Formal Methods in Quantum Circuit Design

by

Matthew Amy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Simon Perdrix Chargé de recherche, CNRS
Supervisor:	Michele Mosca Professor, Dept. of Combinatorics & Optimization
Internal Member:	Prabhakar Ragde Professor, David R. Cheriton School of Computer Science
Internal Member:	Richard Cleve Professor, David R. Cheriton School of Computer Science
Internal-External Member:	Jon Yard Associate Professor, Dept. of Combinatorics & Optimization

Author's declaration

This thesis consists of material all of which I authored or co-authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The design and compilation of correct, efficient quantum circuits is integral to the future operation of quantum computers. This thesis makes contributions to the problems of optimizing and verifying quantum circuits, with an emphasis on the development of formal models for such purposes. We also present software implementations of these methods, which together form a full stack of tools for the design of optimized, formally verified quantum oracles.

On the optimization side, we study methods for the optimization of R_Z and CNOT gates in Clifford+R_Z circuits. We develop a general, efficient optimization algorithm called *phase folding*, which reduces the number of R_Z gates without increasing any metrics by computing its *phase polynomial*. This algorithm can further be combined with synthesis techniques for CNOT-dihedral operators to optimize circuits with respect to particular costs. We then study the optimal synthesis problem for CNOT-dihedral operators from the perspectives of R_Z and CNOT gate optimization. In the case of R_Z gate optimization, we show that the optimal synthesis problem is polynomial-time equivalent to minimum-distance decoding in certain Reed-Muller codes. For the CNOT optimization problem, we show that the optimal synthesis problem is at least as hard as a combinatorial problem related to Gray codes. In both cases, we develop heuristics for the optimal synthesis problem, which together with phase folding reduces T counts by 42% and CNOT counts by 22% across a suite of real-world benchmarks.

From the perspective of formal verification, we make two contributions. The first is the development of a formal model of quantum circuits with ancillary bits based on the Feynman path integral, along with a concrete verification algorithm. The path integral model, with some syntactic sugar, further doubles as a natural specification language for quantum computations. Our experiments show some practical circuits with up to hundreds of qubits can be efficiently verified. Our second contribution is a formally verified, optimizing compiler for reversible circuits. The compiler compiles a classical, irreversible language to reversible circuits, with a formal, machine-checked proof of correctness written in the proof assistant F*. The compiler is structured as a partial evaluator, allowing verification to be carried out significantly faster than previous results.

Acknowledgements

Having spent a great deal of my adult life in graduate studies, I'm very grateful for the friends, researchers and support staff who have made the process a pleasure.

I first wish to thank my advisor, Michele Mosca, for providing me with guidance, support and useful ideas throughout my time at Waterloo. I also wish to thank my committee members Jon Yard, Prabhakar Ragde, Richard Cleve and Simon Perdrix for reading a thesis that turned out significantly longer than intended, as well as for their helpful comments and conversations over the years. I especially wish to thank Simon for travelling to Waterloo in the worst week of Winter to attend my defence.

Over the many years I spent as a graduate student, I had the privilege of working with talented researchers and fellow students whom have impacted me greatly as a researcher. For that I wish to thank Julien Ross, Vlad Gheorghiu, Martin Roetteler, Vadym Kliuchnikov, Olivia Di Matteo, Mathias Soeken, Alex Parent, John Schanck, Parsiad Azimzadeh, Nathan Killoran and Dmitri Maslov.

The many years I spent at Waterloo would have been far less bearable and far more damaging to my mental stability were it not for the friends I made there. I wish to thank in particular Parsiad Azimzadeh, Alexandre Laplante, Vincent Launchbury, Kyle Robinson, Vlad Gheorghiu, Olivia Di Matteo, Alex Parent, Vincent Russo, Sebastian Verschoor, Mária Kieferová and the memory of Alex S. Forskan for keeping me sane.

Finally, I wish to thank my parents for all of their immeasurable support and for always letting me stay inside to play with computers, and Rebecca for her patience, love, and for agreeing to marry a graduate student.

For my father

Table of Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

Since the conceptualization of a *quantum mechanical computer* by Yuri Manin and Richard Feynman in the early 1980's, researchers have strived to build such machines. Motivated by the promise of exponentially faster solutions to important problems such as simulation of quantum systems [Fey82, Llo96] and later integer factorization [Sho94], researchers in the 80's and 90's laid the groundwork for physical implementations of quantum computers via technologies including NMR [CFH97, GC97], trapped ions [CZ95], quantum dots [LD98] and quantum optics [KLM01], as well as theoretical results on universal primitives [Tof80, Deu85] and error correction [Sho95]. Despite such advancements, the lack of scalable, reliable quantum computers throughout the 2000's led to some pessimism towards quantum computation outside of the physics community, and as a result few scalable tools for the design and manipulation of quantum computations were developed.

We are now in an era where large-scale quantum computers appear not only feasible but a relative, if still distant, certainty. As a result, the development of software design tools for quantum computers has rapidly accelerated. *Quantum programming languages* and the associated *compilers* in particular have seen extensive work, with a large array of programming environments having been developed – for instance Quipper [GLR+13], ScaffCC [JPK+15], Quil/PyQuil [SCZ16], ProjectQ [SHT18], $Q|SI\rangle$ [LWZ+17], Q# [SGT+18] and Strawberry Fields [KIQ+18] to name just a few. Such compilers allow a programmer to implement a quantum algorithm in a high-level language or API, which is then compiled to a form suitable for running on physical hardware, either real or otherwise. This low-level form is typically described as a *quantum circuit* – a straight-line program sequentially applying *quantum logic gates* and *measurements* to individual *quantum bits (qubits)*. In some cases, portable assembly-style languages representing quantum circuits have themselves been developed and adopted as a target language for compilers [SCZ16, CBSG17].

A complementary problem to the physical operation of quantum computing hardware which has likewise driven the development of quantum compilers is that of *resource estimation* [IAR13] – estimating the amount of *time* and *space* resources required to implement quantum algorithms on realistic hardware. Motivated by the need to understand the power of quantum computing in order to make policy decisions and asses cryptographic security in a post-quantum setting, quantum resource estimates have become increasingly common tools for establishing the practicality of different algorithms [GLRS16, AMG+16, SVM+17, ABL+18]. For such applications, large-scale, reliable compilation tools are particularly important, as instance sizes are typically on the order of thousands or millions of qubits and gates.

In this thesis we study two problems related to the compilation of quantum circuits: optimization and verification. A major theme running through this thesis is the development and use of *formal methods* for the analysis of quantum circuits, by which we mean the use of mathematical models to rigorously prove properties of quantum circuits related to optimization and verification. We focus on *scalability* and *practicality* – in particular, alongside theoretical investigations we develop concrete algorithms which are automated, light-weight and scalable. The techniques developed here are implemented in a circuit-level compiler infrastructure Feynman[1] inspired by LLVM [LA04], as well as ReVerC[2], a compiler from classical programs to reversible circuits. Together, they form a toolchain for the verified compilation of *oracles* – classical functions implemented on quantum hardware. We now briefly outline these contributions.

1.1 Quantum circuit optimization

The quantum circuit model forms the standard model of quantum computation [NC00], used ubiquitously to describe and implement quantum algorithms. In order to reduce the concrete resources needed to implement quantum algorithms, it is important to *optimize* circuits for various properties, depending on the particular architecture or level of abstraction. For instance, to fit a quantum circuit simultaneously using n *qubits* – the basic units of quantum information – onto a computer with only $m < n$ available qubits, optimization must be performed to reduce the number of bits in simultaneous use. Likewise, optimization may be performed to use extra available physical qubits to reduce other resources such as execution time, or to reduce the error rate by replacing low-fidelity gates with higher fidelity ones.

[1] https://github.com/meamy/feynman
[2] https://github.com/msr-quarc/ReVerC

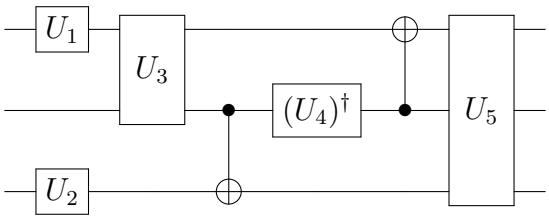


Figure 1.1: A quantum circuit diagram

Such optimizations can have a significant impact on the size of a quantum computer needed to implement an algorithm, and moreover to outperform its classical counterpart.

In the standard model, the state of an n-qubit register is defined as a unit vector in the 2^n-dimensional Hilbert space $\mathbb{C}^{2^n}$, and quantum gates are defined as elements of the unitary group $\mathrm{U}(2^n)$ for some n, acting on a state via matrix-vector multiplication. For the purposes of this thesis, a *circuit* over a particular *gate set* $\mathcal{G} \subseteq \mathrm{U}(2^n)$ will generally be a sequence of gates of $\mathcal{G}$ or their inverse – that is, a circuit is some element of the free group $\langle \mathcal{G} \rangle$. Quantum circuits are displayed diagrammatically, as shown in Figure 1.1, with horizontal lines carrying qubits from left to right and labelled boxes representing gates acting non-trivially on a subset of qubits. A circuit $C \in \langle \mathcal{G} \rangle$ is further said to *implement* the unitary operator $[\![C]\!]$, where $[\![\cdot]\!] : \langle \mathcal{G} \rangle \to \mathrm{U}(2^n)$ is the unique homomorphism into $\mathrm{U}(2^n)$ sending a gate U to itself. We call two circuits implementing the same unitary operator *equivalent*.

We define the problem of *quantum circuit optimization* over a gate set $\mathcal{G}$ and cost function $\phi : \langle \mathcal{G} \rangle \to \mathbb{R}$ as that of finding an equivalent circuit over $\mathcal{G}$ minimizing ϕ.

Problem 1.1.1 (Circuit optimization for $\mathcal{G}$, ϕ)**.** Given $C \in \langle \mathcal{G} \rangle$, find $C' \in \langle \mathcal{G} \rangle$ such that $[\![C]\!] = [\![C']\!]$ and C' minimizes $\phi(C')$.

An important distinction is whether an algorithm solves Problem 1.1.1 *exactly* – i.e. finds some circuit C' with a provably minimal cost – or as a *heuristic*, returning some circuit C' which is *better*, but not necessarily optimal with respect to ϕ. As circuit optimization is typically intractable[3] many of the methods we develop are heuristic optimizations.

Two closely related problems are those of *optimal synthesis* and *optimal mapping*, defined below.

[3]Classical logic minimization which is contained in the optimization problem over certain gate sets was recently shown to be $\sum_1^{\mathrm{P}}$-complete [BU11] and hence strictly more difficult than NP, barring a collapse in the polynomial hierarchy

Problem 1.1.2 (Optimal synthesis for $\mathcal{G}$, ϕ)**.** Given $U \in \mathrm{U}(2^n)$, find $C \in \langle \mathcal{G} \rangle$ such that $[\![C]\!] = U$ minimizing $\phi(C)$.

Problem 1.1.3 (Optimal mapping for $\mathcal{G}, \mathcal{G}'$, ϕ)**.** Given $C \in \langle \mathcal{G} \rangle$, find $C' \in \langle \mathcal{G}' \rangle$ such that $[\![C]\!] = [\![C']\!]$ minimizing $\phi(C')$.

While all three problems are closely related and frequently interchangeable, we distinguish them as each serves a distinct purpose in a quantum compiler toolchain. Optimal synthesis is typically reserved for a few qubits (e.g., [DN06, KMM13b, AMMR13, KMM13a, Sel15, RS16, PS14, BRS15]) as the size of the unitary representation scales exponentially in n, or for restricted gate sets (e.g., [PMH08]) which admit efficient representations. On the other hand, optimal mapping (assuming $\mathcal{G} \neq \mathcal{G}'$) is only applicable in compilation contexts when moving from a higher-level gate set to a lower-level (e.g., [ASD14, AADS16, Mas16]) or machine-specific one (e.g., [ZPW18]). In the either case, strict circuit optimizations are typically combined with per-gate mappings to achieve effective results (e.g., [NRS+18]).

1.1.1 Gate sets

An important factor in the circuit optimization problem is the choice of gate set. The gate sets we consider in this thesis are largely motivated by *fault-tolerant quantum computing*, which we briefly outline.

Due to the high error rate of physical qubits and gates – on the order of 10^{-3} – it is widely believed that quantum computers will require some form of error correction to execute non-trivial algorithms to reasonable accuracy. This is typically done by using an *error-correcting code* (ECC), which encodes the state of a *logical* qubit in the state of multiple *physical* qubits. As the *no-cloning* theorem [WZ82] states that a quantum state can't be copied, in contrast to classical error correction where information can be redundantly encoded by simply repeating each bit, quantum ECCs operate by encoding *basis vectors* of $\mathbb{C}^{2^n}$ according to an ECC.

As decoding a quantum state to apply gates would expose the state to un-correctable errors, fault-tolerant quantum computation combines an ECC with a discrete set of gates which can be performed *directly* on encoded values without first decoding. Such *logical* gates are typically implemented as circuits, though in modern planar codes such as *surface codes* they may be implemented in various other ways [FMMC12, HFDM12]. In either case, the *Clifford group*, generated by the *Hadamard* (H), *controlled-NOT* (CNOT) and $\pi/4$ *phase* (S) gates below, is usually implemented as primitive logic gates due to its importance

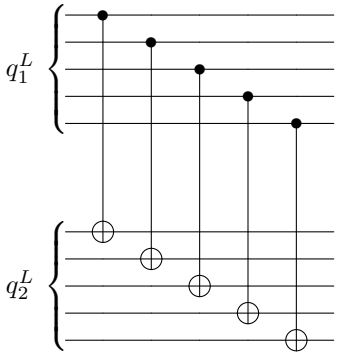


Figure 1.2: Transversal implementation of a logical CNOT gate in the 5-qubit code.

in encoding & decoding large classes of codes, hence allowing codes to be recursively applied or *concatenated*.

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \quad \text{CNOT} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix} \quad S = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$$

The generators of the Clifford group also have the convenient property that they can usually be implemented *transversally*; given an n-bit code, the logical operation U is implemented transversally with n physical copies of U, as in Figure 1.2.

As the Clifford group is not universal for quantum computing [Got98], at least one additional operation is needed to be able to approximate any unitary transformation to arbitrary accuracy. A typical choice is the $\pi/8$ phase gate ($T = \text{diag}(1, e^{\frac{i\pi}{4}})$), giving the *Clifford+T* gate set. Contrary to the Clifford group, the logical T gate is *not* transversal in most codes, and is instead implemented via *magic state distillation* and *gate teleportation*. Indeed, it is known that for any universal set of gates, at least one must be non-traversal in standard (stabilizer) codes [EK09].

The main gate set we consider is an extension of the Clifford+T gate set consisting of H, CNOT, $X = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$, and $R_Z(\theta) = \begin{pmatrix}1 & 0\\ 0 & e^{i\theta}\end{pmatrix}$ gates for any $\theta \in \mathbb{R}$. We call this gate set the *Clifford+R_Z* set, which strictly contains Clifford+T since $S = R_Z(\pi/2)$ and $T = R_Z(\pi/4)$. We collectively refer to R_Z gates as phase gates or Z-axis rotations. Such gates have applications to fault-tolerance schemes distilling higher-order rotations [CAB12], and are used in most standard quantum algorithms, including the Quantum Fourier Transform [NC00], Shor's integer factorization algorithm [Sho94] and quantum simulation [Llo96].

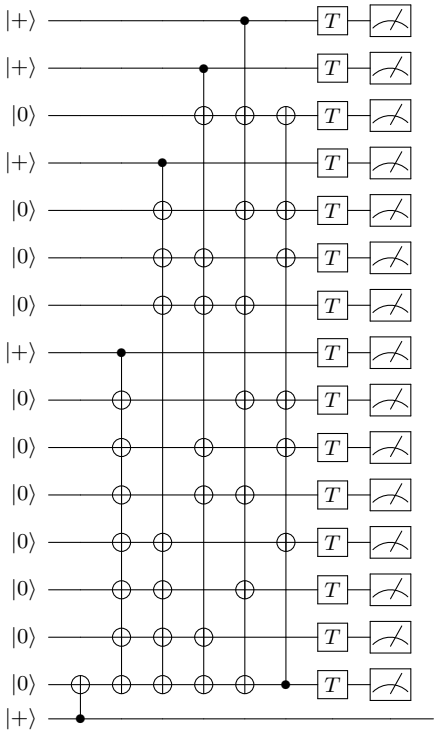


Figure 1.3: The Bravyi-Kitaev 15-to-1 A-state distillation circuit.

One other gate set we consider is the *CNOT-dihedral* gate set, consisting of CNOT, X and R_Z gates of arbitrary angle. The name CNOT-dihedral arises from the fact that for any θ of order m in $\mathbb{R}/2\pi\mathbb{Z}$, X and $R_Z(\theta)$ generate a subgroup of $\mathrm{U}(2^n)$ which is isomorphic to the dihedral group of order $2m$, up to global phase. That is, the following equations suffice to generate all equalities over X and $R_Z(\theta)$ gates:

$$X^2 = I, \qquad R_Z(\theta)^m = I, \qquad XR_Z(\theta)XR_Z(\theta) = e^{i\theta}I.$$

1.1.2 Cost functions

As with gate sets, the cost functions we consider in this thesis are largely motivated by fault-tolerance. In particular, since magic state distillation is orders of magnitude more expensive than transversal gates [BK05] (see also Figure 1.3), in the Clifford+T gate set the Clifford group is typically considered "free," and the number of T gates determine the cost of the circuit. Likewise, R_Z gates of smaller angles are implemented either via Clifford+T

approximation or, for angles of the form $1/2^k$, magic state distillation – in either case, the resulting cost dwarfs the cost of Clifford gates, or even an individual T gate [CO16].

The main types of cost functions we consider are *U-counts* – that is, the number of U or $U^\dagger$ gates in a circuit.

Definition 1.1.4. For a gate $U \in \mathcal{G}$, the U-count of a circuit $C \in \langle \mathcal{G} \rangle$ is given by

$$\phi_U(C) = \#\text{ of } U \text{ or } U^\dagger \text{ gates in } C.$$

In the case of R_Z-counts, depending on the application we consider either the total number of R_Z gates, or the number of R_Z gates *of highest multiplicative order*, as such gates are the most difficult to implement fault-tolerantly.

In this thesis we develop methods for the R_Z-count optimization of Clifford+R_Z circuits. In particular, we provide a heuristic optimization for the Clifford+R_Z gate set and the R_Z-count cost function. The heuristic allows further optimization of Clifford+R_Z circuits by solving the optimal synthesis problem for the subset of CNOT-dihedral circuits. We then solve the optimal synthesis problem for CNOT-dihedral circuits and $R_Z(\theta)$-count for any θ by giving a polynomial-time equivalence between optimal synthesis and minimum-distance decoding of certain Reed-Muller codes, and further give an efficient heuristic.

Another metric that is particularly important for near-term quantum computers is CNOT-count. Near-term, or *NISQ* [Pre18], devices operate without error correction, and hence are able to efficiently implement any single-qubit gates, including R_Z gates of arbitrary angle. On the other hand, two-qubit gates – usually the CNOT gate – are commonly the most costly in terms of both time and fidelity. To perform optimization for such applications, we study the optimal synthesis problem for CNOT-dihedral circuits with respect to CNOT-count. We characterize the problem as reducing to a certain combinatorial problem we call *parity network synthesis* in particular cases, and give an efficient heuristic based on this characterization.

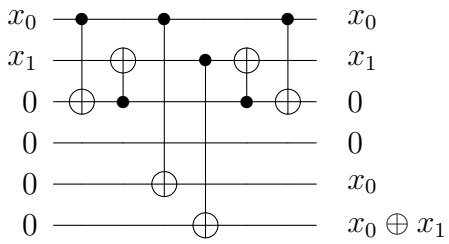
Other cost functions which are relevant to fault-tolerant circuit design are *gate depths*, and in particular the T-depth, meaning the minimum number of stages of parallel T gates in a circuit. We do not directly study T-depth optimization in this thesis, but note that previous related work by the author [AMM14] examined this problem.

1.2 Verification

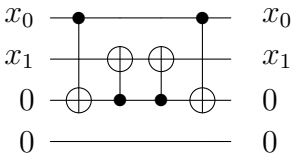
Inexorably linked with the concept of optimization is that of verification. To achieve good performance complex algorithms which are difficult to program correctly, and are sometimes

not even valid in all cases, are used to optimize software. The -O3 optimizations of the GCC compiler for instance are explicitly *not* valid for all standard-compliant C programs [GCC16], and many more bugs have been found in the less aggressive optimizations [LAS14]. Similar bugs occur in optimizing quantum compilers, such as the following semantically different circuits compiled from the same program by the REVS compiler, [PRS15] with and without optimizations enabled.

Without optimization:



With optimization:



In classical computing, for most non-critical applications testing suffices to ensure correctness. In the quantum domain, however, testing is not realistic in the vast majority of cases due to the classical difficulty of simulating quantum circuits, as well as the lack of general purpose, large-scale quantum computers. For this reason formal verification – formal proof of correctness or other properties with respect to the underlying mathematical model such as linear algebra – is crucial for asserting the correctness of quantum circuits and, by extension, resource analyses.

A question which arises in the context of quantum circuit verification is *which properties are useful to verify?* While classically, verification typically focuses on *safety properties* such as buffer overflow and out of bounds memory access, such properties are largely irrelevant to pure quantum circuits as they lack branching control and memory. In effect, for pure quantum circuits, quantum circuits cannot fail or crash; they are either correct or incorrect. We call the problem of verifying the correctness of a circuit with respect to a unitary operator *functional verification*, in line with its use in electronic design automation.

Problem 1.2.1 (Functional verification)**.** Given a unitary $U \in \mathrm{U}(2^n)$ and a circuit C, determine if $[\![C]\!] = U$.

For more complex quantum computations such as communication and cryptographic protocols, safety properties are relevant and verification methods for those have been devised (e.g., [AC04, GNP08, BCM08, FYY13, FHTZ15]). This thesis only considers the question of functional correctness, as it is directly relevant to compiling pure quantum circuits.

