_qubit } \Gamma \rightarrow \Gamma' \vdash p : \text{Pat}$.

Lemma typed_pat_typed : $\forall W \Gamma (p : \text{Pat } W) \Gamma', (p, \Gamma') = \text{mk_typed_pat } W \Gamma \rightarrow \Gamma' \vdash p : \text{Pat}$.

C.7 HOASCircuits.v

Lemma `compose_typing` : $\forall \Gamma_1 \Gamma_1' \Gamma W W' (c : \text{Circuit } W)$
 $(f : \text{Pat } W \rightarrow \text{Circuit } W')$,
 $\Gamma_1 \vdash c : \text{Circ} \rightarrow$
 $\Gamma \vdash f : \text{Fun} \rightarrow$
 $\forall \{pf : \Gamma_1' == \Gamma_1 \bullet \Gamma\}$,
 $\Gamma_1' \vdash \text{compose } c f : \text{Circ}$.

Lemma `unbox_typing` : $\forall \Gamma W_1 W_2 (p : \text{Pat } W_1) (c : \text{Box } W_1 W_2)$,
 $\Gamma \vdash p : \text{Pat} \rightarrow$
 $\text{Typed_Box } c \rightarrow$
 $\Gamma \vdash \text{unbox } c p : \text{Circ}$.

C.8 DBCircuits.v

Lemma `get_fresh_split` : $\forall w \Gamma$,
 $\text{get_fresh } w \Gamma = (\text{get_fresh_pat } w \Gamma, \text{get_fresh_state } w \Gamma)$.

Lemma `get_fresh_merge_valid` : $\forall w \Gamma \Gamma_0 (p : \text{Pat } w)$,
 $(p, \Gamma) = \text{get_fresh } w \Gamma_0 \rightarrow \text{is_valid } (\Gamma_0 \uplus \Gamma)$.

Lemma `get_fresh_typed` : $\forall w \Gamma_0 p \Gamma$,
 $(p, \Gamma) = \text{get_fresh } w \Gamma_0 \rightarrow \Gamma \vdash p : \text{Pat}$.

Lemma `add_fresh_split` : $\forall w \Gamma$,
 $\text{add_fresh } w \Gamma = (\text{add_fresh_pat } w \Gamma, \text{add_fresh_state } w \Gamma)$.

Lemma `add_fresh_state_merge` : $\forall w (\Gamma \Gamma' : \text{Ctx})$,
 $\Gamma' = \text{add_fresh_state } w \Gamma \rightarrow$
 $\text{Valid } \Gamma' = \Gamma \uplus \text{get_fresh_state } w \Gamma$.

Lemma `add_fresh_pat_eq` : $\forall w \Gamma$, $\text{add_fresh_pat } w \Gamma = \text{get_fresh_pat } w \Gamma$.

Lemma `add_fresh_typed` : $\forall w w_0 (p : \text{Pat } w) (p_0 : \text{Pat } w_0) \Gamma \Gamma_0$,
 $(p, \Gamma) = \text{add_fresh } w \Gamma_0 \rightarrow$
 $\Gamma_0 \vdash p_0 : \text{Pat} \rightarrow$
 $\Gamma \vdash (\text{pair } p_0 p) : \text{Pat}$.

Lemma `add_fresh_typed_empty` : $\forall w (p : \text{Pat } w) \Gamma$,
 $(p, \Gamma) = \text{add_fresh } w [] \rightarrow \Gamma \vdash p : \text{Pat}$.

Lemma `maps_to_singleton` : $\forall v W$, $\text{maps_to } v (\text{singleton } v W) = \text{Some } 0$.

Lemma SingletonCtx_dom : $\forall x w \Gamma,$
 SingletonCtx x w $\Gamma \rightarrow$
 ctx_dom $\Gamma = [x]$.

Lemma SingletonCtx_flatten : $\forall x w \Gamma,$
 SingletonCtx x w $\Gamma \rightarrow$
 flatten_ctx $\Gamma = [Some w]$.

Lemma remove_indices_empty : $\forall \Gamma,$ remove_indices $\Gamma [] = \Gamma$.

Lemma remove_indices_merge : $\forall (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}) \text{idxs},$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
 remove_indices $\Gamma \text{idxs} == \text{remove_indices } \Gamma_1 \text{idxs} \bullet \text{remove_indices } \Gamma_2 \text{idxs}.$

Lemma map_unmap : $\forall l,$ map pred (map S l) = l.

Lemma remove_flatten : $\forall \Gamma,$
 remove_indices $\Gamma (\text{get_nones } \Gamma) = \text{flatten_ctx } \Gamma.$

Lemma fmap_S_seq : $\forall \text{len start},$
 fmap S (seq start len) = seq (S start) len.

Lemma seq_S : $\forall \text{len start},$
 seq start (S len) = seq start len ++ [start + len].

C.9 Denotation.v

Lemma WF_Matrix_U : $\forall \{W\} (U : \text{Unitary } W),$
 WF_Matrix $(2^{\llbracket W \rrbracket}) (2^{\llbracket W \rrbracket}) (\llbracket U \rrbracket).$

Lemma unitary_gate_unitary : $\forall \{W\} (U : \text{Unitary } W),$ WF_Unitary $(\llbracket U \rrbracket).$

Lemma denote_unitary_transpose : $\forall \{W\} (U : \text{Unitary } W),$ $\llbracket \text{trans } U \rrbracket = \llbracket U \rrbracket^\dagger.$

Lemma pow_gt_0 : $\forall n,$ $(2^n > 0).$

Lemma WF_denote_gate : $\forall \text{safe } n W_1 W_2 (g : \text{Gate } W_1 W_2) \rho,$
 WF_Matrix $(2^{\llbracket W_1 \rrbracket} * 2^n) (2^{\llbracket W_1 \rrbracket} * 2^n) \rho \rightarrow$
 WF_Matrix $(2^{\llbracket W_2 \rrbracket} * 2^n) (2^{\llbracket W_2 \rrbracket} * 2^n) (\text{denote_gate' safe } n g \rho).$

Lemma discard_qubit_correct : $\forall (\rho : \text{Matrix } 2 \ 2),$ Mixed_State $\rho \rightarrow$
 Mixed_State $(\langle 0 | \times \rho \times | 0 \rangle .+ \langle 1 | \times \rho \times | 1 \rangle).$

Lemma denote_gate_correct : $\forall \{W_1\} \{W_2\} (g : \text{Gate } W_1 \ W_2),$
 WF_Superoperator (denote_gate true g).

Lemma swap_list_swap : swap_list 2 [S 0] = swap.

Lemma WF_pad : $\forall m \ n (A : \text{Square } m),$
 $(m \leq n) \rightarrow$
 WF_Matrix (2^m) (2^m) A \rightarrow
 WF_Matrix (2^n) (2^n) (@pad m n A).

Lemma pad_nothing : $\forall m \ A, \text{@pad } m \ m \ A = A.$

Lemma WF_swap_to_0_aux : $\forall n \ i,$
 $(i + 1 < n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_to_0_aux n i).

Lemma WF_swap_to_0 : $\forall i \ n, (i < n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_to_0 n i).

Lemma WF_swap_two_aux : $\forall n \ i \ j, (i < j < n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_two_aux n i j).

Lemma WF_swap_two : $\forall n \ i \ j, (i < n) \rightarrow (j < n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_two n i j).

Lemma WF_swap_list_aux : $\forall m \ n \ l,$
 $(\forall i \ j, \text{In } (i,j) \ l \rightarrow (i < n) \wedge (j < n)) \rightarrow$
 $(m \leq n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_list_aux m n l).

Lemma WF_swap_list : $\forall n \ l, (\text{length } l \leq n) \rightarrow$
 $(\forall x, \text{In } x \ l \rightarrow x < n) \rightarrow$
 WF_Matrix (2^n) (2^n) (swap_list n l).

Lemma swap_to_0_aux_unitary : $\forall n \ i, (i + 1 < n) \rightarrow$
 WF_Unitary (swap_to_0_aux n i).

Lemma swap_to_0_unitary : $\forall i \ n, (i < n) \rightarrow \text{WF_Unitary } (\text{swap_to_0 } n \ i).$

Lemma swap_two_aux_unitary : $\forall n \ i \ j, (i < j < n) \rightarrow$
 WF_Unitary (swap_two_aux n i j).

Lemma swap_two_unitary : $\forall n \ i \ j, (i < n) \rightarrow (j < n) \rightarrow$
 WF_Unitary (swap_two n i j).

Lemma `swap_list_aux_unitary` : $\forall m n l,$
 $(\forall i j, \text{In } (i,j) l \rightarrow (i < n) \wedge (j < n)) \rightarrow$
 $(m \leq n) \rightarrow$
`WF_Unitary` (`swap_list_aux` $m n l$).

Lemma `swap_list_unitary` : $\forall n l, (\text{length } l \leq n) \rightarrow$
 $(\forall x, \text{In } x l \rightarrow x < n) \rightarrow$
`WF_Unitary` (`swap_list` $n l$).

Lemma `ctrl_list_to_unitary_r_false` : $\forall n (u : \text{Matrix } 2\ 2),$
`ctrl_list_to_unitary_r` (`repeat false` n) $u = u \otimes 'I_ (2^n)$.

Lemma `ctrl_list_to_unitary_false` : $\forall m n (u : \text{Matrix } 2\ 2),$
`WF_Matrix` $2\ 2\ u \rightarrow$
`ctrl_list_to_unitary` (`repeat false` m) (`repeat false` n) $u =$
 $'I_ (2^m) \otimes u \otimes 'I_ (2^n)$.

Lemma `ctrls_to_list_empty` : $\forall W lb u,$
`@ctrls_to_list` $W lb [] u = (0, [], \text{Zero } 2\ 2)$.

Lemma `denote_ctrls_empty` : $\forall W (n : \mathbb{N}) (u : \text{Unitary } W),$
`denote_ctrls` $n u [] = \text{Zero } (2^n) (2^n)$.

Lemma `denote_ctrls_ctrl_u` : $\forall (u : \text{Unitary Qubit}),$
`denote_ctrls` $2 (\text{ctrl } u) [0;1] = (\text{control } (\text{denote } u))$.

Lemma `denote_ctrls_ctrl_u'` : $\forall (u : \text{Unitary Qubit}),$
`denote_ctrls` $2 (\text{ctrl } u) [1;0] = \text{swap} \times (\text{control } (\text{denote } u)) \times \text{swap}$.

Lemma `denote_ctrls_qubit` : $\forall n (u : \text{Unitary Qubit}) k,$
 $(k < n) \rightarrow$
`denote_ctrls` $n u [k] = 'I_ (2^k) \otimes [u] \otimes 'I_ (2^{(n-k-1)})$.

Lemma `ctrl_list_to_unitary_r_unitary` : $\forall r (u : \text{Square } 2), \text{WF_Unitary } u \rightarrow$
`WF_Unitary` (`ctrl_list_to_unitary_r` $r u$).

Lemma `ctrl_list_to_unitary_unitary` : $\forall l r (u : \text{Square } 2), \text{WF_Unitary } u \rightarrow$
`WF_Unitary` (`ctrl_list_to_unitary` $l r u$).

Lemma `ctrls_to_list_spec` : $\forall W l (g : \text{Unitary } W) k lb lb' u,$
 $(\text{length } l = \llbracket W \rrbracket) \rightarrow$
`ctrls_to_list` $lb l g = (k, lb', u) \rightarrow$
`@WF_Unitary` $2 u \wedge \text{length } lb' = \text{length } lb \wedge \text{In } k l$.

Lemma denote_ctrls_unitary : $\forall W n (g : \text{Unitary } W) l,$
 (forallb (fun x \Rightarrow x <? n) l = true) \rightarrow
 (length l = $\llbracket W \rrbracket$) \rightarrow
 WF_Unitary (denote_ctrls n g l).

Lemma denote_ctrls_transpose_qubit : $\forall (n : \mathbb{N}) (u : \text{Unitary Qubit})$
 (li : list \mathbb{N}), denote_ctrls n (trans u) li = (denote_ctrls n u li) † .

Lemma ctrl_to_list_transpose : $\forall W lb li (u : \text{Unitary } W) n lb' u',$
 ctrl_to_list lb li u = (n, lb', u') \rightarrow
 ctrl_to_list lb li (trans u) = (n, lb', u' †).

Lemma ctrl_list_to_unitary_transpose : $\forall l r u,$
 ctrl_list_to_unitary l r (u †) = (ctrl_list_to_unitary l r u) † .

Lemma denote_ctrls_transpose : $\forall W (n : \mathbb{N}) (u : \text{Unitary } W) li,$
 denote_ctrls n (trans u) li = (denote_ctrls n u li) † .

Lemma apply_to_first_correct : $\forall k n (u : \text{Square } 2),$
 WF_Unitary u \rightarrow
 (k < n) \rightarrow
 WF_Superoperator (apply_to_first (@apply_qubit_unitary n u) [k]).

Lemma apply_U_correct : $\forall W n (U : \text{Unitary } W) l,$
 length l = $\llbracket W \rrbracket$ \rightarrow
 (forallb (fun x \Rightarrow x <? n) l = true) \rightarrow
 WF_Superoperator (apply_U n U l).

Lemma discard_superoperator : $\forall (n i j : \mathbb{N}) (\rho : \text{Square } (2 \wedge n)),$
 (i * j * 2 = 2 $^\wedge$ n) \rightarrow
 Mixed_State $\rho \rightarrow$
 Mixed_State (('I_i \otimes $\langle 0| \otimes$ 'I_j) \times $\rho \times$ ('I_i \otimes $|0\rangle \otimes$ 'I_j) .+
 ('I_i \otimes $\langle 1| \otimes$ 'I_j) \times $\rho \times$ ('I_i \otimes $|1\rangle \otimes$ 'I_j)).

Lemma measure_superoperator : $\forall (n i j : \mathbb{N}) (\rho : \text{Square } (2 \wedge n)),$
 (i * j * 2 = 2 $^\wedge$ n) \rightarrow
 Mixed_State $\rho \rightarrow$
 Mixed_State (('I_i \otimes $|0\rangle\langle 0| \otimes$ 'I_j) \times $\rho \times$ ('I_i \otimes $|0\rangle\langle 0| \otimes$ 'I_j) .+
 ('I_i \otimes $|1\rangle\langle 1| \otimes$ 'I_j) \times $\rho \times$ ('I_i \otimes $|1\rangle\langle 1| \otimes$ 'I_j)).

Lemma init_0_superoperator : $\forall (n i j : \mathbb{N}) (\rho : \text{Square } (2 \wedge n)),$
 (i * j = 2 $^\wedge$ n) \rightarrow
 Mixed_State $\rho \rightarrow$
 Mixed_State (('I_i \otimes $|0\rangle \otimes$ 'I_j) \times $\rho \times$ ('I_i \otimes $\langle 0| \otimes$ 'I_j)).


```

Lemma init1_superoperator : ∀ (n i j : ℕ) (ρ : Square (2 ^ n)),
  (i * j = 2^n) →
  Mixed_State ρ →
  Mixed_State (('I_i ⊗ |1⟩ ⊗ 'I_j) × ρ × ('I_i ⊗ ⟨1| ⊗ 'I_j)).

Lemma init0_end_superoperator : ∀ (n i : ℕ) (ρ : Square (2 ^ n)),
  (i = 2^n) →
  Mixed_State ρ →
  Mixed_State (('I_i ⊗ |0⟩) × ρ × ('I_i ⊗ ⟨0|)).

Lemma init1_end_superoperator : ∀ (n i : ℕ) (ρ : Square (2 ^ n)),
  (i = 2^n) →
  Mixed_State ρ →
  Mixed_State (('I_i ⊗ |1⟩) × ρ × ('I_i ⊗ ⟨1|)).

Lemma apply_discard_correct : ∀ n k, (k < n) →
  WF_Superoperator (@apply_discard n k).

Fact apply_meas_correct : ∀ n k, (k < n) →
  WF_Superoperator (@apply_meas n k).

Lemma apply_new0_correct : ∀ n, WF_Superoperator (@apply_new0 n).

Lemma apply_new1_correct : ∀ n, WF_Superoperator (@apply_new1 n).

Lemma apply_gate_correct : ∀ W1 W2 n (g : Gate W1 W2) l,
  length l = [|W1|] →
  length l ≤ n →
  forallb (fun x => x <? n) l = true →
  WF_Superoperator (@apply_gate n W1 W2 true g l).

Lemma map_same_id : ∀ a l,
  map (fun z : ℕ * ℕ => if a =? snd z then (fst z, a) else z) (combine l l)
  = combine l l.

Lemma swap_list_aux_id : ∀ m n l,
  swap_list_aux m n (combine l l) = Id (2 ^ n).

Lemma swap_list_n_id : ∀ n, swap_list n (seq 0 n) = Id (2^n).

Lemma apply_U_σ : ∀ m n (U : Square (2^m)),
  WF_Matrix (2^m) (2^m) U →
  (m ≤ n) →
  @apply_U m n U (σ_{n}) = super (pad n U).

```

Lemma `apply_U_spec_1` : $\forall n i j (A1 : \text{Square } (2^i)) (A2 : \text{Square } (2^j))$
 $(U : \text{Square } (2^1)) (\rho : \text{Square } (2^1)), (i + j + 1 = n) \rightarrow$
 $@\text{apply_U } 1 n U [i] (A1 \otimes \rho \otimes A2) = A1 \otimes (\text{super } U \rho) \otimes A2.$

Lemma `apply_U_spec_2` : $\forall n i j k$
 $(A1 : \text{Square } (2^i)) (A2 : \text{Square } (2^j)) (A3 : \text{Square } (2^k))$
 $(U : \text{Square } (2^2)) (\rho_1 \rho_2 \rho_1' \rho_2' : \text{Square } (2^1)), (i + j + k + 2 = n) \rightarrow$
 $(\text{super } U (\rho_1 \otimes \rho_2)) = \rho_1' \otimes \rho_2' \rightarrow$
 $@\text{apply_U } 2 n U [i; (i+j+1)] (A1 \otimes \rho_1 \otimes A2 \otimes \rho_2 \otimes A3) =$
 $A1 \otimes \rho_1' \otimes A2 \otimes \rho_2' \otimes A3.$

Lemma `denote_output` : $\forall \Gamma_0 \Gamma \{w\} (p : \text{Pat } w),$
 $\langle \Gamma_0 \mid \Gamma \text{ output } p \rangle$
 $= \text{super } (\text{pad } (\llbracket \Gamma_0 \rrbracket + \llbracket \Gamma \rrbracket)) (\text{denote_pat } (\text{subst_pat } \Gamma p)).$

Lemma `length_fresh_state` : $\forall w \Gamma \Gamma',$
 $\Gamma' = \text{add_fresh_state } w \Gamma \rightarrow$
 $\text{length } \Gamma' = (\text{length } \Gamma + \text{size_wtype } w).$

Lemma `swap_fresh_seq` : $\forall w (\Gamma : \text{Ctx}),$
 $\text{pat_to_list } (\text{add_fresh_pat } w \Gamma) = \text{seq } (\text{length } \Gamma) (\text{size_wtype } w).$

Lemma `denote_pat_fresh_id` : $\forall w,$
 $\text{denote_pat } (\text{add_fresh_pat } w []) = \text{Id } (2^{\llbracket w \rrbracket}).$

Lemma `maps_to_app` : $\forall \Gamma w,$
 $\text{maps_to } (\text{length } \Gamma) (\Gamma ++ [\text{Some } w]) = \text{Some } (\text{size_ctx } \Gamma).$

Lemma `no_gaps_size` : $\forall \Gamma, \text{no_gaps } \Gamma \rightarrow \text{size_ctx } \Gamma = \text{length } \Gamma.$

Lemma `size_ctx_le_length` : $\forall \Gamma, (\text{size_ctx } \Gamma \leq \text{length } \Gamma).$

Lemma `size_eq_no_gaps` : $\forall \Gamma, \text{size_ctx } \Gamma = \text{length } \Gamma \rightarrow \text{no_gaps } \Gamma.$

Lemma `no_gaps_app` : $\forall \Gamma \Gamma', \text{no_gaps } \Gamma \rightarrow \text{no_gaps } \Gamma' \rightarrow \text{no_gaps } (\Gamma ++ \Gamma').$

Lemma `add_fresh_state_no_gaps` : $\forall W \Gamma,$
 $\text{no_gaps } \Gamma \rightarrow \text{no_gaps } (\text{add_fresh_state } W \Gamma).$

Lemma `bounded_pat_le` : $\forall W (p : \text{Pat } W) n n',$
 $(n \leq n') \rightarrow \text{Bounded_Pat } n p \rightarrow \text{Bounded_Pat } n' p.$

Lemma `add_fresh_pat_bounded` : $\forall W \Gamma,$
 $\text{no_gaps } \Gamma \rightarrow$
 $\text{Bounded_Pat } (\text{length } \Gamma + \text{size_wtype } W) (\text{add_fresh_pat } W \Gamma).$

Lemma subst_var_no_gaps : $\forall \Gamma w,$
no_gaps $\Gamma \rightarrow$
subst_var ($\Gamma ++ [\text{Some } w]$) (length Γ) = length Γ .

Lemma maps_to_no_gaps : $\forall v \Gamma,$
($v < \text{length } \Gamma$) \rightarrow
no_gaps $\Gamma \rightarrow$
maps_to $v \Gamma = \text{Some } v$.

Lemma subst_var_no_gaps : $\forall \Gamma v,$
($v < \text{length } \Gamma$) \rightarrow
no_gaps $\Gamma \rightarrow$
subst_var $\Gamma v = v$.

Lemma subst_pat_no_gaps : $\forall w (\Gamma : \text{Ctx}) (p : \text{Pat } w),$
Bounded_Pat (length Γ) $p \rightarrow$
no_gaps $\Gamma \rightarrow$
subst_pat $\Gamma p = p$.

Lemma subst_units : $\forall w (p : \text{Pat } w) \Gamma, \emptyset \vdash p : \text{Pat} \rightarrow \text{subst_pat } \Gamma p = p$.

Lemma subst_pat_fresh : $\forall w \Gamma,$
no_gaps $\Gamma \rightarrow$
subst_pat (add_fresh_state $w \Gamma$) (add_fresh_pat $w \Gamma$)
= add_fresh_pat $w \Gamma$.

Lemma subst_pat_fresh_empty : $\forall w,$
subst_pat (add_fresh_state $w []$) (add_fresh_pat $w []$)
= add_fresh_pat $w []$.

Lemma size_fresh_ctx : $\forall (w : \text{WType}) (\Gamma : \text{Ctx}),$
size_ctx (add_fresh_state $w \Gamma$) = (size_ctx $\Gamma + \text{size_wtype } w$).

Lemma denote_db_unbox : $\forall \{w1 w2\} (b : \text{Box } w1 w2),$
 $\llbracket b \rrbracket = \langle [] \mid \text{add_fresh_state } w1 [] \text{ unbox } b (\text{add_fresh_pat } w1 []) \rangle$.

Lemma denote_index_update_some : $\forall (\Gamma : \text{Ctx}) x w w',$
index (Valid Γ) $x = \text{Some } w \rightarrow$
 $\llbracket \text{update_at } \Gamma x (\text{Some } w') \rrbracket = \llbracket \Gamma \rrbracket$.

Lemma denote_index_update_none : $\forall (\Gamma : \text{Ctx}) x w,$
index (Valid Γ) $x = \text{Some } w \rightarrow$
 $\llbracket \text{update_at } \Gamma x \text{None} \rrbracket = (\llbracket \Gamma \rrbracket - 1)$.

Lemma singleton_update : $\forall \Gamma W W' v,$
 SingletonCtx $v W \Gamma \rightarrow$
 SingletonCtx $v W' (\text{update_at } \Gamma v (\text{Some } W'))$.

Lemma remove_at_singleton : $\forall x w \Gamma,$
 SingletonCtx $x w \Gamma \rightarrow$
 empty_ctx (remove_at $x \Gamma$).

Lemma update_none_singleton : $\forall x w \Gamma,$
 SingletonCtx $x w \Gamma \rightarrow$
 empty_ctx (update_at $\Gamma x \text{None}$).

Lemma remove_pat_singleton : $\forall x W (p : \text{Pat } W),$
 singleton $x W \vdash p:\text{Pat} \rightarrow$
 remove_pat $p (\text{singleton } x W) = []$.

Lemma index_merge_l : $\forall \Gamma \Gamma_1 \Gamma_2 n w,$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
 index $\Gamma_1 n = \text{Some } w \rightarrow$
 index $\Gamma n = \text{Some } w$.

Lemma index_merge_r : $\forall \Gamma \Gamma_1 \Gamma_2 n w,$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
 index $\Gamma_2 n = \text{Some } w \rightarrow$
 index $\Gamma n = \text{Some } w$.

Lemma remove_at_merge : $\forall (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}) n, \Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
 Valid (remove_at $n \Gamma$) == Valid (remove_at $n \Gamma_1$) • Valid (remove_at $n \Gamma_2$).

Lemma update_none_merge : $\forall (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}) n, \Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
 Valid (update_at $\Gamma n \text{None}$) ==
 Valid (update_at $\Gamma_1 n \text{None}$) • Valid (update_at $\Gamma_2 n \text{None}$).

Lemma remove_at_collision : $\forall n W (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}),$
 SingletonCtx $n W \Gamma_1 \rightarrow$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \text{size_ctx } (\text{remove_at } n \Gamma_2) = \text{size_ctx } \Gamma_2$.

Lemma update_none_collision : $\forall n W (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}),$
 SingletonCtx $n W \Gamma_1 \rightarrow$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \text{size_ctx } (\text{update_at } \Gamma_2 n \text{None}) = \text{size_ctx } \Gamma_2$.