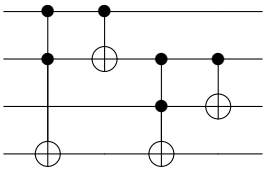
1.2.1 Functional verification

In order to verify the functionality of a circuit, the unitary U must first be specified in some formal mathematical model, for instance a matrix; moreover, the specification *must itself be verified* to ensure that it correctly specifies the intended functionality. Whether this verification is performed via testing of the design or automated generation from some higher-level specification language, human insight is necessarily required to ensure that the specification is correct.

Consider, for example, integer addition of two registers. Even in the case of a one-bit register, the unitary matrix is

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix},$$

which is effectively incomprehensible by humans, and hence unsuitable as a specification – in classical logic design, such a specification corresponds to a *truth table*, which is typically generated from a human-readable logical expression. An alternative is a circuit written over a higher-level gate set, rather than a low-level implementation circuit, such as



Functional verification against circuit-like specifications is called *equivalence checking*, and has seen extensive research (e.g., [WGMD09, YM10, CD11, DL13, GD17, RPZ17]). However, when viewed as a direct (human-level) specification, higher-level circuits are still error-prone. As quantum programming languages typically take the form of *circuit description languages*, human-readable specifications of quantum circuits and algorithms are not yet a reality.

On the verification side, a mathematical model which is *as abstract as possible* – in the sense that implementation details are abstracted away – while remaining efficient to represent

and manipulate is needed in order to scale to useful sizes. A model which has recently been applied to the verification of quantum circuits is the ZX-calculus [CD11, DL13, GD17]. Such graphical models abstract away many syntactic details of circuits, including structural properties of the underlying symmetric monoidal category and motivating the tagline *only the topology matters*, while still remaining efficiently representable. However, re-writing graphs is generally difficult [BGK+16] and these systems typically require human intervention to complete proofs due to a lack of natural direction for proofs to proceed [DL13, GD17].

We develop a formal model – based on the Feynman path integral, which also factors heavily in our optimization work – for quantum circuits which doubles (with some syntactic sugar) as a natural specification language. Using this mathematical model, we give a functional verification algorithm and demonstrate its ability to verify large quantum circuits directly against their functional specifications. Specifically, we provide an automated quantum circuit verification method which allows verification against a *logical* instead of a *circuit-like* description of a quantum computation.

1.2.2 Compiler verification

In contrast to functional verification, which is useful for asserting the correctness of a circuit with respect to a human-readable, logical specification, it is often desirable to simply ensure that a compiler has not *introduced* errors into a program which is otherwise assumed to be correct. While functional verification can double for this purpose via equivalence checking between a source program and the compiled circuit, such methods are typically not scalable and moreover need to be applied individual to *every* compiled circuit. Instead, it is preferable to directly validate that the compiler itself is correct and never introduces errors. When this certification is performed formally, the process is called *compiler verification*.

The idea of formally proving the correctness of a compiler was developed in the late 60's by McCarthy and Painter [MP67], leading up to the first machine-checked proofs of correctness in the early 70's by Diffie, Milner and Weyhrauch [MW72]. However, the advancements in *proof assistants* have recently led to an explosion in popularity of the technique, spurred by Leroy's formally verified, optimizing C compiler CompCert [Ler06]. The novel insight of Leroy, facilitated by these advancements, was to write the compiler, specification, and formal proof *all in the same language* – for instance, Coq [Coq17] – allowing the *implementation* of the compiler to be directly verified, as opposed to just the verification algorithm as in [MW72].

As an illustration, an implementation of the factorial function, together with a proof that the factorial function always returns a positive integer on non-negative integers, is

```
val factorial : nat -> nat
let rec factorial x = match x with
        | 0 -> 1
        | _ -> x * factorial (x - 1)

val factorial_is_pos: (x:nat) -> Lemma (factorial x > 0)
let rec factorial_is_pos x = match x with
        | 0 -> () (* 0 > 0 *)
        | _ -> factorial_is_pos (x - 1)
                  (* factorial (x - 1) > 0 => x * factorial (x - 1) > 0 *)
```

given in the proof assistant F⋆ [SHK+16] below. The type of `factorial_is_pos` specifies that the *program computing the factorial function* maps natural numbers to non-negative integers, while the implementation of `factorial_is_pos` functions as a proof which is machine checked by the F⋆ compiler. In contrast to proof assistants based more directly on *constructive logic* such as Coq, the F⋆ compiler uses an SMT solver to machine check proofs, allowing proofs to be partially specified as above.

We follow the methodology laid out by Leroy [Ler06] to develop a formally verified optimizing compiler for reversible circuits. The compiler ReVerC compiles a classical irreversible language Revs [PRS15] to reversible circuits. A notable aspect of our compiler is that it is built around *partial evaluation*, which offloads most of the proof obligations to a tiny two-instruction language. In practice, this allowed the formal proof to be carried out with a low (see, e.g., [Ler06]) ratio of approximately 1:1 lines of code and programmer time compared to the development of the compiler proper.

1.3 Contributions

In summary, the main research contributions of this thesis are:

- An algorithm for optimizing quantum circuits which merges Z-axis rotation gates and groups them into CNOT-dihedral sub-circuits for further optimization.
- A proof of the polynomial-time equivalence of R_Z-count minimization of CNOT-dihedral circuits and minimum distance decoding of certain Reed-Muller codes, along with an associated optimization algorithm.

- A characterization of CNOT-count minimization of CNOT-dihedral circuits as minimal parity network synthesis. The minimal parity network synthesis problem is shown to be NP-complete for two restricted cases, and an efficient heuristic is given.
- A mathematical model for the formal specification of quantum circuits, along with a verification algorithm which is complete for Clifford circuits and scales to Clifford+R_Z circuits with hundreds of qubits.
- A formally verified reversible circuit compiler which ties together both verification and optimization by optimizing for the total space usage of compiled circuits.

The contributions above are published chronologically in [AM16, ARS17, AAM18, Amy18]. In addition to the work described here, other publications completed during my doctoral studies include [AADS16, AASD16, AMG+16, ACJR17, KIQ+18].

1.4 Outline

This thesis is divided into three parts, which are briefly outlined below.

- Part I, including chapters 1 to 3 covers background material necessary for this thesis. In Chapter 2 we describe the circuit model of quantum computing and its standard interpretation in linear algebraic terms. Chapter 3 introduces the *path integral* model of quantum circuits, which underlies many of the theoretical results in this thesis.
- Part II (chapters 4 to 6) covers optimization algorithms for quantum circuits. Chapter 4 introduces a general optimization algorithm called *phase folding* to merge Z-axis rotations in arbitrary quantum circuits. The remaining chapters examine the problem of optimal synthesis of CNOT-dihedral circuits, which arises as a sub-problem of the phase folding algorithm, from the perspective of R_Z-count (Chapter 5) and CNOT-count (Chapter 6) optimization.
- Part III (chapters 7 and 8) studies the problem of verifying compiled quantum circuits. In Chapter 7 a framework for the functional specification and verification of quantum circuits is developed, along with a concrete verification algorithm for circuits in the Clifford hierarchy. Chapter 8 culminates with the formal verification of an optimizing reversible circuit compiler, compiler circuits which are provably correct implementations of their source.

Chapter 2

Quantum computation

In this chapter we review the relevant background on quantum computation and the standard linear algebraic model of quantum circuits. We begin by covering the standard model of quantum computation, then cover relevant aspects of quantum circuits followed by discussions of reversible computation.

2.1 The standard model

In the standard model of quantum computation, the state of a *qubit* – a unit of quantum information – is described as a unit vector in $\mathbb{C}^2$. We use Dirac (*Bra-Ket*) notation, where $|\psi\rangle$ denotes a vector in $\mathbb{C}$ and $\langle\psi|$ denotes its complex conjugate $|\psi\rangle^\dagger$; the vector space inner product is further written as $\langle\varphi|\psi\rangle$. As is customary we fix a basis $\{|0\rangle = e_1, |1\rangle = e_2\}$ of $\mathbb{C}^2$ called the *computational basis*. The states $|0\rangle$ and $|1\rangle$ are known as the *classical* states, and the state of a qubit $|\psi\rangle$ is said to be a *superposition* of classical states if

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$$

where α and β are non-zero complex numbers. As the state of a qubit is required to be a unit vector in $\mathbb{C}^2$, we have that $|\alpha|^2 + |\beta|^2 = 1$.

The state of n qubits is likewise given by a unit vector in the 2^n-dimensional complex vector space $\mathbb{C}^{2^n}$. We denote the states of the computational basis by $|\mathbf{x}\rangle = |x_1 x_2 \dots x_n\rangle$ where $\mathbf{x} \in \mathbb{F}_2^n$ and $\mathbb{F}_2 = (\{0,1\}, \oplus, \cdot)$ is the binary field – when confusion is unlikely to arise, we may also use $|i\rangle$ to refer to the classical state corresponding to the n-digit binary

expansion of i. States of distinct subsystems $|\psi_1\rangle \in \mathbb{C}^{2^n}$ and $|\psi_2\rangle \in \mathbb{C}^{2^m}$ may be combined by taking their *tensor* product $|\psi_1\rangle \otimes |\psi_2\rangle \in \mathbb{C}^{2^{n+m}}$, defined as the standard vector space tensor product:

$$|\psi_1\rangle \otimes |\psi_2\rangle = \left(\sum_i \alpha_i |i\rangle\right) \otimes \left(\sum_j \beta_j |j\rangle\right) = \sum_{i,j} \alpha_i \beta_j (|i\rangle \otimes |j\rangle).$$

The tensor product of two basis states $|\mathbf{x}\rangle$, $|\mathbf{y}\rangle$ is defined as the basis state labelled by concatenation $\mathbf{xy}$ of $\mathbf{x}$ and $\mathbf{y}$:

$$|\mathbf{x}\rangle \otimes |\mathbf{y}\rangle = |\mathbf{xy}\rangle = |x_1 x_2 \dots x_n y_1 y_2 \dots y_m\rangle.$$

A multi-qubit state $|\psi\rangle$ which can be written as a tensor product of one-qubit states is said to be *separable*. Conversely, a state which is not separable is said to be *entangled*.

Example 2.1.1. The Bell state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

is not separable.

We typically drop the $\otimes$ and simply write $|\psi_1\rangle|\psi_2\rangle$ to refer to the tensor product of $|\psi_1\rangle$ and $|\psi_2\rangle$. As the tensor product is symmetric – e.g., there exists a natural isomorphism $\sigma_{n,m}$ between $\mathbb{C}^n \otimes \mathbb{C}^m$ and $\mathbb{C}^m \otimes \mathbb{C}^n$ – we assume the order of subspaces can be arbitrarily rearranged.

A quantum state evolves according to some *unitary* transformation $U : \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$. By unitary we mean an invertible linear operator U on $\mathbb{C}^{2^n}$ such that $U^\dagger = U^{-1}$, where $U^\dagger$ is the conjugate-transpose of U, obtained by transposing U and taking the complex conjugate of each entry. Equivalently, U is a linear operator that preserves the Euclidean norm, and hence maps unit vectors to unit vectors. We write $\mathrm{U}(d)$ to denote the set of unitary operators on a complex vector space of dimension d. As with states, unitaries $U \in \mathrm{U}(2^n), V \in \mathrm{U}(2^m)$ operating on distinct subsystems may be combined with the vector space tensor product $U \otimes V \in \mathrm{U}(2^{n+m})$, where it can be readily verified that

$$(U \otimes V)(|\psi_1\rangle \otimes |\psi_2\rangle) = U|\psi_1\rangle \otimes V|\psi_2\rangle.$$

A qubit or system of qubits may also be *measured* in some orthonormal basis $\{|b_i\rangle\}$ of $\mathbb{C}^{2^n}$. If $|\psi\rangle = \sum_i \alpha_i |b_i\rangle$, then (non-destructive) measurement returns the outcome b_i

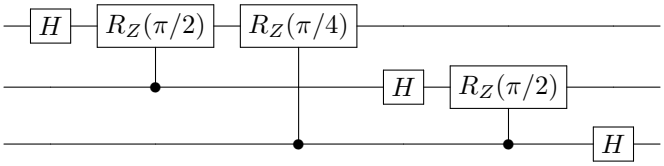


Figure 2.1: An example of a quantum circuit, implementing the Quantum Fourier Transform.

and leaves the qubit in the state $|b_i\rangle$ with probability $|\alpha_i|^2$. If only a subset of a quantum state is measured, then when $|\psi\rangle = \sum_i \alpha_i |b_i\rangle|\psi_i'\rangle$ where each ψ_i is a normalized quantum state, measurement returns the outcome b_i and leaves the system in the state $|b_i\rangle|\psi\rangle$ with probability $|\alpha_i|^2$. Measurement is typically performed in the computational basis, with measurement in other bases being emulated by applying unitary transformations first.

2.2 Quantum circuits

Quantum computations are typically described via *circuit diagrams* in analogy to classical computing. Such diagrams carry qubits along horizontal lines called *wires* into unitary gates and measurements which transform their state. Figure 2.1 gives an example of a quantum circuit diagram. We restrict our attention to *pure* quantum circuits – those not containing measurements.

Formally, given a finite set of *gates* $\mathcal{G} \subseteq \bigcup_n \mathrm{U}(2^n)$ called a *gate set*, each acting on a non-zero number of qubits called its *arity* and denoted $\mathrm{ar}(U), U \in \mathcal{G}$, we can construct circuits C over $\mathcal{G}$ recursively as terms of the form

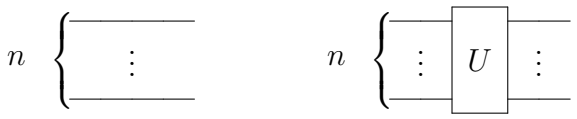
$$C := I_n, n \in \mathbb{N} \mid \sigma_{n,m}, n, m \in \mathbb{N} \mid U, V \in \mathcal{G} \mid C^\dagger \mid C \fatsemi C' \mid C \otimes C'.$$

The notation $\fatsemi$ denotes left-to-right functional composition, in contrast to the standard right-to-left composition $\circ$. We extend arity to circuits by defining

$$\begin{aligned} \mathrm{ar}(I_n) &= n \\ \mathrm{ar}(C^\dagger) &= \mathrm{ar}(C) \\ \mathrm{ar}(C \fatsemi C') &= \mathrm{ar}(C) \\ \mathrm{ar}(C \otimes C') &= \mathrm{ar}(C) + \mathrm{ar}(C'). \end{aligned}$$

We say a circuit is *well-formed* if for every sub-term $C \fatsemi C'$, $\mathrm{ar}(C) = \mathrm{ar}(C')$.

Diagrammatically, we write the *identity* circuit I_n as a n horizontal lines, and an n-qubit gate U as a box with n input and output lines:



We typically drop the subscript from I. The tensor product $U \otimes V$ is written as *vertical* composition, while the (left-to-right) functional composition $U \fatsemi V$ is written as *horizontal* composition, shown respectively below:

As is typical we assume certain structural properties of quantum circuits hold – in particular, that $\dagger$ and $\fatsemi$ form a group, $\otimes$ is associative, and that for any U, U', V, V' the *bifunctorial law*

$$(U \fatsemi V) \otimes (U' \fatsemi V') = (U \otimes U') \fatsemi (V \otimes V')$$

holds. Further, we assume that the $\sigma_{n,m}$ gate, swapping the first n wires of a diagram with the last m, satisfies

$$\sigma_{n,m} \fatsemi (I_m \otimes C) = (C \otimes I_m) \fatsemi \sigma_{n,m}.$$

We use this swap operator to allow the application of multi-qubit gates to non-sequential qubits – in particular, we write $U_{i_1,i_2,\ldots,i_k}$ to denote an n-qubit gate U of arity k applied to qubits i_1 through i_k. Concretely, the notation $U_{i_1,i_2,\ldots,i_k}$ corresponds to some circuit $\sigma^\dagger \fatsemi U \otimes I_{n-k} \fatsemi \sigma$, where σ is some sequence of swaps mapping $1 \mapsto i_1, \ldots, k \mapsto i_k$.

Using the above assumptions – corresponding to a *symmetric monoidal groupoid* – any n-qubit circuit over $\mathcal{G}$ can be written as an element of the freely generated group $\langle \mathcal{G}_n \rangle$, where $\mathcal{G}_n$ is the n-qubit completion of $\mathcal{G}$, consisting of the images $U_{i_1,i_2,\ldots,i_k}$ of each gate on n qubits. We write $\langle \mathcal{G} \rangle$ to refer to well-formed circuits over $\mathcal{G}$ and use either the symmetric monoidal $(\mathcal{G}, \fatsemi, \otimes, \dagger)$ or group $(\mathcal{G}_n, \fatsemi, \dagger)$ presentations depending on which is most convenient in the particular context.

Semantics The *semantics* of a circuit defines its meaning as a mathematical object of some domain. Typically, the semantics of a unitary circuit is taken as the unitary matrix obtained by interpreting $\otimes$, $\fatsemi$, and $\dagger$ as their vector-space equivalents – we call this the *standard interpretation* of a circuit.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \qquad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix},$$

$$\mathrm{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \qquad R_Z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, \qquad R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{bmatrix}.$$

Figure 2.2: Standard quantum gates & their matrix interpretations

Definition 2.2.1 (standard interpretation)**.** The standard interpretation of an circuit $C \in \langle \mathcal{G} \rangle$ for some $\mathcal{G} \subseteq \bigcup_{n \in \mathbb{N}} \mathrm{U}(2^n)$ is denoted $[\![C]\!] \in \mathrm{U}(2^{\mathrm{ar}(C)})$, defined recursively as

$$\begin{aligned} [\![I_n]\!] &= \underbrace{I \otimes \cdots \otimes I}_{n \text{ times}} \\ [\![\sigma_{n,m}]\!] &= \sigma_{n,m}, \\ [\![U]\!] &= U \\ [\![C^\dagger]\!] &= [\![C]\!]^\dagger \\ [\![C \fatsemi C']\!] &= [\![C']\!][\![C]\!] \\ [\![C \otimes C']\!] &= [\![C]\!] \otimes [\![C']\!] \end{aligned}$$

Standard gates Standard gates we will use throughout this thesis are listed in Figure 2.2. We now describe several important gate sets formed from subsets of the listed gates.

The well-known *Clifford group* on n qubits, $\mathcal{C}_n$, arises as the normalizer of the n-qubit *Pauli group* $\mathcal{P}_n = \langle X_i, Y_i, Z_i \mid i \in [n] \rangle$. In particular,

$$\mathcal{C}_n = \{U \in \mathrm{U}(2^n) \mid U \mathcal{P}_n U^{-1} \subseteq \mathcal{P}_n\}.$$

A minimal generating set for $\mathcal{C}_n$ is $\{H, S, \mathrm{CNOT}\}$, though for practical reasons we typically include X in the Clifford gate set.

By adjoining the T gate to the Clifford gate set, we generate the Clifford+T group, on n qubits denoted $\mathcal{C}_n^3$. Again, the Clifford+T group arises as the normalizer of the Clifford group – this recursive construction is called the *Clifford hierarchy*, where $\mathcal{C}_n^2 = \mathcal{C}_n$ and $\mathcal{C}_n^1 = \mathcal{P}_n$.

While for $k > 3$, $\mathcal{C}_n^k$ does not form a group [CGK17], it is known that $R_k = R_Z(2\pi/2^k) \in \mathcal{C}_n^k$ but not $\mathcal{C}_n^{k-1}$ for any k, hence we refer to circuits over $\{H, \mathrm{CNOT}, X, R_k\}$ as *Clifford hierarchy* circuits. Note that for any k, $R_k^{2^k} = I$, and in particular 2^k is the least such exponent, hence we say that R_k has *order* 2^k in $\mathrm{U}(2^n)$. Equivalently, since $R_Z(2\pi) = I$, we say that $\theta = 2\pi/2^k$ has order 2^k in the additive group $\mathbb{R}/2\pi\mathbb{Z}$. If we adjoin a Z-axis rotation of any *non even-power order* to the Clifford group, we get a universal gate set which is *not* contained in the Clifford hierarchy. We generally call such circuits *Clifford+R_Z* circuits.

One other relevant gate set is the CNOT-dihedral gate set $\{\mathrm{CNOT}, X, R_Z(\theta) \mid \theta \in \mathbb{R}\}$. In the standard interpretation – and indeed in any symmetric monoidal groupoid – the image of all n-qubit circuits over $\{\mathrm{CNOT}, X, R_Z(\theta)\}$ *for a fixed θ of order k* is isomorphic to the dihedral group of order $2k$. We call circuits generated by $\{\mathrm{CNOT}, X, R_Z(\theta)\}$ with $R_Z(\theta)$ of order k the CNOT-dihedral group of order $2k$.

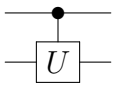
Two other gates which are relevant include the *Toffoli* gate TOF, and the *doubly-controlled Z* gate CCZ, given as matrices below:

$$\mathrm{TOF} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \qquad CCZ = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

The TOF and CCZ gates, as well as the CNOT gate, are instances of *controlled* gates. For instance the CNOT gate computes the transformation

$$\mathrm{CNOT}(\alpha|0\rangle|\psi\rangle + \beta|1\rangle|\psi'\rangle) = \alpha|0\rangle|\psi\rangle + \beta|1\rangle(X|\psi'\rangle),$$

effectively applying the X gate to the second qubit – called the *target* – if the first – the *control* – is in the state $|1\rangle$, and the identity transformation otherwise. Likewise, the TOF and CCZ gates apply X and Z, respectively, "controlled" on the value of the first two qubits both being 1. We use the special graphical notation



to denote a controlled-U gate with the top qubit acting as the control and the bottom qubit as the target. In the context of controlled-X gates (e.g., CNOT, TOF) we typically denote the X gate as

 .

2.3 Reversible computation

An important subset of quantum circuits are *reversible circuits*. Such circuits implement self-inverse mappings from computational basis states to other computational basis states, and hence correspond to classical computations which can be inverted by *reversing* the computation. We say a gate is a *reversible* gate if its standard interpretation is a permutation matrix.