Fact process_gate_ctx_size : $\forall w1 w2 (\Gamma \Gamma_1 \Gamma_2 : \text{Ctx}) (g : \text{Gate } w1 w2)$
 (p1 : Pat $w1$),
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \Gamma_1 \vdash p1 : \text{Pat} \rightarrow$
 (size_ctx (process_gate_state $g p1 \Gamma$)) = (size_ctx $\Gamma + \llbracket w2 \rrbracket - \llbracket w1 \rrbracket$).

Lemma denote_gate_circuit : $\forall \{w1\ w2\ w'\} \text{ (safe:}\mathbb{B}\text{)} (g : \text{Gate } w1\ w2)$
 $(p1 : \text{Pat } w1) (f : \text{Pat } w2 \rightarrow \text{Circuit } w')$ $(\Gamma_0\ \Gamma\ \Gamma_1\ \Gamma_2 : \text{Ctx}),$
 $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow \Gamma_1 \vdash p1 : \text{Pat} \rightarrow$
denote_circuit safe (gate g p1 f) $\Gamma_0\ \Gamma =$
compose_super
(denote_circuit safe (f (process_gate_pat g p1 Γ)) Γ_0
(process_gate_state g p1 Γ))
(apply_gate safe g (pat_to_list (subst_pat Γ p1))).

Lemma lookup_maybe_S : $\forall x\ l,$
lookup_maybe (S x) (map S l) = lookup_maybe x l.

Lemma subst_qubit_bounded : $\forall v\ (\Gamma\ \Gamma_v\ \Gamma_o : \text{Ctx}),$
 $\Gamma_v \vdash (\text{qubit } v) : \text{Pat} \rightarrow$
 $\Gamma == \Gamma_v \bullet \Gamma_o \rightarrow$
(subst_var $\Gamma\ v < \text{size_octx } \Gamma$).

Lemma subst_bit_bounded : $\forall v\ (\Gamma\ \Gamma_v\ \Gamma_o : \text{Ctx}),$
 $\Gamma_v \vdash (\text{bit } v) : \text{Pat} \rightarrow$
 $\Gamma == \Gamma_v \bullet \Gamma_o \rightarrow$
(subst_var $\Gamma\ v < \text{size_octx } \Gamma$).

Lemma pat_to_list_bounded : $\forall W\ (\Gamma\ \Gamma_p\ \Gamma_o : \text{Ctx}) (p : \text{Pat } W) x,$
 $\Gamma == \Gamma_p \bullet \Gamma_o \rightarrow$
 $\Gamma_p \vdash p : \text{Pat} \rightarrow$
In x (pat_to_list (subst_pat Γ p)) \rightarrow
 $(x < \text{size_ctx } \Gamma)$.

Lemma update_at_singleton : $\forall v\ W\ W'\ \Gamma\ \Gamma',$
SingletonCtx v W $\Gamma \rightarrow$
SingletonCtx v W' $\Gamma' \rightarrow$
update_at $\Gamma\ v$ (Some W') = Γ' .

Lemma update_at_merge : $\forall v\ W\ W'\ \Gamma\ \Gamma_1\ \Gamma_1'\ \Gamma_2,$
SingletonCtx v W $\Gamma_1 \rightarrow$
SingletonCtx v W' $\Gamma_1' \rightarrow$
Valid $\Gamma == \Gamma_1 \bullet \Gamma_2 \rightarrow$
Valid (update_at $\Gamma\ v$ (Some W')) == $\Gamma_1' \bullet \Gamma_2$.

Lemma types_pat_no_trail : $\forall \Gamma\ W (p : \text{Pat } W),$
Valid $\Gamma \vdash p : \text{Pat} \rightarrow \text{trim } \Gamma = \Gamma$.

Lemma remove_bit_merge' : $\forall (\Gamma\ \Gamma' : \text{Ctx}) v,$
 $\Gamma' == \text{singleton } v\ \text{Bit} \bullet \Gamma \rightarrow \text{remove_pat } (\text{bit } v)\ \Gamma' = \text{trim } \Gamma$.

Lemma `remove_bit_merge` : $\forall (\Gamma \Gamma' : \text{Ctx}) W (p : \text{Pat } W) v,$
 $\Gamma \vdash p:\text{Pat} \rightarrow$
 $\Gamma' == \text{singleton } v \text{ Bit} \bullet \Gamma \rightarrow$
`remove_pat (bit v) Γ' = Γ .`

Lemma `remove_qubit_merge'` : $\forall (\Gamma \Gamma' : \text{Ctx}) v,$
 $\Gamma' == \text{singleton } v \text{ Qubit} \bullet \Gamma \rightarrow$
`remove_pat (qubit v) Γ' = trim Γ .`

Lemma `remove_qubit_merge` : $\forall (\Gamma \Gamma' : \text{Ctx}) W (p : \text{Pat } W) v,$
 $\Gamma \vdash p:\text{Pat} \rightarrow$
 $\Gamma' == \text{singleton } v \text{ Qubit} \bullet \Gamma \rightarrow$
`remove_pat (qubit v) Γ' = Γ .`

Lemma `remove_bit_pred` : $\forall (\Gamma \Gamma' : \text{Ctx}) v,$
 $\Gamma' == (\text{singleton } v \text{ Bit}) \bullet \Gamma \rightarrow$
`size_ctx (remove_pat (bit v) Γ') = (size_ctx Γ' - 1).`

Lemma `remove_qubit_pred` : $\forall (\Gamma \Gamma' : \text{Ctx}) v,$
 $\Gamma' == (\text{singleton } v \text{ Qubit}) \bullet \Gamma \rightarrow$
`size_ctx (remove_pat (qubit v) Γ') = (size_ctx Γ' - 1).`

Lemma `maps_to_trim` : $\forall \Gamma v, \text{maps_to } v (\text{trim } \Gamma) = \text{maps_to } v \Gamma.$

Lemma `subst_pat_trim` : $\forall W \Gamma (p : \text{Pat } W),$
`subst_pat (trim Γ) p = subst_pat Γ p.`

Lemma `trim_types_circ` : $\forall W (c : \text{Circuit } W) (\Gamma : \text{Ctx}), \Gamma \vdash c : \text{Circ} \rightarrow$
`trim Γ \vdash c :Circ.`

Theorem `denote_static_circuit_correct` : $\forall W (\Gamma_0 \Gamma : \text{Ctx}) (c : \text{Circuit } W),$
`Static_Circuit c \rightarrow`
 $\Gamma \vdash c:\text{Circ} \rightarrow$
`WF_Superoperator ($\langle \Gamma_0 \mid \Gamma \quad c \rangle$).`

Theorem `denote_static_box_correct` : $\forall W_1 W_2 (c : \text{Box } W_1 W_2),$
`Static_Box c \rightarrow`
`Typed_Box c \rightarrow`
`WF_Superoperator ($\llbracket c \rrbracket$).`

Lemma `inSeq_id_l` : $\forall w_1 w_2 (c : \text{Box } w_1 w_2), \text{id_circ} \cdot c = c.$

Lemma `inSeq_id_r` : $\forall w_1 w_2 (c : \text{Box } w_1 w_2), c \cdot \text{id_circ} = c.$

Lemma HOAS_Equiv_refl : $\forall w_1 w_2 (c : \text{Box } w_1 w_2), c \equiv c$.

Lemma HOAS_Equiv_sym : $\forall w_1 w_2 (c_1 c_2 : \text{Box } w_1 w_2), (c_1 \equiv c_2) \rightarrow c_2 \equiv c_1$.

Lemma HOAS_Equiv_trans : $\forall w_1 w_2 (c_1 c_2 c_3 : \text{Box } w_1 w_2),$
 $(c_1 \equiv c_2) \rightarrow (c_2 \equiv c_3) \rightarrow c_1 \equiv c_3$.

Lemma inSeq_assoc : $\forall \{w_1 w_2 w_3 w_4\} (c_1 : \text{Box } w_1 w_2) (c_2 : \text{Box } w_2 w_3)$
 $(c_3 : \text{Box } w_3 w_4), c_3 \cdot (c_2 \cdot c_1) = (c_3 \cdot c_2) \cdot c_1$.

C.10 HOASLib.v

Lemma boxed_gate_WT $\{W_1 W_2\} (g : \text{Gate } W_1 W_2) : \text{Typed_Box } (\text{boxed_gate } g)$.

Lemma types_circuit_valid : $\forall w (c : \text{Circuit } w) \Gamma, \Gamma \vdash c : \text{Circ} \rightarrow \text{is_valid}$
 Γ .

Lemma apply_box_WT : $\forall w_1 w_2 (b : \text{Box } w_1 w_2) (c : \text{Circuit } w_1) \Gamma,$
 $\text{Typed_Box } b \rightarrow \Gamma \vdash c : \text{Circ} \rightarrow \Gamma \vdash \text{apply_box } b c : \text{Circ}$.

Lemma id_circ_WT : $\forall W, \text{Typed_Box } (@\text{id_circ } W)$.

Lemma WT_SWAP : $\text{Typed_Box } \text{SWAP}$.

Lemma new_WT : $\forall b, \text{Typed_Box } (\text{new } b)$.

Lemma init_WT : $\forall b, \text{Typed_Box } (\text{init } b)$.

Lemma assert_WT : $\forall b, \text{Typed_Box } (\text{assert } b)$.

Lemma inSeq_WT : $\forall W_1 W_2 W_2 (c_1 : \text{Box } W_1 W_2) (c_2 : \text{Box } W_2 W_2),$
 $\text{Typed_Box } c_1 \rightarrow \text{Typed_Box } c_2 \rightarrow \text{Typed_Box } (\text{inSeq } c_1 c_2)$.

Lemma inPar_WT : $\forall W_1 W_1' W_2 W_2' (c_1 : \text{Box } W_1 W_2) (c_2 : \text{Box } W_1' W_2'),$
 $\text{Typed_Box } c_1 \rightarrow \text{Typed_Box } c_2 \rightarrow$
 $\text{Typed_Box } (\text{inPar } c_1 c_2)$.

Lemma types_units : $\forall n, \text{Types_Pat } \emptyset (\text{units } n)$.

Lemma initMany_WT : $\forall b n, \text{Typed_Box } (\text{initMany } b n)$.

Lemma inSeqMany_WT : $\forall n W (c : \text{Box } W W),$
 $\text{Typed_Box } c \rightarrow \text{Typed_Box } (\text{inSeqMany } n c)$.

Lemma `inParMany_WT` : $\forall n W W' (c : \text{Box } W W'), \text{Typed_Box } c \rightarrow \text{Typed_Box } (\text{inParMany } n c)$.

Lemma `CL_AND_WT` : `Typed_Box CL_AND`.

Lemma `CL_XOR_WT` : `Typed_Box CL_XOR`.

Lemma `CL_OR_WT` : `Typed_Box CL_OR`.

Lemma `TRUE_WT` : `Typed_Box TRUE`.

Lemma `FALSE_WT` : `Typed_Box FALSE`.

Lemma `NOT_WT` : `Typed_Box NOT`.

Lemma `AND_WT` : `Typed_Box AND`.

Lemma `XOR_WT` : `Typed_Box XOR`.

C.11 SemanticLib.v

Lemma `id_circ_spec` : $\forall W \rho \text{ safe}, \text{WF_Matrix } (2^{\llbracket W \rrbracket}) (2^{\llbracket W \rrbracket}) \rho \rightarrow \text{denote_box safe } (@\text{id_circ } W) \rho = \rho$.

Lemma `X_spec` : $\forall (b \text{ safe} : \mathbb{B}), \text{denote_box safe } (\text{boxed_gate } _X) (\text{bool_to_matrix } b) = \text{bool_to_matrix } (\neg b)$.

Lemma `init0_spec` : $\forall \text{ safe}, \text{denote_box safe } \text{init}_0 (\text{Id } (2^0)) = |0\rangle\langle 0|$.

Lemma `init1_spec` : $\forall \text{ safe}, \text{denote_box safe } \text{init}_1 (\text{Id } (2^0)) = |1\rangle\langle 1|$.

Lemma `assert0_spec` : $\forall \text{ safe}, \text{denote_box safe } \text{assert}_0 |0\rangle\langle 0| = \text{Id } 1$.

Lemma `assert1_spec` : $\forall \text{ safe}, \text{denote_box safe } \text{assert}_1 |1\rangle\langle 1| = \text{Id } 1$.

Lemma `init_spec` : $\forall b \text{ safe}, \text{denote_box safe } (\text{init } b) (\text{Id } (2^0)) = \text{bool_to_matrix } b$.

Lemma `assert_spec` : $\forall b \text{ safe}, \text{denote_box safe } (\text{assert } b) (\text{bool_to_matrix } b) = \text{Id } 1$.

Lemma CNOT_spec : \forall (b1 b2 safe : \mathbb{B}),
denote_box safe CNOT (bool_to_matrix b1 \otimes bool_to_matrix b2) =
bool_to_matrix b1 \otimes bool_to_matrix (b1 \oplus b2).

Lemma TRUE_spec : \forall z safe,
denote_box safe TRUE (bool_to_matrix z) = bool_to_matrix (true \oplus z).

Lemma FALSE_spec : \forall z safe,
denote_box safe FALSE (bool_to_matrix z) = bool_to_matrix (false \oplus z).