Example 2.3.1. The X, CNOT and TOF gates are all reversible gates. As functions on computational basis states,

$$X : |x\rangle \mapsto |1 \oplus x\rangle, \quad \text{CNOT} : |x\rangle|y\rangle \mapsto |x\rangle|x \oplus y\rangle, \quad \text{TOF} : |x\rangle|y\rangle|z\rangle \mapsto |x\rangle|y\rangle|z \oplus xy\rangle,$$

where $\oplus$ denotes addition in $\mathbb{F}_2$.

It can be readily observed that not all classical functions from n to m bits are reversible, as for instance the AND function $x \wedge y$ is not invertible. Even restricted to classical functions of the form $\{0,1\}^n \to \{0,1\}^n$, not all such functions can be reversed. However, with the addition of *ancillas* – extra qubits initialized in the $|0\rangle$ state – we can recover the full power of classical computing by emulating the functionally complete NAND and FANOUT gates with X, CNOT and TOF gates:

$$(I \otimes I \otimes X)\text{TOF} : |x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|1 \oplus xy\rangle$$
$$\text{CNOT} : |x\rangle|0\rangle \mapsto |x\rangle|x\rangle$$

A consequence of using ancillas to implement arbitrary classical circuits is that the space usage is linear in the size of the circuit. While we could simply measure temporary values after they are no longer needed to project the ancilla back into the $|0\rangle$ or $|1\rangle$ state, thus freeing the ancilla up to be reused later, if the qubit is entangled with the rest of the system it may affect the state. On the other hand, if entangled ancillas are left around they may *decohere* on their own, reducing the fidelity of the entire state.

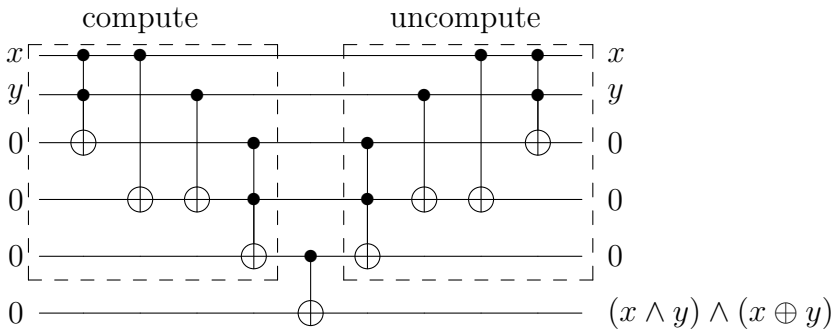
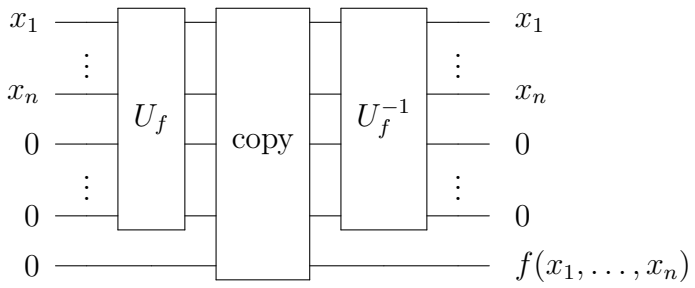
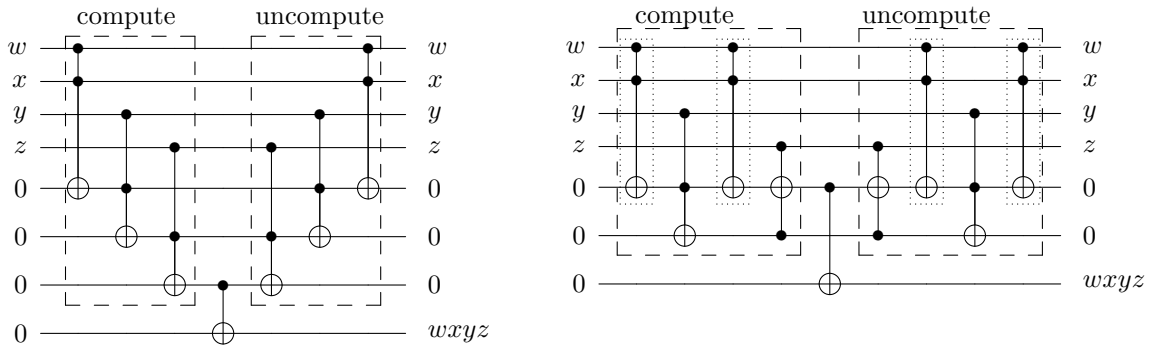


Figure 2.3: A reversible circuit uncomputing all temporary ancillas.

Bennett [Ben73] showed that rather than measuring a temporary value, temporary bits can be returned to the zero-state by copying out the result and then running the circuit in reverse – in effect *uncomputing* the temporary values and freeing any allocated ancillas. Figure 2.3 gives an example of this compute-copy-uncompute sequence for the Boolean expression $(x \wedge y) \wedge (x \oplus y)$. In general, Bennett's method can be summarized by the diagram below, where copy only involves CNOT gates targeting the output register:



The choice of *when* to clean temporary bits allows a trade-off between time and space [Ben89]. For instance, the circuit below on the left computes the product $wxyz$ with only one final round of cleanup, while the circuit on the right applies intermediate cleanup of the product xy. Though still not optimal, the result is a reduction of one ancilla, at the cost of doubling the number of compute-uncompute cycles for the first sub-expression xy. Such trade-offs are an instance of *pebble games*, and it can be noted that given a sequence of k intermediate expressions, if each expression is uncomputed immediately after use, the ith sub-expression will be computed or uncomputed 2^{k-i} times.



Classical linear operators A special subclass of reversible computations are those which are *linear* or otherwise *affine* permutations. Specifically, an n-qubit reversible circuit implements a linear (resp. affine) permutation if it maps a basis state $|\mathbf{x}\rangle$ to $|A\mathbf{x}\rangle$ for some invertible linear (resp. affine) operator A on $\mathbb{F}_2^n$. We denote by $\mathrm{GL}(n, \mathbb{F}_2)$ the *general linear group* of invertible $n \times n$ matrices over $\mathbb{F}_2$, and likewise $\mathrm{GA}(n, \mathbb{F}_2) = \mathbb{F}_2^n \rtimes \mathrm{GL}(n, \mathbb{F}_2)$ denotes the general affine group, where $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$ acts on $\mathbf{x} \in \mathbb{F}_2^n$ as $A\mathbf{x} + \mathbf{b}$ (note that $\mathbf{b} \in \mathbb{F}_2^n$ gives the affine subspace). The CNOT and X gates implement linear and affine permutations, respectively, while the Toffoli gate is a *non-linear* permutation.

It will sometimes be useful to refer to *non-invertible* linear or affine operators on $\mathbb{F}_2$. We denote by $\mathrm{L}(\mathbb{F}_2^n, \mathbb{F}_2^m)$ the space of linear operators from $\mathbb{F}_2^n \to \mathbb{F}_2^m$ – that is, m by n Boolean matrices. Likewise, $\mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^m) = \mathbb{F}_2^m \rtimes \mathrm{L}(\mathbb{F}_2^n, \mathbb{F}_2^m)$ denotes the space of affine operators.

Chapter 3

Path integrals

In this chapter, we introduce the *path integral* model of quantum computation, which has proven useful for computational methods in Quantum Field Theory and underlies much of the formal reasoning in this thesis. In particular, we give an algebraic account of the *sum-over-paths action* of a quantum circuit, which can then be analyzed and manipulated as a mathematical object. We begin by describing the background and intuition of the path integral formulation, then introduce the path integral representations of Clifford+R_Z circuits we will use in various forms throughout this thesis. We close with a discussion of representations of the *phase function* of a path integral, particularly by their *Fourier expansions* which have a close connection to the structure of Clifford+R_Z circuits.

3.1 The Path Integral formulation

The path integral formulation of quantum mechanics, formalized by Feynman in the 40's [FH65], serves as an alternate, equivalent formulation of quantum mechanics to the standard model. In a general sense, the idea is to describe the probability amplitude of the transition from one state to another (say, the position of a particle) by a sum over all possible transitions between intermediate states or *paths* leading to that state. Figure 3.1 shows possible trajectories of a particle moving from states A to B; in the path integral formulation, the final amplitude is the equal-weighted sum of the *phase* – $e^{i\theta}$ for some $\theta \in \mathbb{R}$ – acquired along each such path.

As a quantum process, the action of a quantum circuit can likewise be described as a sum over all transitions between intermediate states. However, as quantum gates are

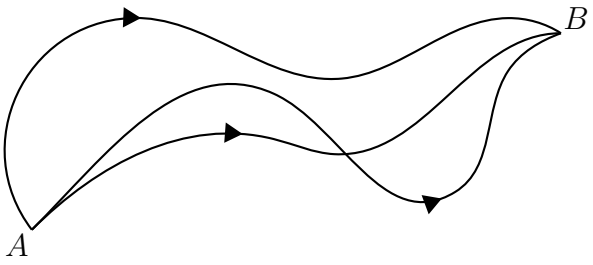


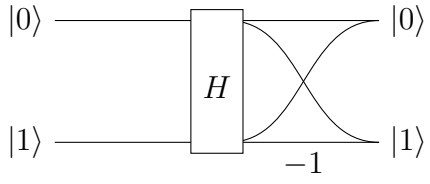
Figure 3.1: The paths of a particle from point A to B.

typically discretized and modelled as operators on a finite Hilbert space ($\mathbb{C}^{2^n}$), a discrete sum is used rather than a continuous integral.

Viewing the computational basis states as the physical states of a qubit system, a unitary operator sends a particular state to a sum of states, each with a particular amplitude – this process can be viewed as an initial state (possibly in a superposition) taking some superposition of paths to other states. For instance, consider the Hadamard gate H, modelled in the standard interpretation by the unitary matrix

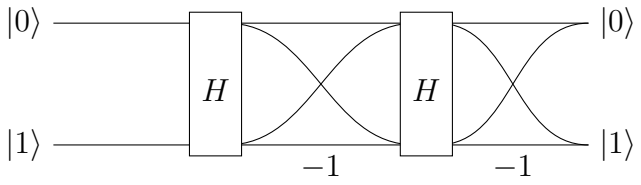
$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

In effect, the Hadamard gate sends a single basis state $|0\rangle$ or $|1\rangle$, along two distinct, equal-weight paths to states $|0\rangle$ and $|1\rangle$ with phases 1 and $(-1)^x$, respectively. Visually, we can represent the paths each input state takes together with the phase acquired along each segment as



.

The effect of *interference* between paths becomes clear when by composing the outputs with a second Hadamard gate, corresponding to the circuit HH, which in the standard interpretation is the identity matrix:



.

To compute the amplitude with which the state $|0\rangle$ transitions to state $|0\rangle$ – i.e. $\langle 0|HH|0\rangle$ – we simply sum the phases along each path from $|0\rangle$ to $|0\rangle$ and scale by $\frac{1}{2}$, since each path has amplitude $\frac{1}{2}$. It can be easily verified that the total amplitude is $\frac{1}{2}(1+1) = 1$. Likewise, if we sum the phases along all paths from $|0\rangle$ to $|1\rangle$, the result is an amplitude of $\frac{1}{2}(1-1) = 0$ as expected. In this case, the paths leading to $|1\rangle$ *destructively interfere* while the paths to $|0\rangle$ *constructively interfere*.

3.1.1 Sum-over-paths actions

While path integrals can be used to a particular transition amplitude – that is, $\langle \mathbf{x}'|U|\mathbf{x}\rangle$ – as a sum of all internal paths from the input state to the output state, a general unitary operator involves the transition amplitudes between *all* input and output states. Rather than give a separate path integral for each transition, it is possible to give a self-contained path integral account of unitary dynamics as a collection of paths Π between basis states, together with a set of amplitudes $\phi : \Pi \to \mathbb{C}$. This set of paths and amplitudes naturally corresponds to the operator

$$U = \sum_{\pi:\mathbf{x}\to\mathbf{x}'\in\Pi} \phi(\pi)|\mathbf{x}'\rangle\langle\mathbf{x}|,$$

where $\pi : \mathbf{x} \to \mathbf{x}'$ denotes that π is a path from $|\mathbf{x}\rangle$ to $|\mathbf{x}'\rangle$. Writing the operator above by its action on a basis state,

$$U : |\mathbf{x}\rangle \mapsto \sum_{\pi:\mathbf{x}\to\mathbf{x}'\in\Pi_{\mathbf{x}}} \phi(\pi)|\mathbf{x}'\rangle,$$

we call it the *sum-over-paths action* defined by Π and ϕ, where $\Pi_{\mathbf{x}} \subseteq \Pi$ gives the paths starting at $|\mathbf{x}\rangle$.

Any unitary matrix $U \in \mathrm{U}(2^n)$ can also be described as a sum-over-paths action by letting Π contain one path $\pi_{\mathbf{x},\mathbf{x}'} : \mathbf{x} \to \mathbf{x}'$ for each $\mathbf{x}, \mathbf{x}' \in \mathbb{F}_2^n$, and $\phi(\pi_{\mathbf{x},\mathbf{x}'}) = \langle \mathbf{x}'|U|\mathbf{x}\rangle$. Then

$$U : |\mathbf{x}\rangle \mapsto \sum_{\mathbf{x}'\in\mathbb{F}_2^n} \left(\langle \mathbf{x}'|U|\mathbf{x}\rangle\right)|\mathbf{x}'\rangle = \sum_{\pi:\mathbf{x}\to\mathbf{x}\in\Pi_{\mathbf{x}}} \phi(\pi)|\mathbf{x}'\rangle.$$

In this way, the path integral and linear algebraic views are complimentary; either can be encoded in the other.

3.1.2 Composition

One of the advantages of path integrals for computational problems is that an evaluation strategy for quantum amplitudes is not fixed *a priori*. In particular, functional composition

can be defined by taking the familiar *relational composition* of paths

$$\Pi \fatsemi \Pi' = \{(\pi \fatsemi \pi') : \mathbf{x} \to \mathbf{x}' \mid \exists \mathbf{x}''.\pi : \mathbf{x} \to \mathbf{x}'' \in \Pi \wedge \pi' : \mathbf{x}'' \to \mathbf{x}' \in \Pi'\}.$$

That is, the resulting collection of paths is given by connecting the outputs of paths in Π with the inputs of paths in Π'. The amplitude of each path is likewise the product of the amplitudes of each segment,

$$\phi \fatsemi \phi' : \pi \fatsemi \pi' \mapsto \phi(\pi) \cdot \phi'(\pi').$$

The sum-over-paths action defined by composing path integrals in this way unsurprisingly coincides with the composition of their respective linear operators U, V:

$$VU : |\mathbf{x}\rangle \mapsto \sum_{\pi:\mathbf{x}\to\mathbf{x}'\in(\Pi\fatsemi\Pi')_{\mathbf{x}}} (\phi \fatsemi \phi')(\pi)|\mathbf{x}'\rangle.$$

In effect, the composed sum-over-paths action above delays all evaluation of amplitudes, in contrast to composition in the standard model (matrix multiplication) which directly evaluates the sum of interfering paths. This idea of *lazy evaluation* of amplitudes was previously used to show the containment of the *bounded-error quantum polynomial time* complexity class (**BQP**) in **PSPACE** by Bernstein and Vazirani [BV97], **PP** by Adleman, DeMarrais, and Huang [ADH97] and later **AWPP** by Fortnow and Rogers [FR99], allowing analysis of the complexity of quantum amplitudes.

Tensor composition is similarly intuitive in the path integral view, corresponding to a product[1] of all paths $\Pi \times \Pi'$ and their amplitudes $\phi \times \phi' : \Pi \times \Pi' \to \mathbb{C}$:

$$U \otimes V : |\mathbf{x}\rangle|\mathbf{y}\rangle \mapsto \sum_{\pi:(\mathbf{x},\mathbf{y})\to(\mathbf{x}',\mathbf{y}')\in(\Pi\times\Pi')_{(\mathbf{x},\mathbf{y})}} (\phi \times \phi')(\pi)|\mathbf{x}'\rangle|\mathbf{y}'\rangle.$$

3.1.3 Representation

The other major advantage of the path integral formulation appears when restricted to paths which have algebraically "nice" amplitudes. In these cases, the sum-over-paths action of a quantum process may be given by a simple algebraic description.

For instance, recall that the Hadamard gate maps an initial state $|x\rangle$ to the equal-weight superposition of states $|x'\rangle$, $x' \in \{0, 1\}$, together with a phase in $\{1, -1\}$. We call a gate where each path has equal weight *balanced*; in this case, only the phase of a path is needed

[1] More formally, as the collection of paths may not generally be separable we need a monoidal product.

to compute its transition amplitude. Over gate sets consisting solely of balanced gates, every path contributes equal amplitude but with a different phase, hence the sum-over-paths action may be given by a set of paths together with a *phase function* $f : \Pi \to \mathbb{R}$ – i.e.

$$|\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{|\Pi|}} \sum_{\pi:\mathbf{x}\to\mathbf{x}'\in\Pi_{\mathbf{x}}} e^{if(\pi)}|\mathbf{x}'\rangle.$$

Since the Hadamard gate induces a branching into two output states, $|0\rangle$ and $|1\rangle$, the paths in $\Pi_{\mathbf{x}}$ may also be indexed by the particular branch taken and the output state given as a function of this branch. Writing the action of H as a sum over the branches $x' \in \mathbb{F}_2$ with output state x' and writing the phase as a function of the initial state x and x', we arrive at the familiar expression

$$H : |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{x'\in\mathbb{F}_2} (-1)^{xx'}|x'\rangle.$$

We call x' a *path variable*, as its values index the different computational paths taken simultaneously by the Hadamard gate.

The relational composition of sets of paths indexed by path variables can likewise be indexed by *all* of the component variables. However, as some paths are not feasible – for instance, the composition of $\pi : 0 \to 0$ and $\pi' : 1 \to 1$ – care must be taken to ensure the correct paths are composed. Intuitively, this can be achieved by setting the initial state of the latter path to be equal to the output (as a function of the path variables) of the former. For instance, the Hadamard gate can be composed with itself as below:

$$HH : |x\rangle \mapsto \frac{1}{2} \sum_{x'',x'\in\mathbb{F}_2} (-1)^{xx''+x''x'}|x'\rangle.$$

We call the path variable x'' an *internal* variable, as the output of a path does not depend on x''; equivalently, paths which differ only in values of internal variables interfere. We call path variables which are not internal *external* variables.

The use of variables to index paths appears in Dawson *et al.* [DHM+05], where the fact that paths over $\{H, \text{TOF}\}$ are balanced and have phase in $\{1, -1\}$ was used to reduce the amplitude function to a Boolean polynomial. Using this sum-over-paths action, computing the transition amplitudes of a quantum circuit reduces to computing the *gap* of a Boolean polynomial – the difference between the number of solutions to $P(\mathbf{x}) = 0$ and $P(\mathbf{x}) = 1$. As the problem of computing the gap of a Boolean polynomial is contained in **PP**, they gave a significantly simpler proof of the containment $\mathbf{BQP} \subseteq \mathbf{PP}$. This proof was later simplified further by Montanaro [Mon17], who showed that the phase function over the universal $\{H, Z, CZ, CCZ\}$ gate set can be described by a degree 3 Boolean polynomial.

3.2 Clifford+R_Z path integrals

We now consider circuits over a concrete gate set, Clifford+R_Z, and give a representation for their sum-over-paths action. Recall that we define the Clifford+R_Z gate set as $\{H, \text{CNOT}, X, R_Z(\theta) \mid \theta \in \mathbb{R}\}$. Each gate in this set can be defined by a sum-over-paths action as below:

$$\begin{aligned} H &: |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \mathbb{F}_2} e^{i\pi xy} |y\rangle \\ \text{CNOT} &: |x\rangle|y\rangle \mapsto |x\rangle|x \oplus y\rangle \\ X &: |x\rangle \mapsto |1 \oplus x\rangle \\ R_Z(\theta) &: |x\rangle \mapsto e^{i\theta x} |x\rangle \end{aligned}$$

Each of the above actions is balanced, with phase and output given by a pseudo-Boolean function and an affine function of the input and path variables, respectively. In particular, each above the above actions can be written as

$$|\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{if(\mathbf{x},\mathbf{y})} |A(\mathbf{x},\mathbf{y}) + \mathbf{b}\rangle,$$

which is completely specified by $m \in \mathbb{N}$, $f : \mathbb{F}_2^{n+m} \to \mathbb{R}$, and $A_{\mathbf{b}} \in \text{A}(\mathbb{F}_2^{n+m}, \mathbb{F}_2^n)$. Note that we denote the application of f and A to the concatenation of $\mathbf{x}$ and $\mathbf{y}$ by $f(\mathbf{x},\mathbf{y})$ and $A(\mathbf{x},\mathbf{y})$ respectively. Moreover, if $U : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^{m'}}} \sum_{\mathbf{y} \in \mathbb{F}_2^{m'}} e^{if'(\mathbf{x},\mathbf{y})} |A'(\mathbf{x},\mathbf{y}) + \mathbf{b}'\rangle$ by linearity it follows that that

$$\begin{aligned} U\left(\frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{if(\mathbf{x},\mathbf{y})} |A(\mathbf{x},\mathbf{y}) + \mathbf{b}\rangle\right) &= \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{if(\mathbf{x},\mathbf{y})} (U|A(\mathbf{x},\mathbf{y}) + \mathbf{b}\rangle) \\ &= \frac{1}{\sqrt{2^{m+m'}}} \sum_{\mathbf{y} \in \mathbb{F}_2^{m+m'}} e^{if''(\mathbf{x},\mathbf{y})} |A''(\mathbf{x},\mathbf{y}) + \mathbf{b}''\rangle \end{aligned}$$

where f'' and A'' are a pseudo-Boolean and an affine function of the input and $m + m'$ path variables, as both compose with affine Boolean transformations. In particular,

$$A'' = A'(A \bigoplus I_{m'}), \qquad \mathbf{b}'' = A'\mathbf{b} + \mathbf{b}',$$

$$f''(\mathbf{x},\mathbf{y},\mathbf{y}') = f(\mathbf{x},\mathbf{y}) + f'(A(\mathbf{x},\mathbf{y}) + \mathbf{b}, \mathbf{y}'),$$

where $\bigoplus$ denotes the direct sum, i.e.

$$A \bigoplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}.$$

The tensor product of two such path integrals is similarly expressible, and so it follows that any Clifford+R_Z circuit is expressible as a phase function and affine transformation over a set of path variables.