Lemma NOT_spec : \forall (x z : \mathbb{B}),
 \forall safe, denote_box safe NOT (bool_to_matrix x \otimes bool_to_matrix z) =
bool_to_matrix x \otimes bool_to_matrix (\neg x \oplus z).

Lemma XOR_spec : \forall (x y z safe : \mathbb{B}),
denote_box safe XOR
(bool_to_matrix x \otimes bool_to_matrix y \otimes bool_to_matrix z) =
bool_to_matrix x \otimes bool_to_matrix y \otimes bool_to_matrix (x \oplus y \oplus z).

Lemma AND_spec : \forall (x y z safe : \mathbb{B}),
denote_box safe AND
(bool_to_matrix x \otimes bool_to_matrix y \otimes bool_to_matrix z) =
bool_to_matrix x \otimes bool_to_matrix y \otimes bool_to_matrix ((x && y) \oplus z).

C.12 HOASExamples.v

Lemma new_discard_WT : Typed_Box new_discard.

Lemma init_discard_WT : Typed_Box init_discard.

Lemma hadamard_measure_WT : Typed_Box hadamard_measure.

Lemma U_deutsch_WT : \forall U_f , Typed_Box (U_deutsch U_f).

Lemma lift_deutsch_WT : \forall U_f , Typed_Box $U_f \rightarrow$ Typed_Box (lift_deutsch U_f).

Lemma deutsch_WF : \forall U_f , Typed_Box $U_f \rightarrow$ Typed_Box (deutsch U_f).

Lemma deutsch_basic_eq : \forall U_f , deutsch_basic U_f = deutsch U_f .

Lemma Deutsch_Jozsa_WT : \forall n U_f , Typed_Box $U_f \rightarrow$
Typed_Box (Deutsch_Jozsa n U_f).

Lemma Deutsch_Jozsa_WT' : $\forall n U_f, \text{Typed_Box } U_f \rightarrow \text{Typed_Box } (\text{Deutsch_Jozsa } n U_f)$.

Lemma bell100_WT : $\text{Typed_Box } \text{bell100}$.

Lemma bell_old_style_WT : $\text{Typed_Box } \text{bell_old_style}$.

Lemma bell_one_line_WT : $\text{Typed_Box } \text{bell_one_line}$.

Lemma alice_WT : $\text{Typed_Box } \text{alice}$.

Lemma bob_WT : $\text{Typed_Box } \text{bob}$.

Lemma teleport_WT : $\text{Typed_Box } \text{teleport}$.

Lemma bob_lift_WT : $\text{Typed_Box } \text{bob_lift}$.

Lemma bob_lift_WT' : $\text{Typed_Box } \text{bob_lift}'$.

Lemma teleport_lift_WT : $\text{Typed_Box } \text{teleport_lift}$.

Lemma bob_distant_WT : $\forall b1 b2, \text{Typed_Box } (\text{bob_distant } b1 b2)$.

Lemma teleport_distant_WT : $\text{Typed_Box } \text{teleport_distant}$.

Lemma superdense_WT : $\text{Typed_Box } \text{superdense}$.

Lemma superdense_distant_WT : $\forall b1 b2, \text{Typed_Box } (\text{superdense_distant } b1 b2)$.

Lemma rotations_WT : $\forall n m, \text{Typed_Box } (\text{rotations } n m)$.

Lemma qft_WT : $\forall n, \text{Typed_Box } (\text{qft } n)$.

Lemma coin_flip_WT : $\text{Typed_Box } \text{coin_flip}$.

Lemma coin_flips_WT : $\forall n, \text{Typed_Box } (\text{coin_flips } n)$.

Lemma coin_flips_lift_WT : $\forall n, \text{Typed_Box } (\text{coin_flips_lift } n)$.

Lemma coin_flips_lift_WT : $\forall n, \text{Typed_Box } (\text{coin_flips_lift } n)$.

Lemma n_coins_WT : $\forall n, \text{Typed_Box } (\text{n_coins } n)$.

Lemma `n_coins_WT'` : $\forall n, \text{Typed_Box } (n_coins' n)$.

Lemma `unitary_transpose_WT` : $\forall W (U : \text{Unitary } W), \text{Typed_Box } (\text{unitary_transpose } U)$.

Lemma `prepare_basis_WT` : $\forall li, \text{Typed_Box } (\text{prepare_basis } li)$.

Lemma `share_WT` : $\forall n, \text{Typed_Box } (\text{share } n)$.

Lemma `lift_eta_bit_WT` : $\text{Typed_Box } \text{lift_eta}$.

Lemma `lift_meas_WT` : $\text{Typed_Box } \text{lift_meas}$.

Lemma `AND_WT` : $\text{Typed_Box } \text{AND}$.

Lemma `XOR_WT` : $\text{Typed_Box } \text{XOR}$.

Lemma `OR_WT` : $\text{Typed_Box } \text{OR}$.

C.13 HOASProofs.v

Lemma `init_ket1` : $\llbracket \text{init true} \rrbracket I1 = (|1\rangle\langle 1| : \text{Density } 2)$.

Lemma `unitary_transpose_id_qubit` : $\forall (U : \text{Unitary } \text{Qubit}), \text{unitary_transpose } U \equiv \text{id_circ}$.

Lemma `apply_U_trans` : $\forall n W (U : \text{Unitary } W) li, \text{compose_super } (\text{apply_U } n U li) (\text{apply_U } n (\text{trans } U) li) = \text{fun } x \Rightarrow x$.

Lemma `unitary_transpose_id` : $\forall W (U : \text{Unitary } W), \text{unitary_transpose } U \equiv \text{id_circ}$.

Lemma `bias1` : $\text{biased_coin } 1 = |1\rangle\langle 1|$.

Lemma `even_bias` : $\text{biased_coin } (1/2) = \text{fair_coin}$.

Lemma `fair_toss` : $\llbracket \text{coin_flip} \rrbracket I1 = \text{fair_coin}$.

Lemma `wf_biased_coin` : $\forall c, \text{WF_Matrix } 2\ 2 (\text{biased_coin } c)$.

Lemma `flips_lift_correct_gen` : $\forall n a, \llbracket \text{coin_flips_lift } n \rrbracket (a .* I1) = a .* \text{biased_coin } (1/(2^n))$.

Lemma `flips_lift_correct` : $\forall n,$
 $\llbracket \text{coin_flips_lift } n \rrbracket \text{ I1} = \text{biased_coin } (1/(2^n)).$

Lemma `wf_prep` : $\forall \alpha \beta,$ `WF_Matrix 2 2` `(prep $\alpha \beta$).`

Lemma `f2_WF` : `WF_Matrix 4 4` `f2.`

Lemma `U_deutsch_constant` : $\forall U_f,$ `U_constant` `U_f` \rightarrow
 $\llbracket \text{U_deutsch } U_f \rrbracket \text{ I1} = |0\rangle\langle 0|.$

Lemma `U_deutsch_balanced` : $\forall U_f,$ `U_balanced` `U_f` \rightarrow
 $\llbracket \text{U_deutsch } U_f \rrbracket \text{ I1} = |1\rangle\langle 1|.$

Lemma `deutsch_constant` : $\forall f,$ `constant` `f` \rightarrow
 $\llbracket \text{deutsch } (\text{fun_to_box } f) \rrbracket \text{ I1} = |0\rangle\langle 0|.$

Lemma `deutsch_balanced` : $\forall f,$ `balanced` `f` \rightarrow
 $\llbracket \text{deutsch } (\text{fun_to_box } f) \rrbracket \text{ I1} = |1\rangle\langle 1|.$

Lemma `bell00_spec` : $\llbracket \text{bell00} \rrbracket \text{ I1} = \text{EPR00}.$

Lemma `alice_spec` : $\forall (\rho : \text{Density } 4),$ `WF_Matrix 4 4` `ρ` \rightarrow
 $\llbracket \text{alice} \rrbracket \rho = \text{M_alice } \rho.$

Lemma `bob_spec` : $\forall (\rho : \text{Density } 8),$ `WF_Matrix 8 8` `ρ` \rightarrow
 $\llbracket \text{bob} \rrbracket \rho = \text{M_bob } \rho.$

Lemma `teleport_eq` : `teleport` \equiv `id_circ.`

Lemma `superdense_eq` : $\forall (\rho : \text{Density } 4),$
`Classical` `ρ` \rightarrow
`WF_Matrix 4 4` `ρ` \rightarrow
 $\llbracket \text{superdense} \rrbracket \rho = \rho.$

Lemma `superdense_distant_eq` : $\forall b1 b2,$
 $\llbracket \text{superdense_distant } b1 b2 \rrbracket \text{ I1} = \text{bools_to_matrix } [b1; b2].$

C.14 Equations.v

Lemma `X_meas_WT` : `Typed_Box` `X_meas.`

Lemma `meas_NOT_WT` : `Typed_Box` `meas_NOT.`

Lemma NOT_meas_comm : X_meas \equiv meas_NOT.

Lemma lift_UV_WT : $\forall W (U V : \text{Unitary } W), \text{Typed_Box } (\text{lift_UV } U V)$.

Lemma alternate_WT : $\forall W (U V : \text{Unitary } W), \text{Typed_Box } (\text{alternate } U V)$.

Lemma alt_UV_WT : $\forall W (U V : \text{Unitary } W), \text{Typed_Box } (\text{alt_UV } U V)$.

Lemma U_meas_discard_WT : $\forall (U : \text{Unitary Qubit}), \text{Typed_Box } (\text{U_meas_discard } U)$.

Lemma meas_discard_WT : $\text{Typed_Box } \text{meas_discard}$.

Lemma U_meas_eq_meas : $\forall U, \text{U_meas_discard } U \equiv \text{meas_discard}$.

Lemma init_meas_WT : $\forall b, \text{Typed_Box } (\text{init_meas } b)$.

Lemma init_alt_WT : $\forall W b (U V : \text{Unitary } W), \text{Typed_Box } (\text{init_alt } b U V)$.

Lemma init_if_WT : $\forall W b (U V : \text{Unitary } W), \text{Typed_Box } (\text{init_if } b U V)$.

Lemma init_if_true_qubit : $\forall (U V : \text{Unitary Qubit}) \rho, \text{WF_Matrix } 2\ 2\ \rho \rightarrow \llbracket \text{init_if true } U V \rrbracket \rho = (|1\rangle \otimes 'I_2) \times (\llbracket U \rrbracket \times \rho \times (\llbracket U \rrbracket^\dagger)) \times (|1\rangle \otimes 'I_2)$.

Lemma init_if_false_qubit : $\forall (U V : \text{Unitary Qubit}) \rho, \text{WF_Matrix } 2\ 2\ \rho \rightarrow \llbracket \text{init_if false } U V \rrbracket \rho = (|0\rangle \otimes 'I_2) \times (\llbracket V \rrbracket \times \rho \times (\llbracket V \rrbracket^\dagger)) \times (|0\rangle \otimes 'I_2)$.

Lemma init_alt_if_qubit : $\forall b (U V : \text{Unitary Qubit}), \text{init_alt } b U V \equiv \text{init_if } b U V$.

Lemma init_X_WT : $\forall b, \text{Typed_Box } (\text{init_X } b)$.

Lemma init_X_init : $\forall b, \text{init_X } b \equiv \text{init } (\text{negb } b)$.

Lemma new_WT : $\forall b, \text{Typed_Box } (\text{new } b)$.

Lemma lift_new_WT : $\text{Typed_Box } \text{lift_new}$.

Lemma lift_new_new : $\forall (\rho : \text{Density } 2), \text{WF_Matrix } 2\ 2\ \rho \rightarrow \text{Classical } \rho \rightarrow \llbracket \text{lift_new} \rrbracket \rho = \llbracket @\text{id_circ Bit} \rrbracket \rho$.

Lemma meas_lift_new_WT : $\text{Typed_Box } \text{meas_lift_new}$.

Lemma meas_lift_new_new : $\text{meas_lift_new} \equiv \text{boxed_gate } \text{meas}$.

Lemma `super_super` : $\forall m n o (U : \text{Matrix } o n) (V : \text{Matrix } n m)$
 $(\rho : \text{Square } m),$
 $\text{super } U (\text{super } V \rho) = \text{super } (U \times V) \rho.$

Lemma `super_eq` : $\forall m n (U U' : \text{Matrix } m n) (\rho \rho' : \text{Square } n),$
 $U = U' \rightarrow \rho = \rho' \rightarrow$
 $\text{super } U \rho = \text{super } U' \rho'.$

Lemma `HH_CNOT_HH_eq_NOTC` : `HH_CNOT_HH` \equiv `NOTC`.

Lemma `HZH_X` : `HZH` \equiv `_X`.

C.15 Composition.v

Fact `denote_compose` : $\forall \text{ safe } w (c : \text{Circuit } w) (\Gamma : \text{Ctx}),$
 $\Gamma \vdash c : \text{Circ} \rightarrow$
 $\forall w' (f : \text{Pat } w \rightarrow \text{Circuit } w') (\Gamma_0 \Gamma_1 \Gamma_1' \Gamma_{01} : \text{Ctx}),$
 $\Gamma_1 \vdash f : \text{Fun} \rightarrow$
 $\Gamma_1' == \Gamma_1 \bullet \Gamma \rightarrow$
 $\Gamma_{01} == \Gamma_0 \bullet \Gamma_1 \rightarrow$
 $\text{denote_circuit safe (compose } c f) \Gamma_0 \Gamma_1' =$
 compose_super
 $(\text{denote_circuit safe (f (add_fresh_pat } w \Gamma_1)) \Gamma_0 (\text{add_fresh_state } w \Gamma_1))$
 $(\text{denote_circuit safe } c \Gamma_{01} \Gamma).$

Theorem `inSeq_correct` : $\forall W_1 W_2 W_2' (g : \text{Box } W_2 W_2') (f : \text{Box } W_1 W_2)$
 $(\text{safe} : \mathbb{B}), \text{Typed_Box } g \rightarrow \text{Typed_Box } f \rightarrow$
 $\text{denote_box safe (inSeq } f g) =$
 $\text{compose_super (denote_box safe } g) (\text{denote_box safe } f).$

Fact `inPar_correct` : $\forall W_1 W_1' W_2 W_2' (f : \text{Box } W_1 W_1') (g : \text{Box } W_2 W_2')$
 $(\text{safe} : \mathbb{B}) (\rho_1 : \text{Square } (2^\wedge \llbracket W_1 \rrbracket)) (\rho_2 : \text{Square } (2^\wedge \llbracket W_2 \rrbracket)),$
 $\text{Typed_Box } f \rightarrow \text{Typed_Box } g \rightarrow$
 $\text{WF_Matrix } (2^\wedge \llbracket W_1 \rrbracket) (2^\wedge \llbracket W_1 \rrbracket) \rho_1 \rightarrow$
 $\text{WF_Matrix } (2^\wedge \llbracket W_2 \rrbracket) (2^\wedge \llbracket W_2 \rrbracket) \rho_2 \rightarrow$
 $\text{denote_box safe (inPar } f g) (\rho_1 \otimes \rho_2) =$
 $\text{denote_box safe } f \rho_1 \otimes \text{denote_box safe } g \rho_2.$

Lemma `HOAS_Equiv_inSeq` : $\forall w_1 w_2 w_3$
 $(c_1 c_1' : \text{Box } w_1 w_2) (c_2 c_2' : \text{Box } w_2 w_3),$
 $\text{Typed_Box } c_1 \rightarrow \text{Typed_Box } c_1' \rightarrow \text{Typed_Box } c_2 \rightarrow \text{Typed_Box } c_2' \rightarrow$
 $c_1 \equiv c_1' \rightarrow c_2 \equiv c_2' \rightarrow (c_2 \cdot c_1) \equiv (c_2' \cdot c_1').$

C.16 Ancilla.v

Fact valid_ancillae_box_equal : $\forall W_1 W_2 (c : \text{Box } W_1 W_2),$
valid_ancillae_box c \leftrightarrow valid_ancillae_box' c.

Fact valid_ancillae_unbox : $\forall W W' (c : \text{Pat } W \rightarrow \text{Circuit } W'),$
($\forall p, \text{valid_ancillae } (c \ p)$) \leftrightarrow valid_ancillae_box (box (fun p \Rightarrow c p)).

Lemma id_correct : $\forall W p, \text{valid_ancillae } (@\text{output } W \ p).$

Lemma update_merge : $\forall (\Gamma \Gamma' : \text{Ctx}) W W' v, \Gamma' == \text{singleton } v \ W \bullet \Gamma \rightarrow$
Valid (update_at $\Gamma' \ v \ (\text{Some } W')$) $==$ singleton v W' $\bullet \Gamma$.

Lemma change_type_singleton : $\forall v W W', \text{change_type } v \ W' \ (\text{singleton } v \ W) =$
singleton v W'.

Lemma ancilla_free_valid : $\forall W (c : \text{Circuit } W),$
ancilla_free c \rightarrow
valid_ancillae c.

Lemma ancilla_free_box_valid : $\forall W W' (c : \text{Box } W W'),$
ancilla_free_box c \rightarrow
valid_ancillae_box c.

Lemma valid_denote_true : $\forall W W' (c : \text{Box } W W')$
($\rho : \text{Square } (2^\wedge(\llbracket W \rrbracket))$) ($\rho' : \text{Square } (2^\wedge(\llbracket W \rrbracket))$) (safe : \mathbb{B}),
Typed_Box c \rightarrow
valid_ancillae_box c \rightarrow
denote_box true c $\rho = \rho' \rightarrow$
denote_box safe c $\rho = \rho'$.

Lemma valid_denote_false : $\forall W W' (c : \text{Box } W W')$
($\rho : \text{Square } (2^\wedge(\llbracket W \rrbracket))$) ($\rho' : \text{Square } (2^\wedge(\llbracket W \rrbracket))$) (safe : \mathbb{B}),
Typed_Box c \rightarrow
valid_ancillae_box c \rightarrow
denote_box false c $\rho = \rho' \rightarrow$
denote_box safe c $\rho = \rho'$.

C.17 Symmetric.v

Lemma unitary_at1_WT : $\forall n (U : \text{Unitary Qubit}) i (pf : i < n),$
Typed_Box (unitary_at1 n U i pf).

Lemma X_at_WT : $\forall n i pf, \text{Typed_Box } (X_at\ n\ i\ pf)$.

Lemma lt_leS_le : $\forall i j k,$
 $i < j \rightarrow j \leq S\ k \rightarrow i \leq k$.

Lemma strong_induction' :
 $\forall P : \mathbb{N} \rightarrow \text{Type},$
 $(\forall n : \mathbb{N}, (\forall k : \mathbb{N}, (k < n \rightarrow P\ k)) \rightarrow P\ n) \rightarrow$
 $\forall n i, i \leq n \rightarrow P\ i$.

Theorem strong_induction:
 $\forall P : \mathbb{N} \rightarrow \text{Type},$
 $(\forall n : \mathbb{N}, (\forall k : \mathbb{N}, (k < n \rightarrow P\ k)) \rightarrow P\ n) \rightarrow$
 $\forall n : \mathbb{N}, P\ n$.

Lemma le_hprop : $\forall (a\ b : \mathbb{N}) (pf1\ pf2 : a \leq b), pf1 = pf2$.

Lemma lt_hprop : $\forall (a\ b : \mathbb{N}) (pf1\ pf2 : a < b), pf1 = pf2$.

Lemma False_hprop : $\forall (pf1\ pf2 : \text{False}), pf1 = pf2$.

Lemma N_neq_hprop : $\forall (m\ n : \mathbb{N}) (pf1\ pf2 : m \neq n), pf1 = pf2$.

Lemma CNOT_at_i0_WT : $\forall (n\ j : \mathbb{N}) (pf_j : 0 < j) (pf_n : j < n),$
 $\text{Typed_Box } (\text{CNOT_at_i0}\ n\ j\ pf_j\ pf_n)$.

Lemma CNOT_at_i0_SS : $\forall n\ j$
 $(pfj : 0 < S\ (S\ j)) (pfj' : 0 < S\ j)$
 $(pfn : S\ (S\ j) < S\ (S\ n)) (pfn' : S\ j < S\ n),$
 $\text{CNOT_at_i0}\ (S\ (S\ n))\ (S\ (S\ j))\ pfj\ pfn =$
 $\text{box_q} \Rightarrow \text{let_ } (q0, (q1, qs)) \leftarrow q;$
 $\text{let_ } (q0, qs) \leftarrow \text{CNOT_at_i0}\ (S\ n)\ (S\ j)\ pfj'\ pfn'\ \$\ (q0, qs);$
 $(q0, (q1, qs))$.

Lemma CNOT_at_j0_WT : $\forall (n\ i : \mathbb{N}) (pf_i : 0 < i) (pf_n : i < n),$
 $\text{Typed_Box } (\text{CNOT_at_j0}\ n\ i\ pf_i\ pf_n)$.

Lemma CNOT_at_j0_SS : $\forall n\ i$
 $(pfi : 0 < S\ (S\ i)) (pfi' : 0 < S\ i)$
 $(pfn : S\ (S\ i) < S\ (S\ n)) (pfn' : S\ i < S\ n),$
 $\text{CNOT_at_j0}\ (S\ (S\ n))\ (S\ (S\ i))\ pfi\ pfn =$
 $\text{box_q} \Rightarrow \text{let_ } (q0, (q1, qs)) \leftarrow q;$
 $\text{let_ } (q0, qs) \leftarrow \text{CNOT_at_j0}\ (S\ n)\ (S\ i)\ pfi'\ pfn'\ \$\ (q0, qs);$
 $(q0, (q1, qs))$.

Lemma CNOT_at'_WT : $\forall (n \ i \ j : \mathbb{N})$
 $(pf_i : i < n) (pf_j : j < n) (pf_i_j : i \neq j),$
Typed_Box (CNOT_at' n i j pf_i pf_j pf_i_j).

Theorem CNOT_at_WT : $\forall n \ i \ j, \text{Typed_Box (CNOT_at } n \ i \ j).$

Lemma CNOT_at_0 : $\forall i \ j, \text{CNOT_at } 0 \ i \ j = \text{id_circ.}$

Lemma CNOT_at_1 : $\forall i \ j, \text{CNOT_at } 1 \ i \ j = \text{id_circ.}$

Lemma CNOT_at_n_0_SS : $\forall n' \ j',$
 $j' < n' \rightarrow$
CNOT_at (S (S n')) 0 (S (S j')) =
box_q \Rightarrow let_ (q0,(q1,qs)) \leftarrow q;
let_ (q0,qs) \leftarrow CNOT_at (S n') 0 (S j') \$ (q0,qs);
(q0,(q1,qs)).

Lemma CNOT_at_n_SS_0 : $\forall n' \ i',$
 $i' < n' \rightarrow$
CNOT_at (S (S n')) (S (S i')) 0 =
box_q \Rightarrow let_ (q0,(q1,qs)) \leftarrow q;
let_ (q0,qs) \leftarrow CNOT_at (S n') (S i') 0 \$(q0,qs);
(q0,(q1,qs)).

Lemma CNOT_at_at' : $\forall n \ i \ j (pfi : i < n) (pfj : j < n) (pfi_j : i \neq j),$
CNOT_at n i j = CNOT_at' n i j pfi pfj pfi_j.

Lemma CNOT_at_n_S_S : $\forall n' \ i' \ j',$
 $i' < n' \rightarrow j' < n' \rightarrow i' \neq j' \rightarrow$
CNOT_at (S n') (S i') (S j')
= box_q \Rightarrow let_ (q0,qs) \leftarrow q;
let_ qs \leftarrow CNOT_at n' i' j' \$ qs;
(q0,qs).

Lemma TOF_at_ij01_WT : $\forall n \ k \ pf_j \ pf_n,$
Typed_Box (TOF_at_ij01 n k pf_j pf_n).

Lemma TOF_at_ik01_WT : $\forall n \ j \ pf_j \ pf_n,$
Typed_Box (TOF_at_ik01 n j pf_j pf_n).

Lemma TOF_at_ki01_WT : $\forall n \ j \ pf_j \ pf_n,$
Typed_Box (TOF_at_ki01 n j pf_j pf_n).

Lemma TOF_at_i0_WT : $\forall n \ j \ k \ pf_ij \ pf_ik \ pf_jk \ pf_jn \ pf_kn,$
Typed_Box (TOF_at_i0 n j k pf_ij pf_ik pf_jk pf_jn pf_kn).

Lemma TOF_at_k0_WT : $\forall n i j \text{ pf_ij pf_ik pf_jk pf_in pf_jn}$,
Typed_Box (TOF_at_k0 n i j pf_ij pf_ik pf_jk pf_in pf_jn).

Lemma Toffoli_at'_WT : $\forall n (i j k : \text{Var}) (\text{pf_i} : i < n) (\text{pf_j} : j < n)$
 $(\text{pf_k} : k < n) (\text{pf_i_j} : i \neq j) (\text{pf_i_k} : i \neq k) (\text{pf_j_k} : j \neq k)$,
Typed_Box (Toffoli_at' n i j k pf_i pf_j pf_k pf_i_j pf_i_k pf_j_k).

Lemma Toffoli_at_WT : $\forall n (i j k : \text{Var})$, Typed_Box (Toffoli_at n i j k).

Lemma strip_one_l_in_WT : $\forall W W' (c : \text{Box } (\text{One} \otimes W) W')$,
Typed_Box c \rightarrow Typed_Box (strip_one_l_in c).

Lemma strip_one_l_in_eq : $\forall W W' (c : \text{Box } (\text{One} \otimes W) W')$
 $(\rho : \text{Matrix } (2^{\llbracket W \rrbracket}) (2^{\llbracket W' \rrbracket}))$,
denote_box true (strip_one_l_in c) $\rho =$ denote_box true c ρ .

Lemma strip_one_l_out_WT : $\forall W W' (c : \text{Box } W (\text{One} \otimes W'))$,
Typed_Box c \rightarrow Typed_Box (strip_one_l_out c).