Proposition 3.2.1. *The action of an n-qubit Clifford+R_Z circuit C is expressible by a set of m path variables, phase function $f : \mathbb{F}_2^{n+m} \to \mathbb{R}$ and affine transformation $A_{\mathbf{b}} \in \mathrm{A}(\mathbb{F}_2^{n+m}, \mathbb{F}_2^n)$. In particular,*

$$[\![C]\!] : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{if(\mathbf{x},\mathbf{y})} |A(\mathbf{x}, \mathbf{y}) + \mathbf{b}\rangle$$

In general, a Clifford+R_Z circuit C may have more than representation of the form in Proposition 3.2.1 – in particular, the identity circuit HH is expressible as either

$$HH : |x\rangle \mapsto \frac{1}{2} \sum_{y_1, y_2 \in \mathbb{F}_2} (-1)^{xy_1 + y_1 y_2} |y_2\rangle \quad \text{or} \quad HH : |x\rangle \mapsto |x\rangle,$$

due to the interference of paths. When we refer to the sum-over-paths action of a circuit, we usually refer to the *canonical action*, corresponding to the complete composition of each individual gate's paths. We show below in Section 3.2.1 an intuitive, standard [DHM+05, Mon17] method of calculating the canonical action of a Clifford+R_Z circuit.

Affine gates Without the X gate, the canonical sum-over-paths action of an Clifford+R_Z circuit has an output state which is a strictly *linear* function of the input and path variables. We can nevertheless represent the X gate in this way, by noting that $X = HZH$ which has the canonical sum-over-paths action

$$HZH : |x\rangle \mapsto \frac{1}{2} \sum_{y_1, y_2 \in \mathbb{F}_2^n} e^{i\pi(xy_1 + y_1 + y_1 y_2)} |y_2\rangle.$$

As a result, we could represent path integrals over $\{H, \mathrm{CNOT}, X, R_Z\}$ using linear rather than affine transformations by directly defining the sum-over-paths action of X as

$$X : |x\rangle \mapsto \frac{1}{2} \sum_{y_1, y_2 \in \mathbb{F}_2^n} e^{i\pi(xy_1 + y_1 + y_1 y_2)} |y_2\rangle.$$

This demonstrates the fact that *there is no fixed path integral representation of a given unitary gate or circuit*, and so different representations may be more useful than others for certain circuit analyses. We use this representation as it allows us to model X gates *without using path variables*, which will be important for both optimization and verification.

The phase group While the phase function f for a circuit over Clifford+R_Z is described as a $\mathbb{R}$-valued function, it can be observed that if the angles of R_Z gates are restricted to some subgroup G or the additive group $\mathbb{R}$, then f is a pseudo-Boolean function with codomain contained in G. Moreover, there exists a natural equivalence of phase functions, as

$$e^{if(\mathbf{x},\mathbf{y})} = e^{if'(\mathbf{x},\mathbf{y})}$$

whenever $f(\mathbf{x},\mathbf{y}) = f'(\mathbf{x},\mathbf{y}) \mod 2\pi$ for all $\mathbf{x},\mathbf{y}$. In this case we say that $f \sim f'$.

It will be convenient at times to use a group isomorphic to either G or $G/2\pi\mathbb{Z}$ to represent the phase function. For instance, the Clifford+T gate set restricts the phase group to $\frac{\pi}{4}\mathbb{Z}$; further, since $\frac{\pi}{4}\mathbb{Z}/2\pi\mathbb{Z} \simeq \mathbb{Z}_8$, the phase function for a Clifford+T circuit is isomorphic to some $\mathbb{Z}_8$-valued function. Likewise, for any phase gate $Z_k = R_Z(2\pi/k)$ of order $k \geq 4$, we can represent the action of a Clifford+Z_k circuit with a $\mathbb{Z}_k$-valued phase function.

3.2.1 Annotated circuits

The canonical sum-over-paths action of a Clifford+R_Z circuit C can be easily calculated by constructing an *annotated circuit* [DHM+05, Mon17]. This method is particularly useful for calculations *by hand* and hence is useful for the presentation of results in this thesis. We discuss more explicitly computational methods later in Chapter 7.

We begin by labelling the n inputs of the circuit $x_1, x_2, \ldots, x_n$, and label each output of every gate with an affine combination of inputs and fresh path variables according to is sum-over-paths action. In particular, as the CNOT gate maps $|x\rangle|y\rangle$ to $|x\rangle|x \oplus y\rangle$, the output for the control bit of a CNOT gate has the same label as its input, while the target bit is labelled with the sum of the input labels. Likewise, the X gate adds an affine factor of 1 to its input label, and R_Z has output label the same as its input. The H gate introduces a new path variable y_i and places it along its output.

To construct the path integral from the annotated circuit, the phase contribution of each gate as a function of its input label is summed, and the output labels of the circuit

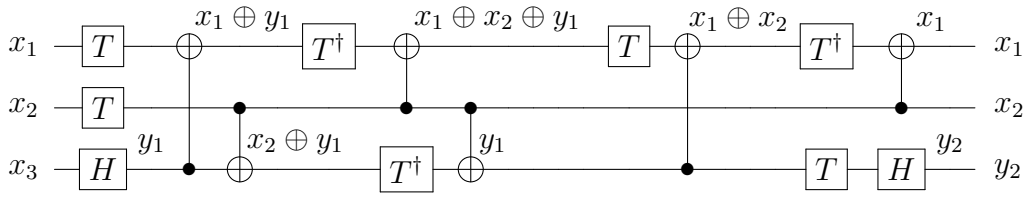


Figure 3.2: An annotated circuit implementing the Toffoli gate.

are taken as the output function. For a gate with phase function $f(\mathbf{x}, \mathbf{y}')$ and input label $A(\mathbf{x}, \mathbf{y}) + b$, the gate contributes a phase of $f(A(\mathbf{x}, \mathbf{y}) + b, \mathbf{y}')$.

As an example, Figure 3.2 shows a Clifford+T circuit implementing the Toffoli gate,

$$\text{TOF} : |x_1\rangle|x_2\rangle|x_3\rangle \mapsto |x_1\rangle|x_2\rangle|x_3 \oplus x_1x_2\rangle.$$

The state of each qubit as a linear combination of the inputs $x_1, x_2, x_3 \in \mathbb{F}_2$ and path variables $y_1, y_2 \in \mathbb{F}_2$ has been annotated after each gate. Annotations are only shown in cases where the state of the qubit is changed. Summing up the phase contributions due to $T, T^\dagger$, and H gates gives the phase function, written in mixed arithmetic as

$$f(\mathbf{x}, \mathbf{y}) = \frac{\pi}{4}\left(4x_3y_1 + x_1 + x_2 + 7(x_1 \oplus y_1) + 7(x_2 \oplus y_1) + (x_1 \oplus x_2 \oplus y_1) + 7(x_1 \oplus x_2) + y_1 + 4y_1y_2\right).$$

As the circuit returns the first two qubits to their initial states, and the third qubit to the value of y_2, we see that

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{b} = \mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

3.2.2 CNOT-dihedral circuits

A special case of path integrals occurs when no *branching* gates appear in a circuit – that is, gates which map a classical state to a superposition of classical states. In this case, the sum-over-paths action of the circuit maps a single input to a single output, corresponding uniquely to the composition of each gate's classical action and phase.

Circuits over the CNOT-dihedral gate set $\{\text{CNOT}, X, R_Z(\theta)\}$ belong to this special case. It can hence be observed that for such circuits, the output state is a pure affine function of the input basis state, and furthermore the phase is a pseudo-Boolean function of the input. Additionally, since quantum circuits are reversible, the mapping from input to output states is one-to-one and hence is given by an invertible affine function $A_\mathbf{b} \in \text{GA}(n, \mathbb{F}_2)$, where $\text{GA}(n, \mathbb{F}_2) \simeq \mathbb{F}_2^n \rtimes \text{GL}(n, \mathbb{F}_2)$ is the *general affine group* of invertible affine operators from $\mathbb{F}_2^n$ to $\mathbb{F}_2^n$. We sum this up in the following proposition.

Proposition 3.2.2. *The action of an n-qubit* CNOT*-dihedral circuit C is expressible by a phase function $f : \mathbb{F}_2^n \to \mathbb{R}$ and affine transformation $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$. In particular,*

$$[\![C]\!] : |\mathbf{x}\rangle \mapsto e^{if(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle.$$

Again, restricting to CNOT-phase circuits – i.e. circuits over $\{\mathrm{CNOT}, R_Z(\theta)\}$ – gives a simpler presentation in terms of general linear operators $A \in \mathrm{GL}(n, \mathbb{F}_2)$, but the addition of X gates allows more non-branching circuits to be represented.

A trivial property of the sum-over-paths action of CNOT-dihedral circuits which will be important for optimal synthesis later, is that the representation as a phase function $f : \mathbb{F}_2^n \to \mathbb{R}$ and invertible affine function $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$ is unique, up to equivalences mod 2π in the phase.

Proposition 3.2.3. *Given two* CNOT*-dihedral circuits C, C' with sum-over-paths actions $(f, A_{\mathbf{b}}), (f', A'_{\mathbf{b}'})$ respectively, then $[\![C]\!] = [\![C']\!]$ if and only if $A_{\mathbf{b}} = A'_{\mathbf{b}'}$, and $f = f' + g$ for some $g : \mathbb{F}_2^n \to 2\pi\mathbb{Z}$.*

Proof. A simple consequence of the fact that affine transformations in $\mathrm{GA}(n, \mathbb{F}_2)$ are invertible, hence for any $A_{\mathbf{b}}, A'_{\mathbf{b}'} \in \mathrm{GA}(n, \mathbb{F}_2)$, $A\mathbf{x} + \mathbf{b} = A'\mathbf{x} + \mathbf{b}'$ for all $\mathbf{x} \in \mathbb{F}_2^n$ if and only if $A = A'$ and $\mathbf{b} = \mathbf{b}'$. Then for any $\mathbf{x}$,

$$e^{if(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle = e^{if'(\mathbf{x})}|A'\mathbf{x} + \mathbf{b}'\rangle$$

if and only if $f(\mathbf{x}) = f'(\mathbf{x}) \mod 2\pi$. Hence $f = f' + g$ for some $g : \mathbb{F}_2^n \to 2\pi\mathbb{Z}$. □

As a trivial corollary, the set of n-qubit (infinite order) CNOT-dihedral circuits is isomorphic to

$$\mathbb{R}^{2^n}/2\pi\mathbb{Z}^{2^n} \times \mathrm{GA}(n, \mathbb{F}_2).$$

In Chapter 5 we further characterize the CNOT-dihedral circuits of *any finite order* in this way – i.e. as $\mathbb{R}^{2^n}/G \times \mathrm{GA}(n, \mathbb{F}_2)$ for some $G \triangleleft \mathbb{R}^{2^n}$.

3.3 Phase polynomials

In the preceding section, the phase function of a path integral was described simply as some pseudo-Boolean function. In this section, we discuss two concrete representations of the phase functions: as *multilinear polynomials* and their *Fourier expansions*.

It is a well known fact (see, e.g., [O'D14]) that any pseudo-Boolean function can be uniquely represented as a *multilinear polynomial*

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \mathbf{x}^{\mathbf{y}}$$

where $\mathbf{x}^{\mathbf{y}} = x_1^{y_1} x_2^{y_2} \cdots x_n^{y_n}$ is a multi-index, and by multilinear we mean each $y_i \in \mathbb{F}_2$. More generally, any function $f : \mathbb{F}_2^n \to G$ for some Abelian group G – i.e. any phase group – can also be uniquely represented in the above form.

There exists however a more direct relationship between the structure Clifford+R_Z circuit and the expression of the phase function as a sum of *parities* of the n variables. Denoting by $\chi_{\mathbf{y}} : \mathbb{F}_2^n \to \mathbb{F}_2$ the *parity function*

$$\chi_{\mathbf{y}}(\mathbf{x}) = \mathbf{y}^T \mathbf{x} = x_1 y_1 \oplus x_1 y_2 \oplus \cdots \oplus x_n y_n$$

for an indicator vector $\mathbf{y} \in \mathbb{F}_2^n$, it is known that any pseudo-Boolean function $f : \mathbb{F}_2^n \to \mathbb{R}$ has a unique presentation [O'D14] as

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \tilde{f}(\mathbf{y}) (-1)^{\chi_{\mathbf{y}}(\mathbf{x})},$$

called the *Fourier expansion* of f. The coefficients $\tilde{f}(\mathbf{y})$ are known as the *Fourier coefficients* of f, with the set of all 2^n coefficients collectively referred to as the *Fourier spectrum*.

It will be convenient to instead write an expansion directly as functions of parities, rather than their image in $\{1, -1\}$. In particular, it can be observed that for any $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$,

$$\tilde{f}(\mathbf{y})(-1)^{\chi_{\mathbf{y}}(\mathbf{x})} = \tilde{f}(\mathbf{y}) - 2\tilde{f}(\mathbf{y})\chi_{\mathbf{y}}(\mathbf{x}).$$

Writing $\widehat{f}(\mathbf{0}) = \tilde{f}(\mathbf{0}) + \sum_{\mathbf{y} \in \mathbb{F}_2^n \setminus \{\mathbf{0}\}} \tilde{f}(\mathbf{y})$ and $\widehat{f}(\mathbf{y}) = -2\tilde{f}(\mathbf{y})$ for $\mathbf{y} \neq \mathbf{0}$, we obtain a unique representation directly over the parity functions.

Proposition 3.3.1. *For any pseudo-Boolean function $f : \mathbb{F}_2^n \to \mathbb{R}$, there exists a unique expansion of f of the form*

$$f(\mathbf{x}) = \widehat{f}(\mathbf{0}) + \sum_{\mathbf{y} \in \mathbb{F}_2^n} \widehat{f}(\mathbf{y}) \chi_{\mathbf{y}}(\mathbf{x}).$$

Uniqueness follows from the fact that the mapping of coefficients $\tilde{f} \mapsto \widehat{f}$ is an isomorphism, with

$$\tilde{f}(\mathbf{0}) = \frac{2\widehat{f}(\mathbf{0}) + \sum_{\mathbf{y} \in \mathbb{F}_2^n \setminus \{\mathbf{0}\}} \widehat{f}(\mathbf{y})}{2}, \qquad \tilde{f}(\mathbf{y}) = -\frac{1}{2}\widehat{f}(\mathbf{y}) \text{ for } \mathbf{y} \neq \mathbf{0}.$$

Note that $\chi_{\mathbf{0}}(\mathbf{x}) = 0$ for all $\mathbf{x} \in \mathbb{F}_2^n$, hence the additional constant factor $\widehat{f}(\mathbf{0})$.

This expression of f, which we call a *phase polynomial*, has a deep connection to the structure of a Clifford+R_Z circuit with phase function f. Specifically, each phase gate in a Clifford+R_Z circuit contributes to exactly *one term* (up to constant factors) of the phase polynomial. Moreover, the phase function $f(x, y) = \pi xy$ of the Hadamard gate can be written as a phase polynomial with only three terms by noting that

$$\pi xy = \frac{\pi}{2}x + \frac{\pi}{2}y - \frac{\pi}{2}(x \oplus y).$$

Hence, if we define the *support* of $\widehat{f}$ as $\mathrm{supp}(\widehat{f}) = \{\mathbf{y} \in \mathbb{F}_2^{n+m} \mid \widehat{f}(\mathbf{y}) \neq 0\}$ and the cardinality of C to be the number of gates, we have the following important fact:

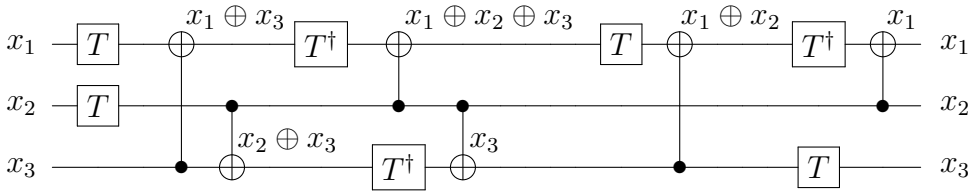
Proposition 3.3.2. *For any Clifford+R_Z circuit C, if C has (canonical) sum-over-paths action*

$$[\![C]\!] : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{if(\mathbf{x},\mathbf{y})} |A(\mathbf{x}, \mathbf{y}) + \mathbf{b}\rangle,$$

then $|\mathrm{supp}(\widehat{f})| \in O(|C|)$.

As the affine transformation $A_{\mathbf{b}}$ has size polynomial in $|C|$, it follows that *the sum-over-paths action of a Clifford+R_Z circuit C can be represented in space polynomial in $|C|$*.

Example 3.3.3. Recall the annotated circuit in Figure 3.2. Removing the Hadamard gates gives the CNOT-dihedral circuit



which has phase polynomial

$$f(\mathbf{x}) = \frac{\pi}{4}\left(x_1 + x_2 + 7(x_1 \oplus x_3) + 7(x_2 \oplus x_3) + (x_1 \oplus x_2 \oplus x_3) + 7(x_1 \oplus x_2) + x_3\right).$$

As each label is some parity $\chi_{\mathbf{y}}(\mathbf{x})$ of the input bits $\mathbf{x}$, the effect of a phase gate is to apply a phase rotation conditioned on the value of that parity. The phase polynomial representation hence directly represents the phase factors applied throughout a circuit.

While every pseudo-Boolean function into $\mathbb{R}$ has both a unique multilinear and Fourier-expanded representation, over $\mathbb{R}/2\pi\mathbb{Z}$ *only the multilinear presentation is unique*. This factors heavily into the optimization problems we consider in chapters 5 and 6.

Part II

Optimization

Chapter 4

Quantum Phase folding

Optimization is a crucial part of the quantum circuit design process, particularly to help best utilize the limited resources available and to accurately assess the crossover points for quantum algorithms. In this chapter, we introduce an efficient circuit optimization heuristic for reducing the number of phase gates

$$R_Z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

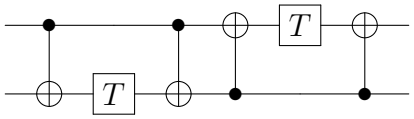
in a quantum circuit, which we call *quantum phase-folding*. Recall that for phase gates outside of the set $\{Z, S, S^\dagger\}$, they are typically the most expensive gates to implement fault-tolerantly, and hence it is highly desirable for fault-tolerant quantum computing to reduce the number of such gates.

Our algorithm uses the intuition developed in Chapter 3 to develop an optimization algorithm which tracks only which phase gates apply to the same set of computational paths, corresponding to the same terms of the phase polynomial, and hence can be merged. As only which paths a phase has been applied to matters, the algorithm applies to gates with arbitrary – even indeterminate – phases. By using abstraction, we further extend applicability to circuits which contain *uninterpreted gates* – gates whose interpretation as a unitary matrix is not known. Since only phase gates are merged, the algorithm is effectively non-increasing in all other metrics, including depth and gate counts, hence making it suitable as a standard quantum compiler optimization. To allow additional more targeted optimizations, a simple modification to the algorithm further breaks the circuit into (overlapping) CNOT-dihedral sub-circuits, which can then be optimally or sub-optimally synthesized; we explore this problem in the following two chapters.

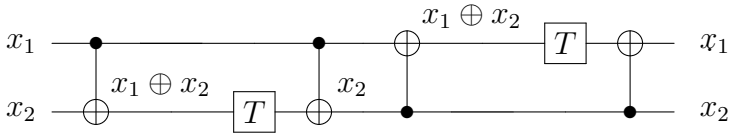
This chapter builds upon work which originally appeared in [AMM14], in particular by separating T-count and T-depth optimization into phase-folding and CNOT-dihedral synthesis, respectively, and extending the algorithm to uninterpreted gates. A proof of correctness is also given and the overall run-time complexity is reduced.

4.1 Overview

Consider the following circuit:



We can calculate the circuit's sum-over-paths action by first annotating the circuit, as in the last chapter:

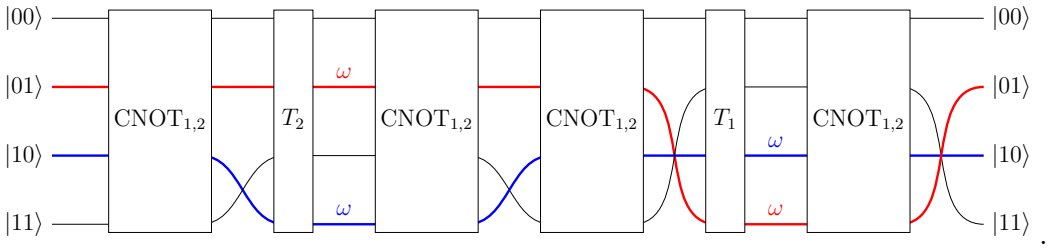


Collecting the phase terms due to the $T = R_Z(2\pi/8)$ gates, we see that the circuit has the phase polynomial

$$f(x_1, x_2) = \frac{4\pi}{8}(x_1 \oplus x_2) = \frac{\pi}{2}(x_1 \oplus x_2).$$

As a result, while the circuit contains two phase gates, *only a single combined phase rotation is performed* with angle $\pi/2$, conditioned on the value of $x_1 \oplus x_2$.

Rather than applying T gates twice to the state $|x_1 \oplus x_2\rangle$ at different points in the circuit, a single $T^2 = S$ gate could instead be applied. As the input to each T gate is $|x_1 \oplus x_2\rangle$, this can be done simply by replacing one T gate with an S gate and removing the other. If we examine the evolution of each initial state through the application of each gate, this becomes particularly clear:

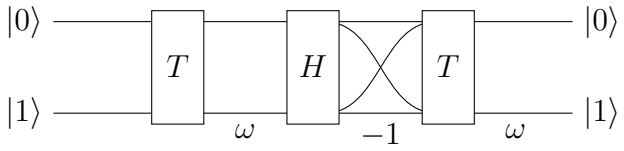


Note that $\omega = e^{i\pi/4}$. Here the red and blue paths – corresponding to the solutions of $x_1 \oplus x_2 = 1$ – each acquire a total phase of $\omega^2 = i$, which moreover can be placed at any point along either path.

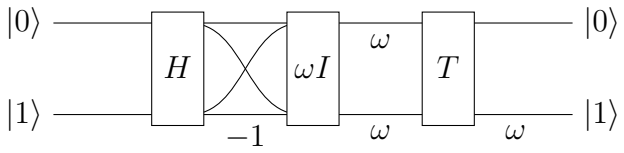
This basic intuition of *merging phases along the same path* can be readily applied to CNOT-dihedral circuits by simply computing the phase polynomial and applying one total phase rotation of angle $\widehat{f}(\mathbf{y})$ for each $\mathbf{y} \in \text{supp}(\widehat{f})$. The rest of this section deals with extending this intuition to universal gate sets including *branching* gates or otherwise arbitrary gates.

4.1.1 Branching gates

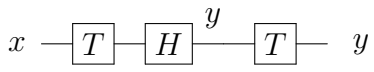
Most useful quantum circuits contain branching gates, such as the Hadamard gate H, which complicates the process of merging phases along a path. In particular, consider the evolution of paths through the circuit THT:



In this case there is no way to "slide" the phases along paths so that the total number of phase gates are reduced. In particular, commuting the first ω phase past the Hadamard on the right copies the ω onto both output paths, corresponding to a single ωI gate, e.g.,



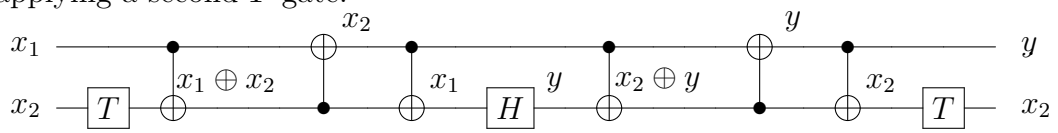
However, this adds an extra phase of ω to both paths starting from the $|0\rangle$ state. The extra phase could be cancelled by applying a ω^{-1} phase on the $|0\rangle$ input state (e.g., with $XT^\dagger X$), but the result is *more* phase gates than the circuit initially had! From an algebraic point-of-view, recall that we label the output of a Hadamard gate with a *path variable*:



Collecting the phase contribution of each gate we see that the the total phase is $\omega^{x+y}(-1)^{xy} = \omega^{x+y+4xy}$, hence no phases are merged.

On the other hand, if phase gates are applied to qubits which are not directly affected by a branching gate, it is expected that such phase gates should be possible to merge.

Fortunately, this is easy to observe in the phase polynomial formalism. For instance, consider the circuit below which applies a T gate to the second qubit before logically swapping it with the first qubit, applying a Hadamard on the second, then swapping back and applying a second T gate:

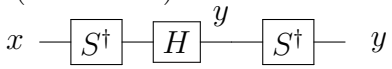


Again the total phase is $\omega^{2x_2+4x_1y}$, hence the T gates – having input $|x_2\rangle$ – can be merged, with a total angle of $\pi/2$.

4.1.2 Abstraction

The intuition of the phase-folding algorithm so far is that phases gates which contribute to the same term in the phase polynomial can be merged. However, non-phase gates – for instance, the Hadamard gate – may also apply phase rotations which contribute to the same term as a phase gate, but can't be merged.

Consider, for example, the (annotated) circuit below:



As the Hadamard gate applies a phase of ω^{4xy} and moreover

$$\omega^{4xy} = \omega^{2x+2y-2(x\oplus y)},$$

the total phase of the circuit is thus $\omega^{-2(x\oplus y)}$, cancelling the two phases of ω^{-2x} and ω^{-2y} from the $S^\dagger$ gates. However, it is not possible (over the Clifford+R_Z gate set) to directly implement the transformation $|x\rangle \mapsto \frac{1}{\sqrt{2}}\sum_{y\in\mathbb{F}_2}\omega^{-2(x\oplus y)}|y\rangle$ – the only way to implement the desired branching is to use a Hadamard gate, which necessarily applies a phase of $\omega^{2x+2y-2(x\oplus y)}$ and hence must be corrected. In effect, the phase contribution of the Hadamard gate is fixed and cannot be merged with any other phase gates over the Clifford+R_Z set.

A natural way to avoid such erroneous phase cancellations is to *only track terms of the phase polynomial which can be merged*. We can do this by abstracting the phase polynomial f to some $f^\sharp$ comprised of only the phase contributions due to R_Z gates. For instance, the above circuit has the real phase polynomial

$$f(x,y) = \frac{-2\pi}{8}x + \frac{4\pi}{8}xy + \frac{-2\pi}{8}y = \frac{-2\pi}{8}(x \oplus y)$$

but abstract phase polynomial

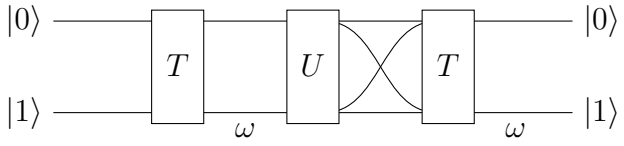
$$f^\sharp(x,y) = \frac{-2\pi}{8}x + \frac{-2\pi}{8}y.$$

Uninterpreted gates We can further use abstraction to soundly approximate the effect of gates U *whose particular semantics are unknown* – i.e. uninterpreted gates. In particular, recall that any unitary transformation can be written as a sum-over-paths:

$$U : |\mathbf{x}\rangle \mapsto \sum_{\mathbf{y} \in \mathbb{F}_2^n} \langle \mathbf{y}|U|\mathbf{x}\rangle |\mathbf{y}\rangle.$$

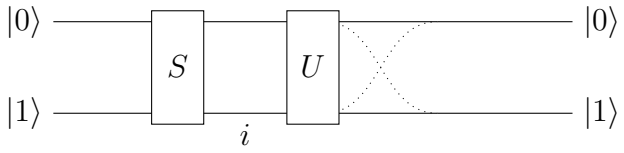
The above expression indicates that a unitary transformation U "takes" every possible branch – however, depending on the semantics some branches may have amplitude $\langle \mathbf{y}|U|\mathbf{x}\rangle = 0$, and hence are *infeasible*.

If we know certain branches from a gate are infeasible, we can commute phases across the gate. For example, consider the circuit TUT below, for some unitary U:



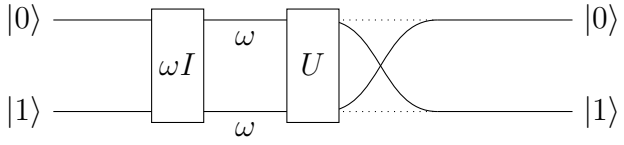
.

If $\langle 1|U|0\rangle = 0 = \langle 0|U|1\rangle$, then we can combine the ω phases as below, since the extra phase on the infeasible $|1\rangle \mapsto |0\rangle$ path doesn't change the computation.



.

Likewise, if instead $\langle 0|U|0\rangle = 0 = \langle 1|U|1\rangle$, we see the circuit is equivalent to $U(\omega I)$:



.

On the other hand, if all branches have non-zero amplitude the phases, as is the case for the Hadamard gate, no useful phase commutations are possible.

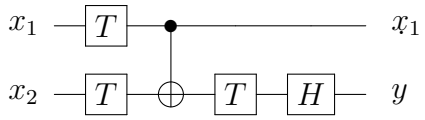
When we encounter some n-qubit uninterpreted gate U we *over-approximate* its feasible branches by assuming that every branch (of qubits on which it acts) may be taken. Practically speaking, the result is that phase gates cannot be commuted through an uninterpreted gate, but can in fact be commuted around them, regardless of their semantics.

Path representatives The final element to our phase-folding algorithm is a further abstraction to reduce the number of variables the phase polynomial is represented over.

Given an n-qubit circuit C with m Hadamard gates, recall that the sum-over-paths action is specified over n input and m path variables – that is, the phase polynomial and the n output states are functions of $n + m$ variables.

We can instead view the n output states themselves as variables $\mathbf{x}'$, subject to a set of (linear) equalities between $\mathbf{x}, \mathbf{y}$ and $\mathbf{x}'$. For instance, the CNOT gate has input $|\mathbf{x}\rangle$ and output $|\mathbf{x}'\rangle$, where $x'_1 = x_1$ and $x'_2 = x_1 \oplus x_2$. We then say that a parity of $\mathbf{x}, \mathbf{y}$, and $\mathbf{x}'$ has a *representative over some subset* $S \subseteq \{\mathbf{x}, \mathbf{y}, \mathbf{x}'\}$ if there exists an equivalent parity over just variables in S.

Example 4.1.1. Consider the circuit



The full phase expression (excluding the Hadamard phase contribution) is $\omega^{x_1+x_2+(x_1\oplus x_2)}$. If we label the outputs $x'_1 = x_1$ and $x'_2 = y$, as a function of $\mathbf{x}'$, $\omega^{x_1} = \omega^{x'_1}$, but x_2 and $x_1 \oplus x_2$ don't have representatives over x'_1 and x'_2 – e.g., $x_2 \neq ax'_1 + bx'_2$ for any $a, b \in \mathbb{F}_2$.

It can be observed that if a particular parity does not have a representative over the "primed" output variables, no subsequent phase gates can apply a rotation conditional on the same parity – in particular, any phase conditional on that linear combination *cannot be merged with subsequent gates*. Intuitively, we can then forget about the *particular* parity and just record the fact that a phase rotation was applied to *some* parity. For instance, in the above example we could (informally) rewrite the phase expression as

$$\exists x_1, x_2 \in \mathbb{F}_2 . \omega^{x'_1+x_2+(x_1\oplus x_2)}$$

which is equivalent as well to $\exists x_1, x_2 \in \mathbb{F}_2 . \omega^{x'_1+x_1+x_2}$. Effectively, we want to existentially quantify the input and path variables, leaving only the n primed outputs.

The practical advantage is that using representatives over just the output states reduces the number of terms in any particular parity to n, as opposed to the naïve $n + m$. As in most realistic circuits $m \gg n$, this can substantially reduce the space requirements of phase-folding and leverage faster non-sparse representations such as bitvector arithmetic. On the other hand, extra effort is spent calculating representatives. The full algorithm, described in the next section, takes a hybrid approach where representatives over a set of n variables spanning the output state are used. As a result, new representatives are only needed when the set of variables changes due to an uninterpreted gate.

4.2 The phase-folding algorithm

We now describe the phase-folding algorithm in full. Our algorithm first performs a *phase analysis*, interpreting a circuit C over X, CNOT, R_Z, and uninterpreted gates U as a state transformer over a domain of (abstract) path integrals, analogous to classical compiler optimizations; the algorithm then uses the phase polynomial to merge phase gates. For convenience we assume each phase gate has been labelled with a unique identifier $\ell \in \mathcal{L}$ where $\mathcal{L} \subseteq \mathbb{N}$, and we write a labelled phase gate as $R_Z(\theta)^\ell$.

We additionally assume that no $\dagger$'s appear in the circuit. Note that any circuit can be normalized to have all inverses at the gate level according to the group laws, and then absorbed into gates with the identities $X^\dagger = X$, $\mathrm{CNOT}^\dagger = \mathrm{CNOT}$, $R_Z(\theta)^\dagger = R_Z(-\theta)$, and $U^\dagger = U'$ for some uninterpreted gate U'. Finally, to simplify the presentation of uninterpreted gates, without loss of generality we assume all uninterpreted gates act on the last k qubits.

4.2.1 Phase analysis

Let $\mathcal{T} = \mathcal{P}(\mathcal{L} \times \{1, -1\})$ be the powerset of *phase terms*, each with a *polarity* in $\{1, -1\}$ denoting the affine subspace at that location in the circuit – that is, whether the qubit state is $\chi_{\mathbf{y}}(\mathbf{x})$ or $1 \oplus \chi_{\mathbf{y}}(\mathbf{x})$, respectively, hence the rotation is $e^{i\theta_\ell \chi_{\mathbf{y}}(\mathbf{x})}$ or

$$e^{i\theta_\ell(1 \oplus \chi_{\mathbf{y}}(\mathbf{x}))} = e^{i(\theta_\ell - \theta_\ell \chi_{\mathbf{y}}(\mathbf{x}))}.$$

We define $\mathcal{F}_n^\sharp = \mathcal{P}(\mathcal{T} \times (\mathbb{F}_2^n \cup \{\bot\}))$ to be the set of abstract phase polynomials on n variables, represented as a set of phases each with an optional representative n-bit parity. We assume that for any $f^\sharp \in \mathcal{F}_n^\sharp$ and $\mathbf{x} \in \mathbb{F}_2^n$, there is at most one pair $(T, \mathbf{x}) \in f^\sharp$, and so we use the functional notation $[r \to T]$ to denote the set $\{(T, r)\}$. We define a union operator $\uplus$ on $\mathcal{F}_n^\sharp \times \mathcal{F}_n^\sharp$ by taking the union of phases with the same representative:

$$f^\sharp \uplus g^\sharp = \bigcup_{\substack{(T,r) \in f^\sharp \\ (T',r') \in g^\sharp}} \begin{cases} \{(T \cup T', r)\} & \text{if } r = r' \text{ and } r \in \mathbb{F}_2^n \\ \{(T, r), (T', r')\} & \text{otherwise} \end{cases}$$

Recall that $\mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ is the set of affine transformations from $\mathbb{F}_2^n$ to $\mathbb{F}_2^n$, and that a linear (or affine) operator B acts on an affine operator $A_{\mathbf{b}}$ as $B(A_{\mathbf{b}}) = (BA)_{(B\mathbf{b})}$. For convenience we write A_i to denote the ith row of a matrix A, $e_i \in \mathbb{F}_2^n$ to denote the ith elementary vector and $E_{i,j} \in \mathrm{L}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ the elementary matrix adding row i to row j.

We now define the domain of our analysis as $\mathcal{A}_n = \mathcal{F}_n^\sharp \times \mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ – i.e. path integrals represented as pairs of phase polynomials and output functions. The analysis proceeds by executing an n-qubit circuit C with the abstract semantics $[\![C]\!]_a : \mathcal{A}_n \to \mathcal{A}_n$ and a suitable input state. The definition of $[\![\cdot]\!]_a$ is given below:

$$
\begin{aligned}
[\![X_i]\!]_a(f^\sharp, A_{\mathbf{b}}) &= (f^\sharp, A_{\mathbf{b} \oplus e_i}) \\
[\![\mathrm{CNOT}_{i,j}]\!]_a(f^\sharp, A_{\mathbf{b}}) &= (f^\sharp, E_{i,j} A_{\mathbf{b}}) \\
[\![R_Z(\theta)_i^\ell]\!]_a(f^\sharp, A_{\mathbf{b}}) &= (f^\sharp \uplus [A_i^T \to \{(\ell, (-1)^{b_i})\}], A_{\mathbf{b}}) \\
[\![U_{n-k+1,\ldots,n}]\!]_a(f^\sharp, A_{\mathbf{b}}) &= (g^\sharp, (A_k A_k^g)_{\mathbf{b}_k}) \text{ where} \\
A_k &= \begin{bmatrix} A_{1,\ldots,n-k} & 0 \\ 0 & I_k \end{bmatrix} \in \mathrm{L}(\mathbb{F}_2^{n+k}, \mathbb{F}_2^n), \qquad \mathbf{b}_k = \begin{bmatrix} b_{1,\ldots,n-k} \\ 0 \end{bmatrix} \in \mathbb{F}_2^n \\
g^\sharp &= \biguplus_{(T,r) \in f^\sharp} \begin{cases} [(A_k^g)^T(\mathbf{y}, \mathbf{0}) \to T] & \text{if } r = \mathbf{y} \text{ and } (A_k^g A_k)^T(\mathbf{y}, \mathbf{0}) = (\mathbf{y}, \mathbf{0}) \\ [\bot \to T] & \text{otherwise} \end{cases} \\
[\![C \fatsemi C']\!]_a(f^\sharp, A_{\mathbf{b}}) &= [\![C']\!]_a \circ [\![C]\!]_a(f^\sharp, A_{\mathbf{b}})
\end{aligned}
$$

Note that $A_{1,\ldots,n-k}$ and $b_{1,\ldots,n-k}$ above denote the first $n-k$ rows of A and $\mathbf{b}$ respectively, and A_k^g is a *generalized inverse* [CM09] of A_k. In particular, a generalized inverse A^g of a matrix $A \in \mathrm{L}(\mathbb{F}_2^m, \mathbb{F}_2^n)$ is an n by m matrix over $\mathbb{F}_2$ such that

$$AA^gA = A,$$

and in particular $AA^g\mathbf{y} = \mathbf{y}$ whenever there exists $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{y}$. We use a generic generalized inverse rather than the Moore-Penrose pseudoinverse, as the latter typically does not exist over finite fields. By contrast, for $A \in \mathrm{L}(\mathbb{F}_2^m, \mathbb{F}_2^n)$, a generalized inverse A^g may be computed [CM09] by finding invertible matrices P, Q over $\mathbb{F}_2$ such that

$$A = P \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} Q$$

and setting

$$A^g = Q^{-1} \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} P^{-1}.$$

Example 4.2.1. Consider the 3-qubit state $(f^\sharp, A_{\mathbf{0}})$ where

$$f^\sharp = [101 \to T_1, 001 \to T_2], \qquad A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

We can interpret this as representing a circuit with sum-over-paths action

$$|\mathbf{x}\rangle \mapsto \sum_{\mathbf{x}' \in \mathbb{F}_2^3} \phi(\mathbf{x}')|(x'_1 \oplus x'_2)(x'_1 \oplus x'_3)x'_3\rangle$$

and applying phase rotations at locations in T_1 and T_2 on the states $|x_1 \oplus x_3\rangle$ and $|x_3\rangle$, respectively.

We can see the effect of next applying a single-qubit uninterpreted gate U_3 by calculating a generalized inverse of A_k:

$$A_k = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad A_k^g = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad A_k A_k^g = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The matrix $A_k A_k^g$ effectively projects a vector onto the row space of A_k – that is, onto the basis

$$\{x'_1 \oplus x'_2, x'_1 \oplus x'_3, y\},$$

where y is the path variable corresponding to the output of U. Intuitively, the effect is a dimension reduction, rewriting the sum-over-paths as

$$\sum_{\mathbf{x}' \in \mathbb{F}_2^3} \sum_{\mathbf{y} \in \mathbb{F}_2} \phi(\mathbf{x}', \mathbf{y})|(x'_1 \oplus x'_2)(x'_1 \oplus x'_3)y_1\rangle = \sum_{\mathbf{x}'' \in \mathbb{F}_2^3} \phi'(\mathbf{x}'')|\mathbf{x}''\rangle.$$

Moreover, we see that the phase term corresponding to T_1 has a representation as a parity of $\{x''_1, x''_2, x''_3\}$, since $x'_1 \oplus x'_3 = x''_2$:

$$(A_k^g A_k)^T \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = A_k^T \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}.$$

On the other hand, since $x'_3 \notin \text{span}\{x'_1 \oplus x'_2, x'_1 \oplus x'_3, y\}$, T_2 does not have a representative which we can verify by seeing that

$$(A_k^g A_k)^T \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = A_k^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

4.2.2 Phase-folding

Algorithm 4.1 gives the phase folding algorithm. The algorithm takes as input a circuit C and an initial state $A_{\mathbf{b}} \in \mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^n)$ giving the intended input space of the circuit (i.e., which qubits are initialized), and returns a circuit with merged phase gates.

Example 4.2.2. For a circuit over 2 input qubits and 1 ancilla initialized in the $|1\rangle$ state,

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Algorithm 4.1 Quantum phase-folding

```
function PHASE-FOLD(C, A_b)
    (f♯, A'_b') ← [[C]]_a(∅, A_b)
    for (T, r) ∈ f♯ do
        Select (ℓ, a) ∈ T
        θ_ℓ ← Σ_{(ℓ',a')∈T} a' · θ_ℓ'
        For all remaining (ℓ', a') ∈ T, θ_ℓ' ← 0
    end for
end function
```

The algorithm, in analogy to classical data flow analysis, runs the circuit C with the abstract semantics $[\![\cdot]\!]_a$ to determine sets of phase gates which may be merged (terms $T \in \mathcal{T}$). For each such set of phase gate locations, we then choose a single phase gate location ℓ which will apply the total rotation of $\sum_{(\ell',a')\in T} a' \cdot \theta_{\ell'}$, where $\theta_{\ell'}$ is the argument to the phase gate ℓ'. The multiplicative factor of a adjusts (up to global phase) for whether the rotation is applied to x or $1 \oplus x$, as

$$e^{i\theta x} = e^{i\theta} e^{-i\theta(1\oplus x)}.$$

While the algorithm as described only preserves unitary equivalence up to global phase, it is an easy adjustment to preserve strict equivalence by tracking the global phase and applying a correction at the end, noting that $e^{i\theta} I = R_Z(\theta) X R_Z(\theta) X$.

Theorem 4.2.3. *Algorithm 4.1 takes time polynomial in the volume (qubits × gates) of the input circuit and is strictly non-increasing on all gate counts.*

Proof. A consequence of fact that the state $(f^\sharp, A_{\mathbf{b}})$ is only ever at most quadratic in the volume of the circuit, and moreover no gates are added to the circuit. □

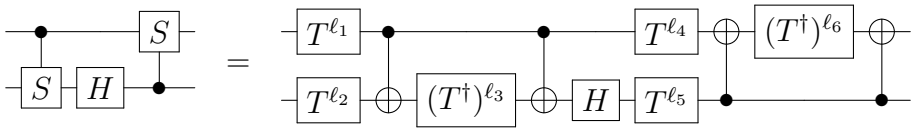
A proof of correctness of Algorithm 4.1 is given in Chapter A.