Fact strip_one_l_out_eq : $\forall W W' (c : \text{Box } W (\text{One} \otimes W'))$
 $(\rho : \text{Matrix } (2^{\llbracket W \rrbracket}) (2^{\llbracket W' \rrbracket}))$,
denote_box true (strip_one_l_out c) $\rho =$ denote_box true c ρ .

Lemma strip_one_r_in_WT : $\forall W W' (c : \text{Box } (W \otimes \text{One}) W')$,
Typed_Box c \rightarrow Typed_Box (strip_one_r_in c).

Lemma strip_one_r_in_eq : $\forall W W' (c : \text{Box } (W \otimes \text{One}) W')$
 $(\rho : \text{Matrix } (2^{\llbracket W \rrbracket}) (2^{\llbracket W' \rrbracket}))$,
denote_box true (strip_one_r_in c) $\rho =$ denote_box true c ρ .

Lemma strip_one_r_out_WT : $\forall W W' (c : \text{Box } W (W' \otimes \text{One}))$,
Typed_Box c \rightarrow Typed_Box (strip_one_r_out c).

Fact strip_one_r_out_eq : $\forall W W' (c : \text{Box } W (W' \otimes \text{One}))$
 $(\rho : \text{Matrix } (2^{\llbracket W \rrbracket}) (2^{\llbracket W' \rrbracket}))$,
denote_box true (strip_one_r_out c) $\rho =$ denote_box true c ρ .

Lemma assert_at_WT : $\forall b n i$, Typed_Box (assert_at b n i).

Lemma init_at_WT : $\forall b n i$, Typed_Box (init_at b n i).

Lemma in_source_in_scope : $\forall n t i$, in_source n t i \rightarrow in_scope n t i.

Lemma `symmetric_reverse_symmetric` : $\forall n t c$
`(pf_sym : source_symmetric n t c),`
`source_symmetric n t (symmetric_reverse n t c pf_sym).`

Lemma `gate_acts_on_WT` : $\forall m (g : \text{Box } (m \otimes \text{Qubit}) (m \otimes \text{Qubit})) k,$
`gate_acts_on k g \rightarrow Typed_Box g.`

Lemma `source_symmetric_WT` : $\forall n t c, \text{source_symmetric } n t c \rightarrow \text{Typed_Box } c.$

Fact `gate_acts_on_noop_at` : $\forall m g k i,$
`@gate_acts_on (S m) k g \rightarrow`
`i \neq k \rightarrow i < S m \rightarrow`
`noop_on m i g.`

Lemma `fresh_state_ntensor` : $\forall n (\Gamma : \text{Ctx}),$
`add_fresh_state (n \otimes Qubit) Γ = Γ ++ List.repeat (Some Qubit) n.`

Fact `init_at_spec_strong` : $\forall b n i (\rho : \text{Square } (2^n)) (\text{safe} : \mathbb{B}),$
`i \leq n \rightarrow`
`denote_box safe (init_at b n i) ρ =`
`('I_ (2^i) \otimes bool_to_ket b \otimes 'I_ (2^(n-i))) \times ρ \times`
`('I_ (2^i) \otimes (bool_to_ket b)† \otimes 'I_ (2^(n-i))).`

Fact `assert_at_spec_safe` : $\forall b n i (\rho : \text{Square } (2^n)),$
`i \leq n \rightarrow`
`denote_box true (assert_at b n i) ρ =`
`('I_ (2^i) \otimes |0> \otimes 'I_ (2^(n-i))) \times ρ \times ('I_ (2^i) \otimes |0> \otimes 'I_ (2^(n-i)))`
`.+`
`('I_ (2^i) \otimes |1> \otimes 'I_ (2^(n-i))) \times ρ \times ('I_ (2^i) \otimes |1> \otimes 'I_ (2^(n-i))).`

Fact `assert_at_spec_unsafe` : $\forall b n i (\rho : \text{Square } (2^n)),$
`i \leq n \rightarrow`
`denote_box false (assert_at b n i) ρ =`
`('I_ (2^i) \otimes (bool_to_ket b)† \otimes 'I_ (2^(n-i))) \times ρ \times`
`('I_ (2^i) \otimes bool_to_ket b \otimes 'I_ (2^(n-i))).`

Lemma `assert_init_at_id` : $\forall b m i, i < S m \rightarrow$
`(assert_at b m i \cdot init_at b m i \equiv id_circ).`

Fact `init_assert_at_valid` : $\forall b m i W_1 (c : \text{Box } W_1 (S m \otimes \text{Qubit})),$
`i < S m \rightarrow`
`valid_ancillae_box' (assert_at b m i \cdot c) \rightarrow`
`init_at b m i \cdot assert_at b m i \cdot c \equiv c.`

Fact valid_ancillae_box'_equiv : $\forall W_1 W_2 (b1\ b2 : \text{Box } W_1\ W_2),$
 $b1 \equiv b2 \rightarrow \text{valid_ancillae_box}'\ b1 \leftrightarrow \text{valid_ancillae_box}'\ b2.$

Fact valid_inSeq : $\forall w1\ w2\ w3 (c1 : \text{Box } w1\ w2)\ (c2 : \text{Box } w2\ w3),$
Typed_Box c1 \rightarrow Typed_Box c2 \rightarrow
valid_ancillae_box' c1 \rightarrow valid_ancillae_box' c2 \rightarrow
valid_ancillae_box' (c2 · c1).

Lemma noop_source_inSeq : $\forall n\ t\ c1\ c2,$
Typed_Box c1 \rightarrow Typed_Box c2 \rightarrow
noop_source n t c1 \rightarrow
noop_source n t c2 \rightarrow
noop_source n t (c2 · c1).

Lemma denote_box_id_circ : $\forall b\ w\ \rho,$ WF_Matrix _ _ $\rho \rightarrow$
denote_box b (id_circ : Box w w) $\rho = \rho.$

Lemma valid_id_circ : $\forall w,$ valid_ancillae_box' (@id_circ w).

Fact symmetric_gate_noop_source : $\forall n\ t\ k\ g\ c,$
gate_acts_on k g \rightarrow
noop_source n t c \rightarrow
noop_source n t (g · c · g).

Fact init_at_noop : $\forall b\ m\ i\ j,$
valid_ancillae_box'
(assert_at b (S m) i · init_at b (S m) j · init_at b m i).

Fact symmetric_ancilla_noop_source : $\forall n\ t\ k\ c\ b,$
 $k < S\ n \rightarrow$
noop_source (S n) t c \rightarrow
noop_source n t (assert_at b (n+t) k · c · init_at b (n+t) k).

Lemma source_symmetric_noop : $\forall n\ t\ c,$
source_symmetric n t c \rightarrow noop_source n t c.

Fact ancilla_free_X_at : $\forall n\ k\ pf_k,$ ancilla_free_box (X_at n k pf_k).

Fact ancilla_free_CNOT_at : $\forall n\ a\ b,$ ancilla_free_box (CNOT_at n a b).

Fact ancilla_free_Toffoli_at : $\forall n\ a\ b\ c,$
ancilla_free_box (Toffoli_at n a b c).

Fact ancilla_free_seq : $\forall W\ W'\ W'' (c1 : \text{Box } W\ W')\ (c2 : \text{Box } W'\ W''),$
ancilla_free_box c1 \rightarrow ancilla_free_box c2 \rightarrow ancilla_free_box (c1 ;; c2).

Theorem `source_symmetric_valid` : $\forall (n\ t : \mathbb{N})$
 $(c : \text{Square_Box } ((n + t) \otimes \text{Qubit}))$,
`source_symmetric` $n\ t\ c \rightarrow$
`valid_ancillae_box` c .

Fact `gate_acts_on_reversible` : $\forall m\ g\ k$ (`@gate_acts_on` $m\ k\ g$),
 $g \cdot g \equiv \text{id_circ}$.

Fact `HOAS_Equiv_inSeq'` :
 $\forall (w1\ w2\ w3 : \text{WType}) (b1\ b1' : \text{Box } w1\ w2) (b2\ b2' : \text{Box } w2\ w3)$,
 $b1 \equiv b1' \rightarrow b2 \equiv b2' \rightarrow b1;; b2 \equiv b1';; b2'$.

Lemma `symmetric_reversible` : $\forall n\ t\ c$ (`pf_sym` : `source_symmetric` $n\ t\ c$),
`symmetric_reverse` $n\ t\ c\ \text{pf_sym} \cdot c \equiv \text{id_circ}$.

C.18 Oracles.v

Lemma `classical_merge_nil_l` : $\forall \Gamma$, $[\] \cup \Gamma = \Gamma$.

Lemma `classical_merge_nil_r` : $\forall \Gamma$, $\Gamma \cup [\] = \Gamma$.

Lemma `subset_classical_merge` : $\forall \Gamma\ \Gamma_1\ \Gamma_2$,
 $\Gamma_1 \cup \Gamma_2 \subset \Gamma \rightarrow (\Gamma_1 \subset \Gamma) * (\Gamma_2 \subset \Gamma)$.

Lemma `position_of_lt` : $\forall v\ \Gamma\ W$,
 $\text{nth } v\ \Gamma\ \text{None} = \text{Some } W \rightarrow (\text{position_of } v\ \Gamma < [[\Gamma]])$.

Lemma `singleton_nth_classical` : $\forall \Gamma\ v$,
 $\text{singleton } v\ \text{Qubit} \subset \Gamma \rightarrow \exists W, \text{nth } v\ \Gamma\ \text{None} = \text{Some } W$.

Lemma `get_wire_WT` : $\forall \Gamma\ m\ n\ \text{default}$ (`p` : `Pat` $(m \otimes \text{Qubit})$),
 $(n < m) \rightarrow$
 $\Gamma \vdash p : \text{Pat} \rightarrow$
 $\{\Gamma_1 : \text{OCtx} \ \& \ \{\Gamma_2 : \text{OCtx} \ \& \ \Gamma == \Gamma_1 \bullet \Gamma_2 \ \&$
 $\Gamma_1 \vdash \text{get_wire } n\ p\ \text{default} : \text{Pat}\}\}$.

Lemma `share_to_WT` : $\forall n\ k$, `Typed_Box` $(\text{share_to } n\ k)$.

Lemma `share_to_WT'` : $\forall n\ k$, `Typed_Box` $(\text{share_to}'\ n\ k)$.

Lemma `size_repeat_ctx` : $\forall n\ W$, `size_ctx` $(\text{repeat } (\text{Some } W)\ n) = n$.

Lemma `ctx_dom_repeat` : $\forall n, \text{ctx_dom } (\text{repeat } (\text{Some Qubit}) n) = \text{seq } 0 n.$

Lemma `maps_to_repeat` : $\forall v n W, v < n \rightarrow$
`maps_to` v `(repeat (Some W) n)` = `Some v`.

Lemma `subst_pat_σ_n`: $\forall W W' n (p : \text{Pat } W), (\text{pat_max } p < n) \rightarrow$
`subst_pat` `(repeat (Some W') n)` $p = p.$

Lemma `pat_max_fresh` : $\forall m n,$
`(pat_max (add_fresh_pat (n \otimes Qubit) (repeat (Some Qubit) m)) < S (m+n)).`

Lemma `singleton_repeat` : $\forall n W,$
`singleton` $n W = \text{repeat } \text{None } n ++ \text{repeat } (\text{Some } W) 1.$

Lemma `ctx_dom_none_repeat` : $\forall m n,$
`ctx_dom` `(repeat None m ++ repeat (Some Qubit) n)` = `seq m n.`

Lemma `size_repeat_none` : $\forall (n : \mathbb{N}), \text{size_ctx } (\text{repeat } \text{None } n) = 0.$

Lemma `types_pat_fresh_ntensor` : $\forall (\Gamma : \text{Ctx}) (n : \mathbb{N}), n \neq 0 \rightarrow$
`Valid` `(repeat None (length Γ) ++ repeat (Some Qubit) n)` \vdash
`add_fresh_pat (n \otimes Qubit) Γ :Pat.`

Lemma `qubit_at_reflect` : $\forall v \Gamma,$
`qubit_at` $v \Gamma = \text{true} \leftrightarrow \text{nth } v \Gamma \text{ None} = \text{Some Qubit}.$

Lemma `ntensor_fold` : $\forall n W, W \otimes (n \otimes W) = (S n \otimes W).$

Lemma `compile_WT` : $\forall (b : \text{bexp}) (\Gamma : \text{Ctx}), \text{Typed_Box } (\text{compile } b \Gamma).$

Lemma `ctx_to_mat_list_length` : $\forall \Gamma f, \text{length } (\text{ctx_to_mat_list } \Gamma f) = \llbracket \Gamma \rrbracket.$

Lemma `WF_ctx_to_matrix` : $\forall \Gamma f,$
`WF_Matrix` $(2^{\llbracket \Gamma \rrbracket}) (2^{\llbracket \Gamma \rrbracket}) (\text{ctx_to_matrix } \Gamma f).$