4.2.3 Examples

To make the phase-folding algorithm more concrete, we now give some examples highlighting various features.

Example 4.2.4 (Phase folding)**.** We first give a simple example showing the merging of phase gates across uninterpreted gates. Note that a Hadamard gate is treated identically to an uninterpreted gate.

Recall that the controlled-S gate can be implemented as $(T \otimes T)$ ⨟ CNOT ⨟ $(I \otimes T^\dagger)$ ⨟ CNOT and consider the following circuit:



After the first controlled-S gate, the state of the analysis is

$$([10 \to \{(\ell_1, 1)\}, 01 \to \{(\ell_2, 1)\}, 11 \to \{(\ell_3, 1)\}], I_{\mathbf{0}})\,.$$

After the Hadamard gate, the second qubit is in the state y, hence parities x_2 and $x_1 \oplus x_2$ corresponding to 01 and 11 above no longer have representatives. The resulting state is

$$([10 \to \{(\ell_1, 1)\}, \bot \to \{(\ell_2, 1)\}, \bot \to \{(\ell_3, 1)\}], I_{\mathbf{0}})\,.$$

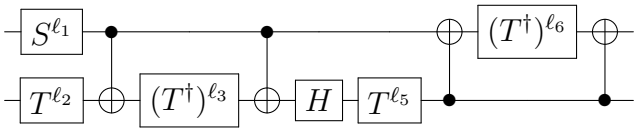
Applying the remaining gates, corresponding to the second controlled-S, results in the set of phase terms

$$[10 \to \{(\ell_1, \ell_4, 1)\}, \bot \to \{(\ell_2, 1)\}, \bot \to \{(\ell_3, 1)\}, 01 \to \{(\ell_5, 1)\}, 11 \to \{(\ell_6, 1)\}].$$

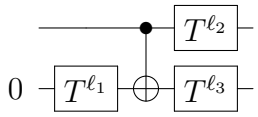
By contrast, the circuit has the sum-over-paths action

$$|x_1 x_2\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \mathbb{F}_2} \omega^{2x_1 + x_2 - (x_1 \oplus x_2) + y - (x_1 \oplus y) + 4x_2 y} |x_1 y\rangle.$$

It can be observed that each term in the final set of phase terms corresponds exactly to one term of the actual phase polynomial, with the $2x_1$ factor representing the fact that phase gates ℓ_1 and ℓ_4 rotate on the same parity (x_1), applying a single total rotation of ω^2. Selecting ℓ_1 for this total rotation gives a final circuit of



Example 4.2.5 (Ancillas)**.** Now we consider the affect of qubit initialization on phase-folding. In particular, the following circuit begins with the second qubit initialized in the $|0\rangle$ state and implements the transformation $|x\rangle|0\rangle \mapsto i^x|x\rangle|x\rangle$:



Starting with the initial state

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \qquad \mathbf{b} = \mathbf{0},$$

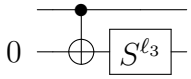
the phase folding algorithm computes

$$[\![C]\!]_a(\emptyset, A_{\mathbf{b}}) = ([00 \to \{(\ell_1, 1)\}, 10 \to \{(\ell_2, 1), (\ell_3, 1)\}], A'_{\mathbf{b}'}),$$

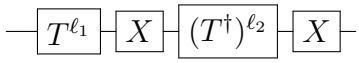
where

$$A' = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \qquad \mathbf{b}' = \mathbf{0}.$$

If for the second phase term, ℓ_3 is chosen, the resulting circuit after phase-folding is



Example 4.2.6 (Global phase)**.** Consider the circuit



The circuit implements the unitary $|x\rangle \mapsto \omega^{7+2x}|x\rangle$, corresponding to a global phase of ω and a relative phase of ω^{-2x}.

To see the effect of phase-folding on the above circuit, we can first observe that

$$[\![T^{\ell_1} \fatsemi X]\!]_a(\emptyset, I_{\mathbf{0}}) = ([1 \to \{(\ell_1, 1)\}], I_{\mathbf{1}}).$$

Since $b_1 = 1$, the $T^\dagger$ gate applies a rotation of $\omega^{-(1 \oplus x)}$ rather than ω^{-x}, and hence the result of the phase analysis is

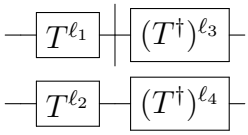
$$f^\sharp = [1 \to \{(\ell_1, 1), (\ell_2, -1)\}]$$

Computing the total phase applied to the state $|x\rangle$ we get $\theta_{\ell_1} - \theta_{\ell_2} = \pi/2$. Choosing either ℓ_1 or ℓ_2 to apply the rotation we get the circuits on the left and right below, implementing

the transformations $|x\rangle \mapsto \omega^{2x}|x\rangle$ and $|x\rangle \mapsto \omega^{6+2x}|x\rangle$ which differ from the original circuit by a global phase of ω^{-1} and ω, respectively.

In either case, the global phase can be corrected by adding a $\omega I = (HS)^3$ or $(\omega I)^\dagger$ gate to the end of the circuit.

Example 4.2.7 (Optimization barriers)**.** The *OpenQASM* language [CBSG17] includes a *barrier* instruction which semantically has no effect, but blocks optimizations from crossing the boundary it imposes. For instance, the T and $T^\dagger$ gates below on either side of the vertical barrier line cannot be cancelled, while the T and $T^\dagger$ gates not separated by the barrier can be cancelled:



Such barriers are useful particularly in circuits for error correction, where the structure of the circuit is often important to preserve error propagation behaviours.

We can precisely emulate the barrier instruction in the phase-folding algorithm by simply treating the barrier as an uninterpreted gate:

$$[\![C]\!]_a(\emptyset, I_{\mathbf{0}}) = ([\bot \to \{(\ell_1, 1)\}, 10 \to \{(\ell_3, 1)\}, 01 \to \{(\ell_2, 1), (\ell_4, 1)\}]), I_{\mathbf{0}}).$$

Hence, even though semantically, the barrier gate has no effect, the analysis concludes that T^{ℓ_1} and $(T^\dagger)^{\ell_3}$ are not merge-able, while T^{ℓ_2} and $(T^\dagger)^{\ell_4}$ are still merge-able as desired.

Example 4.2.8 (Parametrized circuits)**.** The phase-folding algorithm naturally applies to *parametrized* circuits which frequently appear in QRAM-style quantum programs and are integrals to algorithms such as the *variational quantum eigensolver* (VQE). We can model such circuits by extending phase angles to real-valued *terms* $\theta \in \mathbb{R}[v_1, \ldots, v_k]$ in some parameters $v_1, \ldots, v_k$. Then,

$$\theta(v_1, \ldots, v_k) + \theta'(v_1, \ldots, v_k) = (\theta + \theta')(v_1, \ldots, v_k)$$

and hence $R_Z(\theta)R_Z(\theta') = R_Z(\theta + \theta')$ for all values of the parameters.

This opens up phase-folding optimization to any quantum circuit description language with real-valued variables and expressions. For instance, the following QCL [Ö03] code defines a parametrized circuit (`operator`).

```
const pi = 3.14159265358979323846264338327950288419 7;
operator foo(real theta, qureg q) {
        RotZ(theta, q[0]);
        RotZ(pi/2, q[0]);
}
```

By interpreting the $\sum$ in line 5 of Algorithm 4.1 as the `+` expression of QCL, with phase-folding we can optimize the body of `foo` to the equivalent program:

```
const pi = 3.14159265358979323846264338327950288419 7;
operator foo(real theta, qureg q) {
        RotZ(theta + pi/2, q[0]);
}
```

4.3 CNOT-dihedral resynthesis

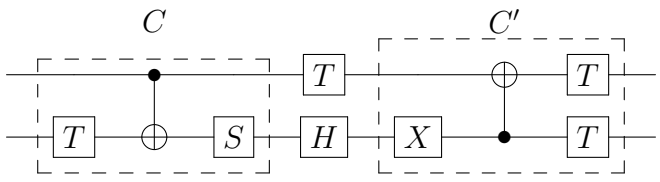
We now discuss a modification of the phase-folding algorithm to allow further optimization by the resynthesis of CNOT-dihedral subcircuits. Recall that an n-qubit CNOT-dihedral circuit C implements some affine permutation $(A, \mathbf{b}) \in \mathrm{GA}(n, \mathbb{F}_2)$ of an input $|\mathbf{x}\rangle$, together with a phase function $f : \mathbb{F}_2^n \to \mathbb{R}$ – i.e.

$$[\![C]\!] : |\mathbf{x}\rangle \mapsto e^{if(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle.$$

Moreover, as $\mathrm{supp}(\widehat{f})$ has size linear in the size of C, CNOT-dihedral circuits are a natural candidate for optimal synthesis.

Since the cost and complexity of CNOT-dihedral synthesis depends largely on the number of terms in the phase polynomial, we naturally wish to apply phase-folding to first reduce the complexity of the phase polynomials, before re-synthesizing CNOT-dihedral subcircuits. In general however, there may be multiple ways to divide a circuit into CNOT-dihedral chunks, which can affect the cost of the individual synthesis problems.

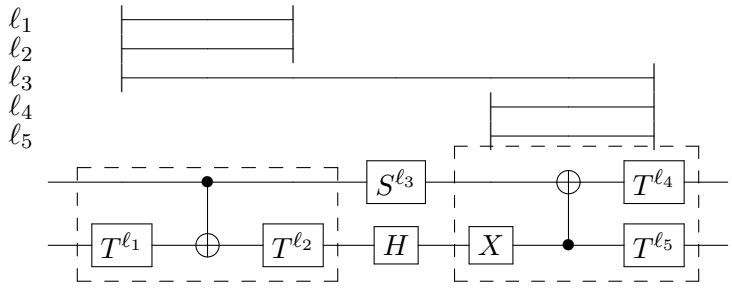
For instance, the T gate below can be associated with either the left or the right CNOT-dihedral chunk C or C', respectively:



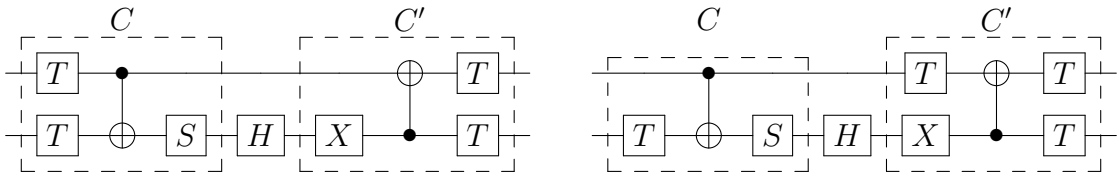
Noting that

$$[\![C]\!] : |x_1 x_2\rangle \mapsto e^{\frac{i\pi}{4}(x_2 + 2(x_1 \oplus x_2))} |x_1\rangle |x_1 \oplus x_2\rangle$$
$$[\![C']\!] : |x_1 x_2\rangle \mapsto e^{\frac{i\pi}{4}(2 + 7x_2 + 7(x_1 \oplus x_2))} |1 \oplus x_1 \oplus x_2\rangle |1 \oplus x_2\rangle,$$

associating the T gate with either subcircuit adds a term of $\frac{\pi}{4}x_1$ to the phase polynomial in either case. We say that the term $\frac{\pi}{4}x_1$ can range between C and C'. Labelling the phase gates $\ell_1, \ldots, \ell_5$ we can visualize the ranges for all terms:



In this case, associating the term $\frac{\pi}{4}x_1$ with the subcircuit C allows the total circuit to be implemented with T-depth 2, while associating it with C' gives a total T-depth of 3, as shown on the left and right below, respectively.



The algorithm we now describe combines phase-folding with an additional backwards pass to determine the range of each phase term. A particular synthesis algorithm may then freely associate terms with particular CNOT-dihedral subcircuits in their range.

4.3.1 Phase range analysis

Informally, the algorithm works by recording the point in a circuit at which a given term no longer has a representative parity over the current state, and hence is no longer computable with just CNOT gates. By applying phase-folding both forwards and backwards, an interval or range for each term is obtained. To simplify the presentation, we describe only the range analysis – that is, we don't account for phase gate merging in this analysis. For practical implementations, phase folding can be easily combined with the initial forward analysis.

We assume that every gate in a circuit C is labelled with a unique identifier $\ell \in \mathcal{L}$, and that if U^{ℓ} occurs before $U'^{\ell'}$ in C, then $\ell < \ell'$. We let $\mathcal{F}_n = \mathbb{F}_2^n \times \mathcal{L}$ denote the set of phase polynomials on n-qubits and as before describe basis states by some $A_{\mathbf{b}} \in \mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^n)$. We denote by $\iota \in \mathcal{R}_n^{\mathcal{L}} = \mathcal{P}(\mathcal{L})^{\mathcal{L}}$ mappings from locations (of phase gates) to subsets of circuit locations called *ranges*. We denote by $[\ell, \ell']$ the range $\{\ell'' \in \mathcal{L} \mid \ell \leq \ell'' \leq \ell'\}$ and by $\pm\infty$ the least or greatest element of $\mathcal{L}$ respectively. For $\iota, \iota' \in \mathcal{R}_n^{\mathcal{L}}$ we define $\iota \cap \iota'$ as

$$(\iota \cap \iota')(\ell) = \iota(\ell) \cap \iota'(\ell).$$

The domain of our analysis is then $\mathcal{A}_n = \mathcal{F}_n \times \mathrm{A}(\mathbb{F}_2^n, \mathbb{F}_2^n) \times \mathcal{R}_n^{\mathcal{L}}$.

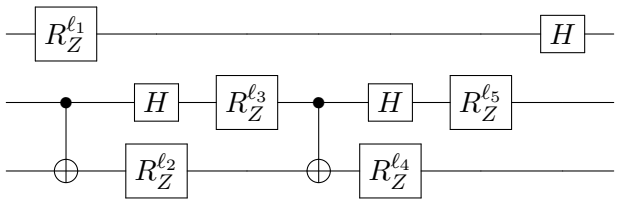
As in the phase-folding analysis, the algorithm proceeds by running a circuit C according to the abstract semantics $[\![C]\!]_a : \mathcal{A}_n \to \mathcal{A}_n$ on an input state. We define $[\![\cdot]\!]_a$ below:

$$\begin{aligned}
[\![X_i^{\ell}]\!]_a(f, A_{\mathbf{b}}, \iota) &= (f, A_{\mathbf{b} \oplus e_i}, \iota) \\
[\![\mathrm{CNOT}_{i,j}^{\ell}]\!]_a(f, A_{\mathbf{b}}, \iota) &= (f, E_{i,j} A_{\mathbf{b}}), \iota) \\
[\![R_Z(\theta)_i^{\ell}]\!]_a(f, A_{\mathbf{b}}, \iota) &= (f \cup \{(A_i^T, \ell)\}, A_{\mathbf{b}}, \iota) \\
[\![U_{n-k+1,\ldots,n}^{\ell}]\!]_a(f, A_{\mathbf{b}}, \iota) &= (f', (A_k A_k^g)_{\mathbf{b}_k}, \iota \cap \iota') \text{ where} \\
f' &= \bigcup_{(\mathbf{y},\ell') \in f \text{ s.t. } (A_k^g A_k)^T(\mathbf{y},\mathbf{0}) = (\mathbf{y},\mathbf{0})} \{((A_k^g)^T(\mathbf{y}, \mathbf{0}), \ell')\} \\
\iota' &= \bigcap_{(\mathbf{y},\ell') \in f \text{ s.t. } (A_k^g A_k)^T(\mathbf{y},\mathbf{0}) \neq (\mathbf{y},\mathbf{0})} \ell' \mapsto [-\infty, \ell] \\
[\![C \fatsemi C']\!]_a &= [\![C']\!]_a \circ [\![C]\!]
\end{aligned}$$

Additionally we define $[\![\cdot]\!]_a^{-1}$ identically, except that in the definition of ι', $\ell' \mapsto [\ell, \infty]$.

The full algorithm is given in Algorithm 4.2. As with phase folding, a labelled circuit and an initial state is given. The algorithm then uses the abstract semantics of C to compute an upper bound on the range of each phase gate, as well as the output state $A'_{\mathbf{b}'}$. Using the output state as the initial state for the (inverse) semantics of $C^{\dagger}$ then achieves a lower bound for each phase gate.

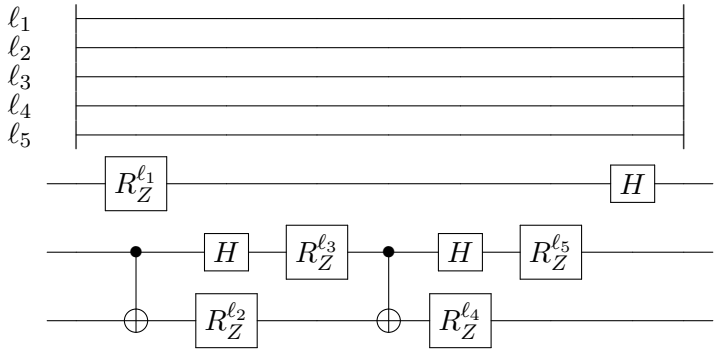
Example 4.3.1. Consider the following circuit:



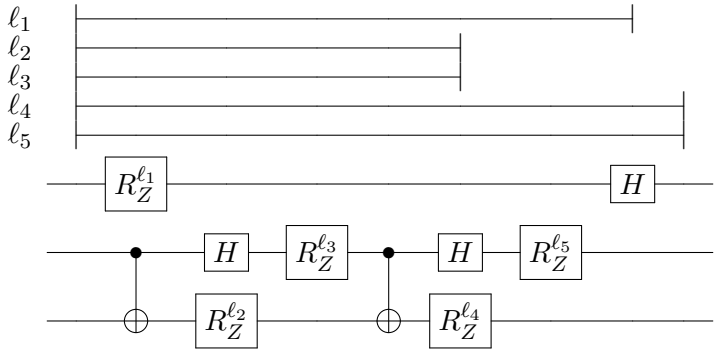
Algorithm 4.2 Phase range analysis

1: **function** PHASE-RANGE(C, $A_{\mathbf{b}}$)
2: Set $\iota(\ell) = [-\infty, \infty]$ for all $\ell \in \mathcal{L}$
3: $(f', A'_{\mathbf{b}'}, \iota') \leftarrow [\![C]\!]_a(\emptyset, A_{\mathbf{b}}, \iota)$
4: $(f'', A''_{\mathbf{b}''}, \iota'') \leftarrow [\![C^\dagger]\!]_a^{-1}(\emptyset, A'_{\mathbf{b}'}, \iota')$
5: **return** ι''
6: **end function**

The algorithm begins with each ℓ_i ranging over the entire circuit:

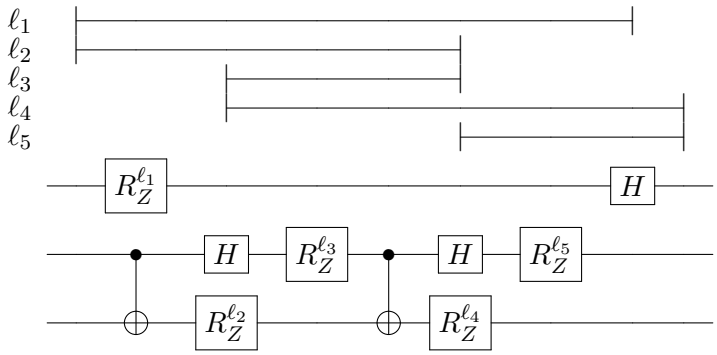


Running the forward phase of the analysis we find an upper bound for the range of each phase gate – in particular, the point after which the parity the phase gate rotates on can no longer appear as a state.



Finally, running the backwards phase of the analysis gives a lower bound, corresponding

to the point before which the parity cannot appear.



In many problems – for instance, T-depth, T-count and CNOT-count optimization – it is often desirable to associate phases such that the number of intervals (CNOT-dihedral chunks) which contain phase terms is minimized. Visually this can be achieved by placing the minimal number of vertical lines bisecting every range. For the circuit above, a minimal solution is obtained by dividing up the phases between the second and third intervals. We leave discussion of more advanced strategies to individual CNOT-dihedral synthesis algorithms.

4.4 Experiments

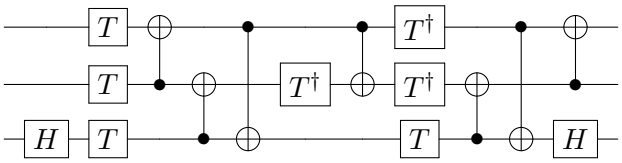
We implemented both Algorithm 4.1 and Algorithm 4.2 in Haskell in the compiler backend Feynman. To evaluate Algorithm 4.1 we ran it on a suite of benchmarks, described below; evaluation of Algorithm 4.2 is left to chapters 5 and 6 where specific CNOT-dihedral synthesis algorithms are given. Experiments were run in Linux on a quad-core 64-bit Intel Core i7 2.40 GHz processor and 8 GB RAM.

Benchmarks As the set of benchmarks we use to evaluate phase-folding will be used throughout this thesis to evaluate other algorithms, we describe them here. The benchmarks suite is comprised of circuits taken from the literature or adapted from publicly available[1] circuits. The full suite is including with Feynman. The benchmarks are primarily reversible implementations of arithmetic, as such circuits are believed to be the primary source of complexity in quantum algorithms [IAR13]. The benchmarks suite also contains implementations of strict quantum algorithms – namely, Grover's algorithm with oracle

[1] http://webhome.cs.uvic.ca/ dmaslov/

$f(\mathbf{x}) = \neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \wedge \neg x_5$ and the Quantum Fourier Transform on 4 qubits with higher-order rotation gates approximated over Clifford+T.

All benchmarks are initially written over Clifford+T and multiply-controlled Toffoli gates. To generate Clifford+T implementations, k-controlled Toffoli gates are decomposed into $2k + 3$ Toffoli gates (i.e. TOF) with $k - 2$ 0-initialized ancillas, as per the standard decomposition in [NC00]. Toffoli gates are further decomposed into Clifford+T gates using the T-depth minimal decomposition [AMMR13]



This decomposition was chosen as it achieves the minimal T-count and T-depth, and hence is the lowest-cost decomposition in most fault-tolerant frameworks.

Results Table 4.1 reports the results of our experiments. The phase-folding algorithm was run on each circuit, followed by a simple gate cancellation phase where adjacent gates that are inverses of one another are cancelled. We include this phase as it is a standard optimization which can further decrease some gate counts since the removal of phase gates allows some new trivial reductions. It should be noted that the phase gate counts are identical with and without the additional gate cancellation pass – that is, *all of the phase gate reductions are a result of phase-folding.* Additionally, average reductions are not given for S and Z gates, as they were typically increased from 0 due to T gate merging.

The results show that T-count is reduced by an average of 42.1% across all benchmarks, with a maximum reduction of 65.7% for the VBE-Adder_3 benchmark. This shows the remarkable amount of phase gate cancellation possible in real-world quantum circuits. Moreover, depth, T-depth, and CNOT-counts were decreased in *every benchmark.* On the other hand, H- and X-counts saw a small *increase* – 0.5% and 1.3%, respectively – which is due to the global phase correction using the $(HS)^3 = \omega$ and $XSXS = i$, $ZXZX = (-1)$ decompositions.

The run-times also show that the algorithm scales to some reasonably large circuits – particularly the GF(2^{256})-Mult and HWB_{12} benchmarks, with circuit volumes $917,291 * 768 = 704,479,488$ and $396,101 * 20 = 7,922,020$ and run-times of $14h$ and $3h$, respectively. The high degree of difficulty of the HWB benchmarks appears to be due to the very high number of Hadamard gates, which induce costly generalized inverse computations. In either case, peak memory usage was less than 1 GB, which could likely be reduced in a less memory-intensive programming language.

Table 4.1: Phase-folding optimization results. n gives the number of qubits in the circuit. Original gives metrics for the original circuit, Phase-folding gives the metrics and run-time after optimization with Algorithm 4.1.

Benchmark	n	Original								Phase-folding								
		H	X	CNOT	Z	S	T	Depth	T-depth	H	X	CNOT	Z	S	T	Depth	T-depth	Time (s)
Grover_5	9	100	65	336	0	0	336	457	144	100	67	296	3	12	154	420	96	0.012
Mod 5_4	5	6	1	32	0	0	28	41	12	6	1	32	0	0	16	37	10	<0.001
VBE-Adder_3	10	10	0	80	0	0	70	79	24	10	0	64	0	6	24	61	13	<0.001
CSLA-MUX_3	15	20	0	90	0	0	70	66	21	20	0	82	0	0	62	65	20	0.001
CSUM-MUX_9	30	28	28	196	0	0	196	59	18	28	28	196	0	28	84	55	6	0.006
QCLA-Com_7	24	37	15	215	0	0	203	81	27	37	15	183	3	5	95	68	19	0.007
QCLA-Mod_7	26	82	7	441	0	0	413	197	66	82	9	425	0	26	237	181	49	0.025
QCLA-Adder_10	36	50	0	267	0	0	238	73	24	50	0	249	0	10	162	66	19	0.014
Adder_8	24	80	12	466	0	0	399	223	69	80	12	436	0	16	215	206	49	0.024
RC-Adder_6	14	22	8	104	0	0	77	104	33	22	8	92	0	0	47	98	29	0.002
Mod-Red_21	11	30	14	122	0	0	119	156	48	30	14	110	0	1	73	148	40	0.002
Mod-Mult_55	9	14	8	55	0	0	49	50	15	17	10	55	1	4	35	49	13	<0.001
Mod-Adder_1024	28	330	0	2,005	0	0	1,995	2,503	831	330	0	1,691	0	7	1,011	2,170	593	0.267
Mod-Adder_1048576	58	2,416	0	16,680	0	0	16,660	21,789	7,292	2,416	0	15,772	111	2,923	7,340	18,485	4,570	49.753
Cycle 17_3	35	684	0	4,742	0	0	4,739	5,968	2,001	684	0	4,028	19	48	1,955	5,053	1,248	1.470
GF(2^4)-Mult	12	14	0	115	0	0	112	99	36	14	0	115	3	2	68	91	26	0.001
GF(2^5)-Mult	15	18	0	179	0	0	175	137	51	18	0	179	2	2	111	126	38	0.002
GF(2^6)-Mult	18	22	0	257	0	0	252	163	60	22	0	257	6	15	150	151	42	0.005
GF(2^7)-Mult	21	26	0	349	0	0	343	195	72	26	0	349	6	4	217	180	56	0.008
GF(2^8)-Mult	24	30	0	469	0	0	448	233	84	30	0	469	6	4	264	209	58	0.013
GF(2^9)-Mult	27	34	0	575	0	0	567	258	96	34	0	575	6	4	351	238	74	0.017
GF(2^{10})-Mult	30	38	0	709	0	0	700	290	108	38	0	709	6	25	410	267	74	0.025
GF(2^{16})-Mult	48	62	0	1,837	0	0	1,792	489	180	64	0	1,837	12	8	1,040	433	122	0.167
GF(2^{32})-Mult	96	126	0	7,292	0	0	7,168	1,001	372	126	0	7,292	24	16	4,128	881	250	3.143
GF(2^{64})-Mult	192	254	0	28,861	0	0	28,672	2,025	756	254	0	28,861	48	32	16,448	1,777	506	173.013
GF(2^{128})-Mult	384	510	0	115,069	0	0	114,688	4,071	1,524	510	0	115,069	96	64	65,664	3,567	1,018	3,268.625
GF(2^{256})-Mult	768	1,022	0	459,517	0	0	458,752	8,162	3,060	1,022	0	459,517	192	128	262,400	7,146	2,042	52,568.267
Ham_15 (low)	17	38	0	259	0	0	161	263	69	38	0	253	0	4	97	249	52	0.006
Ham_15 (med)	17	98	0	616	0	0	574	750	240	98	0	558	7	8	242	641	131	0.024
Ham_15 (high)	20	360	0	2,500	0	0	2,457	3,018	996	360	0	2,238	12	27	1,021	2,558	574	0.316
HWB_6	7	26	6	131	0	0	105	150	45	26	6	129	4	7	75	141	38	0.002
HWB_8	12	1,142	158	7,508	0	0	5,425	7,793	2,106	1,142	160	7,416	135	689	3,531	7,448	1,748	4.451
HWB_10	16	5,240	871	36,087	0	0	26,579	36,530	10,191	5,240	871	35,269	572	4,080	15,921	34,226	8,000	238.162
HWB_12	20	28,412	4,174	204,174	0	0	159,341	216,030	63,090	28,412	4,176	200,868	3,908	26,850	85,897	201,287	46,309	10,434.166
QFT_4	5	42	0	48	0	22	69	142	48	42	0	48	18	0	67	142	48	0.002
$\Lambda_3(X)$	5	6	0	21	0	0	21	27	9	6	0	19	0	0	15	27	9	<0.001
$\Lambda_3(X)$ (Barenco)	5	6	0	28	0	0	28	36	12	6	0	26	0	0	16	36	10	<0.001
$\Lambda_4(X)$	7	10	0	35	0	0	35	45	15	10	0	31	0	0	23	44	14	<0.001
$\Lambda_4(X)$ (Barenco)	7	10	0	56	0	0	56	72	24	10	0	52	0	0	28	69	15	<0.001
$\Lambda_5(X)$	9	14	0	49	0	0	49	63	21	14	0	43	0	0	31	61	19	<0.001
$\Lambda_5(X)$ (Barenco)	9	14	0	84	0	0	84	108	36	14	0	78	0	0	40	102	20	0.001
$\Lambda_{10}(X)$	19	34	0	119	0	0	119	153	51	34	0	103	0	0	71	146	44	0.005
$\Lambda_{10}(X)$ (Barenco)	19	34	0	224	0	0	224	288	96	34	0	208	0	0	100	267	45	0.005
Average reduction (%)										-0.5	-1.3	5.4	N/A	N/A	42.1	8.2	26.7	

4.5 Related work

Phase gate merging Recently, Nam *et al.* [NRS+18] developed a similar algorithm for merging phase gates in Clifford+R_Z circuits, possibly with parametrized gates. Their work also builds on [AMM14], extracting the logic of phase gate merging from the circuit re-synthesis subroutine to allow optimization without increasing any gate costs. In comparison to their work, our algorithm is applicable to and sound in the presence of uninterpreted gates, and hence can merge phase gates *over any gate set*. Their phase merging algorithm was also informally described, while ours was described in an analogous way to classical compiler optimizations (via abstract interpretation), making a formal analysis possible and allowing it to be easily implemented in other compilers.

Our work additionally diverges in the handling of optimization beyond the merging of phase gates. Specifically, they pair phase gate merging with efficient mapping strategies as well as local re-writes through a collection of circuit identities. In contrast, our methods focus on the grouping of phase terms into CNOT-dihedral subcircuits which can then be re-synthesized. This allows us to study and leverage optimal synthesis problems over such circuits, which is the topic of the following two chapters.

Abstract interpretation The presentation of the phase-folding algorithm was inspired by *abstract interpretation* [CC77], and particularly in the quantum domain by Perdrix's *entanglement analysis* [Per08]. However, in comparison to those works we have a very simple program model without recursion, which allows us to do away with most of the heavy machinery of abstract interpretation. On the other hand, the connection to abstract interpretation motivates the possibility of extending phase-folding to quantum programs which include measurement and classical control by abstracting the *circuit collecting semantics* – in analogy to (trace-based) collecting semantics, the set of all possible sequences of gates a program applies during the course of execution. We leave this as a direction for future work.

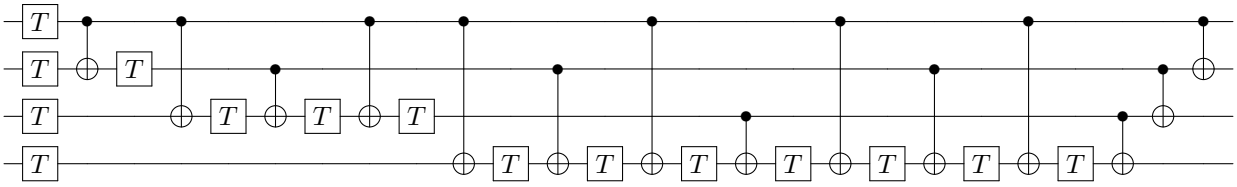
Chapter 5

T-count optimization

The phase-folding optimization of the previous chapter is known to be strictly heuristic, even when restricted to CNOT-dihedral circuits. It was observed in [AMM14] that

$$e^{\frac{2\pi i}{8}\sum_{\mathbf{y}\in\mathbb{F}_2^4}\chi_{\mathbf{y}}(\mathbf{x})} = 1,$$

and hence the circuit



with sum-over-paths action $|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{8}\sum_{\mathbf{y}\in\mathbb{F}_2^4}\chi_{\mathbf{y}}(\mathbf{x})}|\mathbf{x}\rangle$ is equal to the identity circuit. In this case however, phase-folding will not reduce any of the T gates as each is applied to a different parity of the input state. Seen another way, the phase-folding algorithm exploits equalities *between circuits with the same phase polynomial*, since for CNOT-dihedral circuits in particular it computes the exact phase polynomial and implements it with the minimal number ($|\mathrm{supp}(\widehat{f})|$) of R_Z gates. However, as Figure 5.1 illustrates, there exist equalities *between circuits with different phase polynomials* which are not exploited by the phase folding algorithm. Such equalities arise due to the equivalence $f \equiv f' \mod 2\pi$.

In this chapter we consider the problem of optimally synthesizing a CNOT-dihedral circuit given its sum-over-paths action – that is, a phase polynomial

$$f(\mathbf{x}) = \widehat{f}(\mathbf{0}) + \sum_{\mathbf{y}\in\mathbb{F}_2^n} \widehat{f}(\mathbf{y})\chi_{\mathbf{y}}(\mathbf{x}),$$

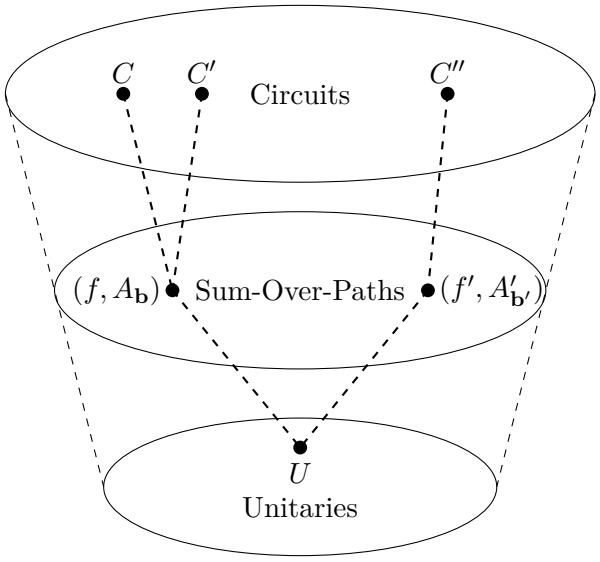


Figure 5.1: Relationship between circuit, path integral and unitary representations. All three circuits have the same unitary semantics, while only C and C' have the same (canonical) sum-over-paths action.

and affine transformation $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$. Here we define the cost to be the number of highest-order phase gates, hence an optimal circuit is one with *the fewest phase gates of highest order*. In particular, we consider the CNOT-dihedral group of order $2m$ for any $m \in \mathbb{N}$, and optimize the number of $Z_m = R_Z(2\pi/m)$ gates. We consider this version of the problem as a phase gate of any angle can always be decomposed as gates of smaller angle – for instance, $Z_m = Z_{m/2}Z_{m/2}$. This situation is reflected in fault-tolerance constructions, where smaller angle gates are generally progressively more difficult to implement [DCP15]. We note that $Z_m^k = R_Z(2\pi k/m)$, $0 \leq k < m$ can be implemented with minimal cost in this model as

$$Z_m^k := \prod_{i=0}^{\lfloor \log m \rfloor} Z_{2^i m}^{k_i},$$

where $k_{\lfloor \log m \rfloor} \dots k_1 k_0$ is the binary expansion of k. We hence say that the Z_m-cost of Z_m^k is 1 if $k = 1 \mod 2$ and 0 otherwise – for instance, the T cost of $T^6 = ZS$ is 0.

We prove that for any $m = 2^k l$ where l is odd, optimizing the number of Z_m gates in the n-qubit CNOT-dihedral group $\langle \mathrm{CNOT}, X, Z_m \rangle$ is polynomial-time equivalent to minimum-distance decoding for the length $2^n - 1$ *punctured Reed-Muller code* of order $n-k-1$, $\mathcal{RM}(n-k-1, n)^*$. As a consequence, we give a concrete optimization algorithm

which leverages existing Reed-Muller decoders to perform phase gate optimization. We also present several important corollaries, including that the optimal synthesis problem is polynomial-time solvable for $k \leq 2$, that the problem reduces to computing symmetric 3-tensors for $k = 3$, and that the number of T gates in a $\{\text{CNOT}, X, T\}$ circuit is $O(2^n)$.

This work appears in [AM16] and is implemented in the T-par[1] software package.

5.1 Coding theory

We first introduce some concepts from coding theory (see, e.g., MacWilliams & Sloane [MS78]). A length n *binary linear code* is a linear subspace C of $\mathbb{F}_2^n$ and elements of C are called *codewords*. A central concept to coding theory is that of *decoding* – given a received vector $\mathbf{w} = \mathbf{x} \oplus \mathbf{e}$ where $\mathbf{x} \in C$ and $\mathbf{e} \in \mathbb{F}_2^n$ is some error vector, find $\mathbf{x}$. We define the *(Hamming) weight* of a binary vector $\mathbf{x} \in \mathbb{F}_2^n$, denoted $|\mathbf{x}|$, as the number of non-zero entries of $\mathbf{x}$, and further define the *(Hamming) distance* $\delta(\mathbf{x}, \mathbf{y})$ between two binary vectors $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$ to be the weight of their sum:

$$\delta(\mathbf{x}, \mathbf{y}) = |\mathbf{x} \oplus \mathbf{y}|.$$

A decoding $\mathbf{y} \in C$ of a received vector $\mathbf{w}$ is said to be a *minimum-distance decoding* of $\mathbf{w}$ in C if it minimizes $\delta(\mathbf{w}, \mathbf{y})$, and the minimum-distance decoding problem is likewise to compute a minimum-distance decoding of a received vector.

Definition 5.1.1 (minimum-distance decoding problem for C)**.** Given a vector $\mathbf{w} \in \mathbb{F}_2^n$, find $\mathbf{y} \in C$ such that for all $\mathbf{z} \in C$, $\delta(\mathbf{w}, \mathbf{y}) \leq \delta(\mathbf{w}, \mathbf{z})$.

The problem of finding a minimum distance decoding is closely related to the more general *closest vector problem* over a lattice, and in fact coincides with the closest vector problem over the lattice C with the Hamming weight as the norm. Minimum distance decoding is commonly studied as it is equivalent to maximum likelihood decoding when errors are independent.

The *covering radius* of a code gives the maximum distance of a minimum-distance decoding.

Definition 5.1.2 (covering radius)**.** The covering radius of a length n binary code C is

$$\rho(C) = \max_{\mathbf{w} \in \mathbb{F}_2^n} \min_{\mathbf{y} \in C} \delta(\mathbf{w}, \mathbf{y}).$$

[1] https://github.com/meamy/t-par

The family of binary *Reed-Muller codes* [Mul54, Ree54] can be defined as the *evaluation vectors* of multivariate Boolean polynomials up to a maximum degree, called the code's *order*. In particular, let $\mathbb{F}_2[X_1, X_2, \ldots, X_n]$ be the ring of polynomials in n variables over $\mathbb{F}_2$. We use the symbols $X_1, X_2, \ldots, X_n$ to denote formal variables so as to differentiate them from binary vectors and values. Given $f \in \mathbb{F}_2[X_1, X_2, \ldots, X_n]$ we define the *evaluation vector* of (the polynomial function) f to be the length 2^n binary vector consisting of the evaluation of f at all non-zero inputs ordered by their (little-endian) integer values – i.e.

$$\begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(2^n-1) \end{bmatrix} = \begin{bmatrix} f(0,0,\ldots,0) \\ f(1,0,\ldots,0) \\ \vdots \\ f(1,1,\ldots,1) \end{bmatrix}.$$

We denote the evaluation vector of a polynomial function f by bold upright font, $\mathbf{f}$.

The *(total) degree* of a monomial $X^{\mathbf{y}} = X_1^{y_1} X_2^{y_2} \cdots X_n^{y_n}$ is the sum of its exponents:

$$\deg(X^{\mathbf{y}}) = \sum_{i=1}^{n} y_i = |\mathbf{y}|.$$

The degree of a polynomial function $f \in \mathbb{F}_2[X_1, X_2, \cdots X_n]$, denoted $\deg(f)$, is defined as the maximum total degree of each monomial. Since $X^2 = X$ for all $X \in \mathbb{F}_2$, the evaluation vectors of polynomial functions are unique over the quotient ring $\mathbb{F}_2[X_1, X_2, \ldots, X_n]/\langle X_1^2 - X_1, \ldots, X_n^2 - X\rangle$, and so we only consider reduced polynomials over this ring – that is, with exponents in $\mathbb{F}_2^n$.

Identifying the monomial $X^{\mathbf{y}}$ with the Boolean function $f(X_1, X_2, \ldots, X_n) = X^{\mathbf{y}}$, we denote the evaluation vector of $X^{\mathbf{y}}$ by $\mathbf{X}^{\mathbf{y}}$, which is equal to the component-wise multiple of evaluation vectors ${\mathbf{X}_{\mathbf{i}}}^{y_i}$. It can be easily verified that for any Boolean polynomial

$$f = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} X^{\mathbf{y}},$$

the evaluation vector of f is

$$\mathbf{f} = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}$$

with exponentiation of a Boolean vector also defined component-wise. Table 5.2 illustrates the evaluation vectors of the 2^n monomials on n variables, from which is can be observed that they form a linear basis of $\mathbb{F}_2^{2^n}$, or functions $\mathbb{F}_2^{2^n} \to \mathbb{F}_2$.

We can now define the Reed-Muller family of codes and their punctured versions.

	0	1	2	3	4	$\cdots$	2^n-1
1	1	1	1	1	1	$\cdots$	1
X_1	0	1	0	1	0	$\cdots$	1
X_2	0	0	1	1	0	$\cdots$	1
X_1X_2	0	0	0	1	0	$\cdots$	1
X_3	0	0	0	0	1	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$X_1X_2\cdots X_n$	0	0	0	0	0	$\cdots$	1

Figure 5.2: Evaluation vectors for monomials of n variables.

Definition 5.1.3 (Reed-Muller codes)**.** The *Reed-Muller code* of order r and length 2^n, denoted $\mathcal{RM}(r,n)$, is the set of evaluation vectors of polynomials $f \in \mathbb{F}_2[X_1, X_2, \ldots, X_n]$ with degree at most r.

The *punctured* Reed-Muller code of order r and length 2^n-1, denoted $\mathcal{RM}(r,n)^*$, is the Reed-Muller code of order r and length 2^n with the first coordinate of every codeword removed. Equivalently, the punctured Reed-Muller code can be defined as the quotient

$$\mathcal{RM}(r,n)/\left\langle (1,0,\ldots,0) \right\rangle.$$

Example 5.1.4. The vector

$$\mathbf{1} + \mathbf{X_1X_2} + \mathbf{X_2} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix} + \begin{bmatrix}0\\0\\0\\1\end{bmatrix} + \begin{bmatrix}0\\0\\1\\1\end{bmatrix} = \begin{bmatrix}1\\1\\0\\1\end{bmatrix}$$

is in $\mathcal{RM}(2,2)$ but not $\mathcal{RM}(1,2)$. The equivalent codeword of $\mathcal{RM}(2,2)^*$ is $\begin{bmatrix}1 & 0 & 1\end{bmatrix}^T$.