Lemma `WF_ctx_to_mat_list` : $\forall \Gamma f,$
`WF_Matrix` $(2^{\llbracket \Gamma \rrbracket}) (2^{\llbracket \Gamma \rrbracket}) (\text{big_kron } (\text{ctx_to_mat_list } \Gamma f)).$

Lemma `pure_bool_to_matrix` : $\forall b, \text{Pure_State } (\text{bool_to_matrix } b).$

Lemma `pure_big_kron` : $\forall (n : \mathbb{N}) (l : \text{list } (\text{Square } n)) (A : \text{Square } n),$
 $(\forall i : \mathbb{N}, \text{Pure_State } (\text{nth } i l A)) \rightarrow \text{Pure_State } (\otimes l).$

Lemma `mixed_big_kron` : $\forall (n : \mathbb{N}) (l : \text{list } (\text{Square } n)) (A : \text{Square } n),$
 $(\forall i : \mathbb{N}, \text{Mixed_State } (\text{nth } i l A)) \rightarrow \text{Mixed_State } (\otimes l).$

Lemma big_kron_append : $\forall m n (l1 l2 : \text{list } (\text{Matrix } m n))$
 $(A B : \text{Matrix } m n),$
 $(\forall j, \text{WF_Matrix } m n (\text{nth } j l1 A)) \rightarrow$
 $(\forall j, \text{WF_Matrix } m n (\text{nth } j l2 B)) \rightarrow$
 $\otimes (l1 ++ l2) = (\otimes l1) \otimes (\otimes l2).$

Lemma pure_ctx_to_matrix : $\forall \Gamma f, \text{Pure_State } (\text{ctx_to_matrix } \Gamma f).$

Lemma is_valid_singleton_merge : $\forall W (\Gamma : \text{Ctx}) n,$
 $(\text{length } \Gamma \leq n) \rightarrow \text{is_valid } (\Gamma \uplus \text{singleton } n W).$

Lemma size_ctx_app : $\forall (\Gamma_1 \Gamma_2 : \text{Ctx}),$
 $\text{size_ctx } (\Gamma_1 ++ \Gamma_2) = (\text{size_ctx } \Gamma_1 + \text{size_ctx } \Gamma_2).$

Lemma singleton_length : $\forall n W, \text{length } (\text{singleton } n W) = (n + 1).$

Lemma ctx_lookup_∃ : $\forall v \Gamma f, \text{get_context } (\text{b_var } v) \subset \Gamma \rightarrow$
 $\text{ctx_to_mat_list } \Gamma f \text{ !! position_of } v \Gamma = \text{Some } (\text{bool_to_matrix } (f v)).$

Fact CNOT_at_spec : $\forall (b1 b2 : \mathbb{B}) (n x y : \mathbb{N}) (li : \text{list } (\text{Matrix } 2 2)),$
 $x < n \rightarrow y < n \rightarrow x \neq y \rightarrow$
 $\text{nth_error } li x = \text{Some } (\text{bool_to_matrix } b1) \rightarrow$
 $\text{nth_error } li y = \text{Some } (\text{bool_to_matrix } b2) \rightarrow$
 $\llbracket \text{CNOT_at } n x y \rrbracket (\otimes li) = \otimes (\text{update_at } li y (\text{bool_to_matrix } (b1 \oplus b2))).$

Fact Toffoli_at_spec : $\forall (b1 b2 b3 : \mathbb{B}) (n x y z : \mathbb{N})$
 $(li : \text{list } (\text{Matrix } 2 2)),$
 $x < n \rightarrow y < n \rightarrow z < n \rightarrow x \neq y \rightarrow x \neq z \rightarrow y \neq z \rightarrow$
 $\text{nth_error } li x = \text{Some } (\text{bool_to_matrix } b1) \rightarrow$
 $\text{nth_error } li y = \text{Some } (\text{bool_to_matrix } b2) \rightarrow$
 $\text{nth_error } li z = \text{Some } (\text{bool_to_matrix } b3) \rightarrow$
 $\llbracket \text{Toffoli_at } n x y z \rrbracket (\otimes li) =$
 $\otimes (\text{update_at } li z (\text{bool_to_matrix } ((b1 \&\& b2) \oplus b3))).$

Lemma init_at_spec : $\forall (b : \mathbb{B}) (n i : \mathbb{N}) (l1 l2 : \text{list } (\text{Square } 2))$
 $(A B : \text{Square } 2), \text{length } l1 = i \rightarrow \text{length } l2 = n - i \rightarrow$
 $(\forall j, \text{Mixed_State } (\text{nth } j l1 A)) \rightarrow (\forall j, \text{Mixed_State } (\text{nth } j l2 B)) \rightarrow$
 $i < S n \rightarrow$
 $\llbracket \text{init_at } b n i \rrbracket (\otimes (l1 ++ l2)) = \otimes (l1 ++ [\text{bool_to_matrix } b] ++ l2).$

Theorem compile_correct : $\forall (b : \text{bexp}) (\Gamma : \text{Ctx}) (f : \text{Var} \rightarrow \mathbb{B}) (t : \mathbb{B}),$
 $\text{get_context } b \subset \Gamma \rightarrow$
 $\llbracket \text{compile } b \Gamma \rrbracket ((\text{bool_to_matrix } t) \otimes (\text{ctx_to_matrix } \Gamma f)) =$
 $\text{bool_to_matrix } (t \oplus b \mid f) \otimes \text{ctx_to_matrix } \Gamma f.$

Bibliography

- Scott Aaronson. *Quantum computing since Democritus*. Cambridge University Press, 2013.
- Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols. In *Logic in computer science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 415–425. IEEE, 2004.
- Dorit Aharonov. A simple proof that toffoli and hadamard are quantum universal. *arXiv preprint quant-ph/0301040*, 2003.
- Dorit Aharonov and Michael Ben-Or. Fault-tolerant quantum computation with constant error. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 176–188. ACM, 1997.
- Dorit Aharonov, Andris Ambainis, Julia Kempe, and Umesh Vazirani. Quantum walks on graphs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 50–59. ACM, 2001.
- Tameem Albash and Daniel A Lidar. Adiabatic quantum computation. *Reviews of Modern Physics*, 90(1):015002, 2018.
- David Z Albert. On quantum-mechanical automata. *Physics Letters A*, 98(5-6): 249–252, 1983.
- Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 249–258. IEEE, 2005.
- Thorsten Altenkirch and Alexander S Green. The quantum IO monad. *Semantic Techniques in Quantum Computation*, pages 173–205, 2010.
- Thorsten Altenkirch, Jonathan Grattage, Juliana K. Vizzotto, and Amr Sabry. An algebra of pure quantum programming. *Electronic Notes in Theoretical Computer Science*, 170(Complete):23–47, 2007. doi:10.1016/j.entcs.2006.12.010.

- Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.
- 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/>.
- Andrew Appel. *Software Foundations Volume 3: Verified Functional Algorithms*. Electronic textbook, 2018. Version 1.4. <https://softwarefoundations.cis.upenn.edu/current/vfa-current>.
- Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.
- Miriam Backens. The zx-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, 2014.
- Costin Badescu and Prakash Panangaden. Quantum alternation: Prospects and problems. In *Proceedings of the 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015.*, pages 33–42, 2015. doi:10.4204/EPTCS.195.3.
- Steven Balensiefer, Lucas Kregor-Stickles, and Mark Oskin. An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 186–196. IEEE Computer Society, 2005.
- Alexandru Baltag and Sonja Smets. LQP: the dynamic logic of quantum information. *Mathematical structures in computer science*, 16(03):491–525, 2006.
- Alexandru Baltag and Sonja Smets. Quantum logic as a dynamic logic. *Synthese*, 179(2):285–306, 2011.
- Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology-CRYPTO 2011*, pages 71–90. Springer, 2011.
- Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *Mathematics of Program Construction*, pages 1–6. Springer, 2012.
- Stephane Beauregard. Circuit for shor’s algorithm using $2n+3$ qubits. *arXiv preprint quant-ph/0205095*, 2002.

- David Beckman, Amalavoyal N Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Physical Review A*, 54(2):1034, 1996.
- Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of statistical physics*, 22(5):563–591, 1980.
- Charles H Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.
- P.N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 1995. doi:10.1007/BFb0022251.
- Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.
- Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. Toward an architecture for quantum programming. *The European Physical Journal D*, 25(2):181–200, 2003.
- Stephen Blaha. Quantum computers and quantum computer languages: quantum assembly language and quantum c language. *arXiv preprint quant-ph/0201082*, 2002.
- Alex Bocharov, Martin Roetteler, and Krysta M Svore. Efficient synthesis of probabilistic quantum circuits with fallback. *Physical Review A*, 91(5):052317, 2015a.
- Alex Bocharov, Martin Roetteler, and Krysta M Svore. Efficient synthesis of universal repeat-until-success quantum circuits. *Physical review letters*, 114(8):080502, 2015b.
- Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–83. Open Publishing Association, 2015. doi:10.4204/EPTCS.195.6.
- Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot. <http://coquelicot.saclay.inria.fr/>, 2015.
- Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda—a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

- Olivier Brunet and Philippe Jorrand. Dynamic quantum logic for quantum programs. *International Journal of Quantum Information*, 2(01):45–54, 2004.
- Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg, and Vincent Siles. CoqEAL - the coq effective algebra library. <https://github.com/CoqEAL/CoqEAL>, 2016.
- Rohit Chadha, Paulo Mateus, and Amílcar Sernadas. Reasoning about imperative quantum programs. *Electronic Notes in Theoretical Computer Science*, 158:19–39, 2006.
- Andrew M Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 59–68. ACM, 2003.
- Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.
- Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936a.
- Alonzo Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(01):40–41, 1936b.
- Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 339–354. The Royal Society, 1998.
- Bob Coecke and Ross Duncan. Interacting quantum observables. In *International Colloquium on Automata, Languages, and Programming*, pages 298–310. Springer, 2008.
- Bob Coecke and Aleks Kissinger. *Picturing quantum processes*. Cambridge University Press, 2017.
- Coq Development Team. The Coq proof assistant reference manual, version 8.8, 2018. Electronic resource, available from <https://coq.inria.fr/refman/>.
- Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.
- Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

- D-Wave Systems, Inc. Programming with D-Wave: Map coloring problem, 2013.
- Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *Journal of the ACM (JACM)*, 54(2):8, 2007.
- Vincent Danos, Elham Kashefi, Prakash Panangaden, and Simon Perdrix. Extended measurement calculus. *Semantic techniques in quantum computation*, pages 235–310, 2009.
- Christopher M Dawson and Michael A Nielsen. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.
- Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In *International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer, 1995.
- David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.
- David Deutsch. Quantum computational networks. *Proc. R. Soc. Lond. A*, 425(1868):73–90, 1989.
- David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 439, pages 553–558. The Royal Society, 1992.
- Ellie D’Hondt and Prakash Panangaden. Quantum weakest preconditions. *Mathematical Structures in Computer Science*, 16(03):429–451, 2006.
- Dennis Dieks. Communication by epr devices. *Physics Letters A*, 92(6):271–272, 1982.
- 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.
- Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, Jan 2012. ISSN 1573-0670. doi:10.1007/s10817-010-9194-x. URL <https://doi.org/10.1007/s10817-010-9194-x>.
- Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488, 1982.

- Richard P Feynman and Albert R Hibbs. *Quantum mechanics and path integrals*. International series in pure and applied physics. McGraw Hill, 1965.
- Jean Baptiste Joseph Fourier. Solution d’une question particuliere du calcul des inégalités. *Nouveau Bulletin des Sciences par la Société philomatique de Paris*, 99: 100, 1826.
- Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012.
- Simon J. Gay. Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581–600, 2006.
- Iulia M. Georgescu, Sahel Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153, 2014.
- Craig Gidney. Factoring with $n+2$ clean qubits and $n-1$ dirty qubits. *arXiv preprint arXiv:1706.07884*, 2017.
- Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11): 1382–1393, 2008.
- Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.
- Daniel Gottesman. Stabilizer codes and quantum error correction. *arXiv preprint quant-ph/9705052*, 1997.
- Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 333–342, 2013a.
- Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in Quipper. In *Proceedings of the 5th International Conference on Reversible Computation*, volume 7948 of *Lecture Notes in Computer Science*, pages 110–124, 2013b. ISBN 978-3-642-38985-6.
- Alexander S Green. *Towards a formally verified functional quantum programming language*. PhD thesis, University of Nottingham, 2010.

- 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*, pages 212–219, New York, NY, USA, 1996. ACM. ISBN 0-89791-785-5. doi:10.1145/237814.237866. URL <http://doi.acm.org/10.1145/237814.237866>.
- Amar Hadzihasanovic. A diagrammatic axiomatisation for qubit entanglement. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 573–584. IEEE Computer Society, 2015.
- Amar Hadzihasanovic. *The algebra of entanglement and the geometry of composition*. PhD thesis, University of Oxford, 2017.
- Brian C Hall. *Quantum theory for mathematicians*, volume 267. Springer, 2013.
- Thomas Häner, Martin Roetteler, and Krysta M Svore. Factoring using $2n+2$ qubits with toffoli based modular multiplication. *arXiv preprint arXiv:1611.07995*, 2016.
- Jeff Heckey, Shruti Patil, Ali Javadi-Abhari, Adam Holmes, Daniel Kudrow, Kenneth R Brown, Diana Franklin, Frederic T Chong, and Margaret Martonosi. Compiler management of communication and parallelism for quantum computation. *ACM SIGARCH Computer Architecture News*, 43(1):445–456, 2015.
- IARPA. IARPA quantum computer science program, April 2010. URL <https://www.fbo.gov/notices/637e87ac1274d030ce2ab69339ccf93c>. Broad Agency Announcement IARPA-BAA-10-02.
- IBM. IBM quantum experience, 2017. URL <http://research.ibm.com/ibm-q/qx/>.
- Ali Javadi-Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amilan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE, 2012.
- Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, 2015.
- Mark W Johnson, Mohammad HS Amin, Suzanne Gildert, Trevor Lanting, Firas Hamze, Neil Dickson, R Harris, Andrew J Berkley, Jan Johansson, Paul Bunyk, et al. Quantum annealing with manufactured spins. *Nature*, 473(7346):194, 2011.
- Yoshihiko Kakutani. A logic for formal verification of quantum programs. In *Advances in Computer Science-ASIAN 2009. Information Security and Privacy*, pages 79–93. Springer, 2009.

- Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry fields: A software platform for photonic quantum computing. *arXiv preprint arXiv:1804.03159*, 2018.
- Aleks Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing*. PhD thesis, University of Oxford, 2011.
- A Yu Kitaev. Quantum error correction with imperfect gates. In *Quantum Communication, Computing, and Measurement*, pages 181–188. Springer, 1997.
- Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.
- Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Practical approximation of single-qubit unitaries by single-qubit quantum clifford and t circuits. *IEEE Transactions on Computers*, 65(1):161–172, 2016.
- Emmanuel H. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.
- Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- Dexter Kozen. A probabilistic pdl. *Journal of Computer and System Sciences*, 30(2):162–178, 1985.
- Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. *SIGPLAN Notices*, 50(1):17–30, January 2015. doi:10.1145/2775051.2676969.
- Samuel A Kutin. Shor’s algorithm on a nearest-neighbor machine. *arXiv preprint quant-ph/0609001*, 2006.
- Leonidas Lampropoulos. *Random Testing for Language Design*. PhD thesis, University of Pennsylvania, 2018.
- Leonidas Lampropoulos and Benjamin C. Pierce. *QuickCHick: Property-Based Testing In Coq*. Software Foundations series, volume 4. Electronic textbook, August 2018. Version 1.0. <https://softwarefoundations.cis.upenn.edu/qc-current/index.html>.
- Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, December 1999. doi:10.1145/331963.331977.
- Yangjia Li and Mingsheng Ying. Algorithmic analysis of termination problems for quantum programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):35, 2018.

- Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. Enriching a linear/non-linear lambda calculus: A programming language for string diagrams. In *Proceedings of the 33rd Annual IEEE Symposium on Logic in Computer Science, LICS 2018*, 2018.
- Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. *arXiv preprint arXiv:1601.03835*, 2016.
- Assia Mahboubi, Enrico Tassi, Yves Bertot, and Georges Gonthier. *Mathematical Components*. 2016. Electronic resource, available from <https://math-comp.github.io/mcb/book.pdf>.
- Mohamed Yousri Mahmoud and Amy P Felty. Formal meta-level analysis framework for quantum programming languages. *Electronic Notes in Theoretical Computer Science*, 338:185–201, 2018.
- Wolfgang Maurerer. Semantics and simulation of communication in quantum programming. Master’s thesis, University Erlangen-Nuremberg, 2005.
- Conor McBride. *I Got Plenty o’ Nuttin’*, pages 207–233. Springer International Publishing, 2016. doi:10.1007/978-3-319-30936-1_12.
- J Miszczak. Models of quantum computation and quantum programming languages. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, 59(3):305–324, 2011.
- Theodore Samuel Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. PhD thesis, University of Basel, 1936.
- Rajagopal Nagarajan, Nikolaos Papanikolaou, and David Williams. Simulating and compiling code for the sequential quantum random access machine. *Electronic Notes in Theoretical Computer Science*, 170:101–124, 2007.
- Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018.
- Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.
- T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science **2283**. Springer-Verlag, 2002.
- Bernhard Ömer. A procedural formalism for quantum computing. Master’s thesis, Department of Theoretical Physics, Technical University of Vienna, 1998.

- Bernhard Ömer. Quantum programming in QCL. Master’s thesis, Institute of Information Systems, Technical University of Vienna, 2000.
- Bernhard Ömer. *Structured quantum programming*. PhD thesis, Technical University of Vienna, 2003.
- Bernhard Ömer. Classical concepts in quantum programming. *International Journal of Theoretical Physics*, 44(7):943–955, 2005.
- Luca Paolini and Margherita Zorzi. qPCF: A language for quantum circuit computations. In *International Conference on Theory and Applications of Models of Computation*, pages 455–469. Springer, 2017.
- Luca Paolini, Luca Roversi, and Margherita Zorzi. Quantum programming made easy. *arXiv preprint arXiv:1711.00774*, 2017.
- Alex Parent, Martin Roetteler, and Krysta M. Svore. Revs: A tool for space-optimized reversible synthesis. In *Proceedings of the 9th International Conference on Reversible Computation (RC 2017)*. Springer, July 2017. URL <https://www.microsoft.com/en-us/research/publication/reversible-circuit-compilation-with-space-constraints-2/>.
- Jennifer Paykin. *Linear/non-linear types for embedded domain-specific languages*. PhD thesis, University of Pennsylvania, 2018.
- Jennifer Paykin and Steve Zdancewic. A linear/producer/consumer model of classical linear logic. *Mathematical Structures in Computer Science*, pages 1–26, 2016.
- Jennifer Paykin and Steve Zdancewic. The linearity monad. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 117–132. ACM, 2017.
- Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009894.
- 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, pages 199–208, New York, NY, USA, 1988. ACM. doi:10.1145/53990.54010.
- Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2018. Version 5.6. <https://softwarefoundations.cis.upenn.edu/>.

- John Preskill. Fault-tolerant quantum computation. In *Introduction to quantum computation and information*, pages 213–269. World Scientific, 1998.
- John Preskill. Quantum computing in the nisq era and beyond. *arXiv preprint arXiv:1801.00862*, 2018.
- Norman Ramsey. Literate programming simplified. *IEEE software*, 11(5):97–105, 1994.
- Robert Rand and Arthur Azevedo de Amorim. Programs and proofs in the coq proof assistant. Tutorial at Principles of Programming Languages, 2016. URL http://www.cis.upenn.edu/~rrand/popl_2016/.
- Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. doi:10.4204/EPTCS.266.8. URL <https://doi.org/10.4204/EPTCS.266.8>.
- Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning about reversible quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018a.
- Robert Rand, Jennifer Paykin, and Steve Zdancewic. Phantom types for quantum programs. The Fourth International Workshop on Coq for Programming Languages, January 2018b.
- Robert Raussendorf and Hans J Briegel. A one-way quantum computer. *Physical Review Letters*, 86(22):5188, 2001.
- Robert Raussendorf and Hans J Briegel. Computational model for the one-way quantum computer: Concepts and summary. *arXiv preprint quant-ph/0207183*, 2002.
- Ran Raz and Avishay Tal. Oracle separation of BQP and PH. *Electronic Colloquium on Computational Complexity (ECCC)*, 25:107, 2018. URL <https://eccc.weizmann.ac.il/report/2018/107>.
- Mathys Rennela. Towards a quantum domain theory: Order-enrichment and fixpoints in w^* -algebras. In *Mathematical Foundations of Programming Semantics Thirtieth Annual Conference*, page 289, 2014.
- Mathys Rennela and Sam Staton. Classical control and quantum circuits in enriched category theory. In *Proceedings of the 33rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2017*, 2017.

- Rigetti Computing. Pyquil documentation. URL <http://pyquil.readthedocs.io/en/latest/>.
- Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. In *Proceedings of the 14th International Conference on Quantum Physics and Logic, QPL 2017*, 2017.
- Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.
- Neil J Ross and Peter Selinger. Optimal ancilla-free clifford+ t approximation of z-rotations. *arXiv preprint arXiv:1403.2975*, 2014.
- Roland Rüdiger. Quantum programming languages: An introductory overview. *The Computer Journal*, 50(2):134–150, 2006.
- Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In *International Conference on Foundations of Software Science and Computation Structures*, pages 348–364. Springer, 2018.
- Mehdi Saeedi and Igor L Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys (CSUR)*, 45(2):21, 2013.
- Jeff W. Sanders and Paolo Zuliani. Quantum programming. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 80–99, 2000.
- Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, August 2004a.
- Peter Selinger. A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming*, pages 1–6. Springer, 2004b.
- Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(03):527–552, 2006.
- Peter Selinger and Benoît Valiron. A linear-non-linear model for a computational call-by-value lambda calculus. In *Proceedings of the Eleventh International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2008*, Springer Lecture Notes in Computer Science 4962, pages 81–96, Budapest, Hungary, April 2008.
- Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.

- Yaoyun Shi. Both toffoli and controlled-not need little help to do universal quantum computing. *Quantum Information & Computation*, 3(1):84–92, 2003.
- Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.
- Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41:303–332, January 1999. doi:10.1137/S0036144598347011.
- Safat Siddiqui, Mohammed Jahirul Islam, and Omar Shehab. Five quantum algorithms using Quipper. *arXiv preprint arXiv:1406.4481*, 2014.
- Alexander Singh, Konstantinos Giannakis, and Theodore Andronikos. Qumin, a minimalist quantum programming language. *arXiv preprint arXiv:1704.04460*, 2017.
- Jonathan M. Smith, Julien Neil Ross, Peter Selinger, and Benoît Valiron. Quipper: Concrete resource estimation in quantum algorithms. In *Proceedings of the Twelfth Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, April 2014. URL <http://arxiv.org/abs/1412.0625>.
- Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.
- Sam Staton. Algebraic effects, linearity, and quantum programming languages. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15*, pages 395–406, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676999.
- Damian S. Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.
- Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.
- Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.
- Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ACM SIGPLAN Notices*, volume 46, pages 266–278. ACM, 2011.

- Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for shor’s factoring algorithm using $2n+ 2$ qubits. *Quantum Information & Computation*, 6(2):184–192, 2006.
- Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages, and Programming*, pages 632–644. Springer, 1980.
- Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.
- Dominique Unruh. Quantum relational hoare logic. *arXiv preprint arXiv:1802.03188*, 2018.
- André van Tonder. A lambda calculus for quantum computation. *SIAM Journal of Computation*, 33(5):1109–1135, 2004.
- André van Tonder and Miquel Dorca. Quantum computation, categorical semantics and linear logic. *arXiv preprint quant-ph/0312174*, 2003.
- Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147, 1996.
- Juliana Kaizer Vizzotto, André Rauber Du Bois, and Amr Sabry. The arrow calculus as a quantum programming language. In Hiroakira Ono, Makoto Kanazawa, and Ruy de Queiroz, editors, *Proceedings of Logic, Language, Information and Computation: 16th International Workshop, WoLLIC 2009*, pages 379–393. Springer Berlin Heidelberg, June 2009a. doi:10.1007/978-3-642-02261-6_30.
- Juliana Kaizer Vizzotto, Giovanni Rubert Librelotto, and Amr Sabry. Reasoning about general quantum programs over mixed states. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, pages 321–335, Berlin, Heidelberg, 2009b. Springer Berlin Heidelberg. doi:10.1007/978-3-642-10452-7_22.
- Juliana Kaizer Vizzotto, Bruno Crestani Calegari, and Eduardo Kessler Piveta. A double effect λ -calculus for quantum computation. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 61–74, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40922-6_5.
- John Watrous. Succinct quantum proofs for properties of finite groups. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 537–546. IEEE, 2000.
- John Watrous. *Theory of Quantum Information*. Cambridge university press, 2018.

- Dave Wecker and Krysta M Svore. LIQUiD: A software design architecture and domain-specific language for quantum computing. *arXiv:1402.4467 [quant-ph]*, 2014.
- WK Wothers and WK Zurek. Quantum no-cloning theorem. *Nature*, 299:802, 1982.
- Andrew Chi-Chih Yao. Quantum circuit complexity. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 352–361. IEEE, 1993.
- Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.
- Mingsheng Ying. Quantum recursion and second quantisation. May 2014. *arXiv:1405.4443 [quant-ph]*.
- Mingsheng Ying, Nengkun Yu, and Yuan Feng. Defining Quantum Control Flow. *ArXiv e-prints*, September 2012.
- Mingsheng Ying, Nengkun Yu, and Yuan Feng. Alternation in quantum programming: From superposition of data to superposition of programs. *CoRR*, abs/1402.5172, 2014. URL <http://arxiv.org/abs/1402.5172>.
- Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. Invariants of quantum programs: characterisations and generation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 818–832. ACM, 2017.
- Christof Zalka. Shor’s algorithm with fewer (pure) qubits. *arXiv preprint quant-ph/0601097*, 2006.