We can now formally state our main result.

Theorem 5.1.5. *Let $m = 2^k l$ where l is odd. Then the optimal synthesis problem for $\langle \mathrm{CNOT}, X, Z_m \rangle$ with respect to Z_m-count, and minimum-distance decoding problem for $\mathcal{RM}(n-k-1,n)^*$ are polynomial-time equivalent – that is, there exist polynomial-time reductions from each to the other.*

The rest of this chapter first frames the Z_m-count optimization problem as a decoding problem, then gives a set of generators and studies some consequences, including Theorem 5.1.5.

5.2 Decoding phase polynomials

In this section we reduce the problem of synthesizing a CNOT-dihedral circuit with a minimum number of Z_m gates given its sum-over-paths action to a decoding problem over $\mathbb{Z}_m^{2^n}$ – specifically, minimizing the number of odd entries over $\mathbf{a} + G$ for some $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ where G is a particular subgroup of $\mathbb{Z}_m^{2^n}$.

First recall that the sum-over-paths action of an n-qubit CNOT-dihedral circuit is given by a phase polynomial $f : \mathbb{F}_2^n \to \mathbb{R}$ and affine transformation $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$ such that

$$|\mathbf{x}\rangle \mapsto e^{if(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle, \qquad f(\mathbf{x}) = \widehat{f}(\mathbf{0}) + \sum_{\mathbf{y} \in \mathbb{F}_2^n} \widehat{f}(\mathbf{y})\chi_{\mathbf{y}}(\mathbf{x}).$$

Since we are strictly working with CNOT-dihedral circuits of order $2m$ – that is, circuits over $\{\mathrm{CNOT}, X, Z_m\}$ for some $m \in \mathbb{N}$ – we factor $\frac{2\pi i}{m}$ out of the phase polynomial so that $f : \mathbb{F}_2^n \to \mathbb{Z}$ and the sum-over-paths action is

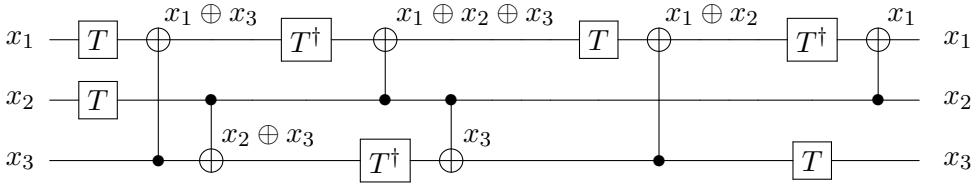
$$|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle.$$

As $e^{\frac{2\pi i}{m} f(\mathbf{x})} = e^{\frac{2\pi i}{m} f'(\mathbf{x})}$ whenever $f'(\mathbf{x}) = f(\mathbf{x}) \mod m$, we can reduce the Fourier coefficients mod m, giving $\widehat{f}(\mathbf{y}) \in \mathbb{Z}_m$. We write these reduced coefficients as a tuple $\mathbf{a} \in \mathbb{Z}_m^{2^n}$. We call this tuple an *implementation* of the phase function f, and denote the phase polynomial defined by a tuple $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ by $f_{\mathbf{a}}$, i.e.,

$$f_{\mathbf{a}}(\mathbf{x}) = a_{\mathbf{0}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}}\chi_{\mathbf{y}}(\mathbf{x}).$$

It was noted in Chapter 3 that *the Fourier coefficients of a sum-over-paths action directly correspond to the phase gates in a circuit.* Given a CNOT-dihedral circuit C, the phase function $f_{\mathbf{a}}$ is obtained by summing up the phase contribution of $\frac{2\pi k}{m}\chi_{\mathbf{y}}(\mathbf{x})$ or $\frac{2\pi k}{m}(1 \oplus \chi_{\mathbf{y}}(\mathbf{x})) = \frac{2\pi k}{m} + \frac{-2\pi k}{m}\chi_{\mathbf{y}}(\mathbf{x})$ of each $(Z_m)^k$ gate, applied to a qubit in the state $|\chi_{\mathbf{y}}(\mathbf{x})\rangle$ or $|1 \oplus \chi_{\mathbf{y}}(\mathbf{x})\rangle$, respectively. In the context of Z_m gates, we see that the number of Z_m gates – the number of $(Z_m)^k$ gates with odd k – is *at least* the number of odd entries of $\mathbf{a}$.

Example 5.2.1. Recall that the doubly-controlled Z gate CCZ can be written as the following (annotated) order 16 CNOT-dihedral circuit:



Summing up the contributions from each T or $T^\dagger$ gate we see that

$$f(x_1, x_2, x_3) = x_1 + x_2 - (x_1 \oplus x_2) + x_3 - (x_1 \oplus x_3) - (x_2 \oplus x_3) + (x_1 \oplus x_2 \oplus x_3),$$

which we can write as a tuple over $\mathbb{Z}_m$ by reducing $-1 = 7 \mod 8$, so $\mathbf{a} = (0, 1, 1, 7, 1, 7, 7, 1)$ Notably, $\mathbf{a}$ has 7 odd entries, the same as the number of T gates in the circuit.

It can be observed that the converse is also true, in that given a set of coefficients $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ with t odd entries, *a circuit implementing* $|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})}|\mathbf{x}\rangle$ *with at most* $t + 1$ Z_m *gates exists.* In particular, for each $\mathbf{y} \neq \mathbf{0}$ such that $a_{\mathbf{y}} \neq 0$, the parity $\chi_{\mathbf{y}}(\mathbf{x})$ is first computed via a sequence of CNOT gates, then a $(Z_m)^{a_{\mathbf{y}}}$ gate is applied to achieve a phase rotation of $e^{\frac{2\pi i}{m} a_{\mathbf{y}} \chi_{\mathbf{y}}(\mathbf{x})}$, and $\chi_{\mathbf{y}}(\mathbf{x})$ is uncomputed. Note in general that the transformation $|\mathbf{x}\rangle \mapsto |A\mathbf{x} + \mathbf{b}\rangle$ is implementable over just CNOT and X gates via Gaussian elimination (e.g., [PMH08]), hence the Z_m cost of $|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})}|A\mathbf{x}+\mathbf{b}\rangle$ is the same as $|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})}|\mathbf{x}\rangle$. Additionally, the global phase $e^{\frac{2\pi i}{m} a_{\mathbf{0}}}$ can be applied with the gate sequence

$$(Z_m)^{a_{\mathbf{0}}} X (Z_m)^{a_{\mathbf{0}}} X = e^{\frac{2\pi i}{m} a_{\mathbf{0}}} I.$$

Up to global phase, the number of Z_m gates required to implement $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ with t odd entries excluding $a_{\mathbf{0}}$ is *exactly* t.

Remark 5.2.2. Applying the global phase correction in this way uses two extra Z_m gates when $a_{\mathbf{0}}$ is odd. However, in cases where there exist $\mathbf{z} \neq \mathbf{0}$ such that $a_{\mathbf{z}}$ is odd, these extra gates can be removed by absorbing part of the global phase into the $e^{\frac{2\pi i}{m} a_{\mathbf{z}} \chi_{\mathbf{z}}(\mathbf{x})}$ phase. In particular,

$$e^{\frac{2\pi i}{m}(-a_{\mathbf{z}})(1 \oplus \chi_{\mathbf{z}}(\mathbf{x}))} = e^{\frac{2\pi i}{m}(-a_{\mathbf{z}} + a_{\mathbf{z}} \chi_{\mathbf{z}}(\mathbf{x}))},$$

so applying $X(Z_m)^{-a_{\mathbf{z}}} X$ instead of $(Z_m)^{a_{\mathbf{z}}}$ gives

$$\begin{aligned} |\mathbf{x}\rangle &\mapsto e^{\frac{2\pi i}{m}(-a_{\mathbf{z}} + f_{\mathbf{a}}(\mathbf{x}))}|A\mathbf{x} + \mathbf{b}\rangle \\ &\mapsto e^{\frac{2\pi i}{m}\left(a_{\mathbf{0}} - a_{\mathbf{z}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \chi_{\mathbf{y}}(\mathbf{x})\right)}|A\mathbf{x} + \mathbf{b}\rangle \end{aligned}$$

where the global phase $a_{\mathbf{0}} - a_{\mathbf{z}} = 0 \mod 2$ can be implemented without Z_m gates. As a result, the *only* situation where global phase requires an extra Z_m gate is when there are no other odd phases, hence we consider global phase effectively free.

This link between the number of odd coefficients of $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ and the number of Z_m gates in a circuit applying the phase rotation $e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})}$ is summed up in the proposition below

Proposition 5.2.3. *Given* $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ *and* $A_{\mathbf{b}} \in \mathrm{GA}(n, \mathbb{F}_2)$*, there exists a circuit* C *over* $\{\mathrm{CNOT}, X, Z_m\}$ *with* t Z_m *gates such that*

$$[\![C]\!] : |\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})} |A\mathbf{x} + \mathbf{b}\rangle$$

up to global phase if and only if there exists $\mathbf{a}' \in \mathbb{Z}_m^{2^n}$ *such that* $f_{\mathbf{a}}(\mathbf{x}) = f_{\mathbf{a}'}(\mathbf{x}) \mod m$ *for all* $\mathbf{x} \in \mathbb{F}_2^n$ *and* $\mathbf{a}'$ *has at most* t *odd entries excluding* $a'_{\mathbf{0}}$*. Moreover, if* $\mathbf{a}'$ *exists, the circuit is polynomial-time synthesizable in the number of non-zero entries of* $\mathbf{a}'$ *and* n*.*

Proof. We first note that for any $\mathbf{a}, \mathbf{a}' \in \mathbb{Z}_m^{2^n}$,

$$e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})} |A\mathbf{x} + \mathbf{b}\rangle\langle\mathbf{x}| = e^{\frac{2\pi i}{m} f_{\mathbf{a}'}(\mathbf{x})} |A\mathbf{x} + \mathbf{b}\rangle\langle\mathbf{x}|$$

if and only if $f_{\mathbf{a}}(\mathbf{x}) = f_{\mathbf{a}'}(\mathbf{x}) \mod m$ for all $\mathbf{x} \in \mathbb{F}_2^n$. Then, as noted above, a circuit C with phase function $f_{\mathbf{a}}(\mathbf{x})$ up to global phase and t Z_m gates gives $\mathbf{a}' \in \mathbb{Z}_m^{2^n}$ with at most t odd entries excluding $a'_{\mathbf{0}}$, and vice versa. Moreover, synthesis takes time polynomial in the volume of the output circuit – itself polynomial in the number of non-zero entries of $a'_{\mathbf{y}}$ and n – as the synthesis of each phase factor is trivial and the final affine synthesis step is polynomial in n.

□

For the rest of this chapter, we taking equality as meaning *equal up to global phase*. Moreover, we take the $\mathbb{Z}_m^{2^n}$ as informally meaning the "punctured" group $\mathbb{Z}_m^{2^n} / \langle (1, 0, \ldots, 0) \rangle \simeq \mathbb{Z}_m^{2^n - 1}$ – i.e. tuples $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ up to the value of $a_{\mathbf{0}}$. We take this informal approach to avoid a heavy notational burden, e.g., the original publication of this result [AM16]. Note also that the "number of odd coefficients excluding $a_{\mathbf{0}}$" in this case is taken to mean the number of odd entries in $\mathbb{Z}_m^{2^n - 1}$.

5.2.1 Coding interpretation

By Proposition 5.2.3, the problem of optimally synthesizing a CNOT-dihedral circuit, given by a set of Fourier coefficients $\mathbf{a} \in \mathbb{Z}_m^{2^n}$, with respect to Z_m is equivalent to the problem of finding some $\mathbf{a}' \in \mathbb{Z}_m^{2^n}$ such that

$$f_{\mathbf{a}}(\mathbf{x}) = f_{\mathbf{a}'}(\mathbf{x}) \mod m$$

for all $\mathbf{x} \in \mathbb{F}_2^n$ minimizing the number of odd entries – indeed, they are polynomial-time equivalent as reading the Fourier coefficients from a circuit takes polynomial time. We now

show that this corresponds to a minimization problem over a coset of $\mathbb{Z}_m^{2^n}$, which moreover is equivalent to minimization of the Hamming weight over the binary residues of this coset.

Given an element $\mathbf{a}$ of $\mathbb{Z}_m^{2^n}$, let $[\mathbf{a}]$ be the equivalence class of implementations of $f_\mathbf{a}$ – that is, $\mathbf{a}' \in [\mathbf{a}]$, denoted $\mathbf{a} \sim \mathbf{a}'$, if and only if

$$f_\mathbf{a}(\mathbf{x}) = f_{\mathbf{a}'}(\mathbf{x}) \mod m$$

for all $\mathbf{x}$. Let

$$C_m^n = \{\mathbf{c} \in \mathbb{Z}_m^{2^n} \mid f_\mathbf{c}(\mathbf{x}) = 0 \mod m \text{ for all } x \in \mathbb{F}_2^n\}$$

be the set of 0-everywhere phase polynomials on $\mathbb{Z}_m^{2^n}$. By the fact that the mapping $\mathbf{a} \mapsto f_\mathbf{a}$ is a group homomorphism, in that

$$f_\mathbf{a}(\mathbf{x}) + f_{\mathbf{a}'}(\mathbf{x}) = f_{\mathbf{a}+\mathbf{a}'}(\mathbf{x})$$

for all $\mathbf{x}$, it is easy to see that $C_m^n \triangleleft \mathbb{Z}_m^{2^n}$ is the kernel of this homomorphism, and moreover that $[\mathbf{a}] = \mathbf{a} + C_m^n$.

Proposition 5.2.4. *For any* $\mathbf{a} \in \mathbb{Z}_m^{2^n}$,

$$[\mathbf{a}] = \mathbf{a} + C_m^n$$

Proof. Suppose $\mathbf{c} \in C_m^n$. Then clearly

$$f_{\mathbf{a}+\mathbf{c}}(\mathbf{x}) = f_\mathbf{a}(\mathbf{x}) + f_\mathbf{c}(\mathbf{x}) = f_\mathbf{a}(\mathbf{x}) \mod m.$$

Likewise, if $\mathbf{a}' \in [\mathbf{a}]$, then

$$f_{\mathbf{a}-\mathbf{a}'}(\mathbf{x}) = f_\mathbf{a}(\mathbf{x}) - f_{\mathbf{a}'}(\mathbf{x}) = 0 \mod m,$$

hence $\mathbf{a} - \mathbf{a}' \in C_m^n$ and in particular $\mathbf{a}' = \mathbf{a} + \mathbf{c}$ for some $\mathbf{c} \in C_m^n$. □

As a consequence of Proposition 5.2.4, we see that the problem of finding an implementation of $f_\mathbf{a}$ minimizing Z_m count is equivalent to finding an element $\mathbf{c} \in C_m^n$ minimizing the number of odd entries in $\mathbf{a} + \mathbf{c}$. Combined with a *presentation* of C_m^n by generators, as given below, we can directly perform this optimization by generating elements of C_m^n.

Lemma 5.2.5. *Let* $m = 2^k l$ *where* l *is odd. Then*

$$C_m^n = \left\langle 2^i l \mathbf{X}^\mathbf{y} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \right\rangle.$$

We defer the proof of Lemma 5.2.5 to Section 5.4.

Example 5.2.6. Consider the case where $n = 4$, $m = 8 = 2^3$, and so

$$C_8^4 = \left\langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - 4 \right\rangle.$$

If we consider just the generators which have odd coefficients, we see that there is exactly one such generator – $\mathbf{X}^{\mathbf{0}} = \mathbf{1}$, the all 1 string. Now we can verify that $f_{\mathbf{1}}(\mathbf{x}) = 0 \mod 8$ for all $\mathbf{x} \in \mathbb{F}_2^4$ by computing:

$$f_{\mathbf{1}}(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^4} \chi_{\mathbf{y}}(\mathbf{x}) = 0 \mod 8,$$

which is exactly the example given at the beginning of this chapter.

Remark 5.2.7. As a consequence of Lemma 5.2.5, we also provide a characterization of the set of n-qubit unitaries implementable over $\{\text{CNOT}, X, Z_m\}$. In particular, it can be observed that

$$\langle \text{CNOT}, X, Z_m \rangle \simeq \mathbb{Z}_m^{2^n} / C_m^n \times \text{GA}(n, \mathbb{F}_2).$$

5.2.2 Connection to Reed-Muller codes

With the above set of generators given in Lemma 5.2.5, the optimization can be performed directly over C_m^n. However, the particular metric of Z_m-count optimality makes such optimization unnatural; as the number of odd entries in an element of $\mathbb{Z}_m^{2^n}$ doesn't form a norm, there are no natural reductions to the obvious problems of shortest vectors in a lattice or minimum distance decoding in more general linear codes. We instead reduce the optimization problem to a decoding problem over a binary code, where minimum-distance decoding corresponds exactly to Z_m-count optimization.

Defining $\text{Res}_2 : \mathbb{Z} \to \mathbb{F}_2$ as the function taking the binary residue of an integer and extending this component-wise to tuples, we see that the number of odd entries in $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ is equal to the weight of the binary residue vector, i.e. $|\text{Res}_2(\mathbf{a})|$. We can further see that

$$|\text{Res}_2(\mathbf{a} + \mathbf{c})| = |\text{Res}_2(\mathbf{a}) \oplus \text{Res}_2(\mathbf{c})| = \delta(\text{Res}_2(\mathbf{a}), \text{Res}_2(\mathbf{c})),$$

that, is the number of odd entries in $\mathbf{a} + \mathbf{c}$ – and hence Z_m-cost of implementing $f_{\mathbf{a}+\mathbf{c}}$ – is the Hamming distance from $\text{Res}_2(\mathbf{a})$ to $\text{Res}_2(\mathbf{c})$. As a result, optimizing the number of odd entries in $\mathbf{a} + \mathbf{c}$ over all $\mathbf{c} \in C_m^n$ is exactly the problem of minimum distance decoding

$\mathrm{Res}_2(\mathbf{a})$ over $\mathrm{Res}_2(C_m^n)$, the set of binary residue vectors of C_m^n. We further note that $\mathrm{Res}_2(C_m^n)$ is in fact a binary linear code, since

$$\mathrm{Res}_2(\mathbf{c}) \oplus \mathrm{Res}_2(\mathbf{c}') = \mathrm{Res}_2(\mathbf{c} + \mathbf{c}') \in \mathrm{Res}_2(C_m^n)$$

for any $\mathbf{c}, \mathbf{c}' \in C_m^n$, and as a direct consequence of Lemma 5.2.5 this code is exactly the length $2^n - 1$ punctured Reed-Muller code of order $n - k - 1$.

Theorem 5.2.8. *Let $m = 2^k l$ where l is coprime with 2. Then*

$$\mathrm{Res}_2(C_m^n) = \mathcal{RM}(n - k - 1, n)^*$$

Proof. By direct calculation and the fact that Res_2 is a homomorphism. In particular,

$$\begin{aligned}
\mathrm{Res}_2(C_m^n) &= \left\langle \mathrm{Res}_2\left(2^i l \mathbf{X}^{\mathbf{y}}\right) \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \right\rangle \\
&= \langle \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| \leq n - k - 1 \rangle \\
&= \mathcal{RM}(n - k - 1, n)^*
\end{aligned}$$

□

Note that $\mathrm{Res}_2(C_m^n)$ is the *punctured* Reed-Muller code as we are really working "up to global phase" – i.e. in $\mathbb{Z}_m^{2^n} / \langle (1, 0, \ldots, 0) \rangle \simeq \mathbb{Z}_m^{2^n - 1}$.

5.3 Applications

Before proving Lemma 5.2.5 in the next section, we discuss some consequences of Theorem 5.2.8.

5.3.1 Upper bounds on phase gate counts

As a consequence of Proposition 5.2.3 and Theorem 5.2.8, the covering radius of $\mathrm{Res}_2(C_{2^k l}^n) = \mathcal{RM}(n - k - 1, n)^*$ gives a tight upper bound on the number of Z_m gates required to implement a circuit over $\{\mathrm{CNOT}, X, Z_m\}$. Here we mean tight in the sense that there exists a unitary operator which requires a minimum of $\rho(\mathcal{RM}(n - k - 1, n)^*)$ Z_m gates to implement over $\{\mathrm{CNOT}, X, Z_m\}$. In particular, if $\rho(\mathcal{RM}(n - k - 1, n)^*) = t$, there there exists some equivalence class $[\mathbf{a}], \mathbf{a} \in \mathbb{Z}_m^{2^n}$ with binary residue $\mathbf{w}$ such that

$$\min_{\mathbf{y} \in \mathcal{RM}(n-k-1,n)^*} \delta(\mathbf{w}, \mathbf{y}) = \min_{\mathbf{c} \in C_{2^k l}^n} \delta(\mathrm{Res}_2(\mathbf{a}), \mathrm{Res}_2(\mathbf{c})) = t.$$

While no analytic formula is known for the covering radius of higher-order Reed-Muller codes, some asymptotic upper bounds are known, which consequently give upper bounds on the number of Z_m gates required to implement. In particular, Cohen and Litsyn [CL92] showed that for large n and orders r where $n - r \geq 3$,

$$\rho(\mathcal{RM}(r,n)) \leq \frac{n^{n-r-2}}{(n-r-2)!}.$$

Since the covering radius of $\mathcal{RM}(r,n)^*$ is trivially bounded above by $\rho(\mathcal{RM}(r,n))$, it turns out that for sufficiently large n and $k \geq 2$, $\rho(\mathcal{RM}(n-k-1,n)^*) \leq \frac{n^{k-1}}{(k-1)!}$. In particular, for the important case of $\{\text{CNOT}, X, T\}$, $k = 3$ and hence we obtain an $O(n^2)$ bound on the number of T gates needed.

Theorem 5.3.1. *Any order* 16 CNOT*-dihedral operator can be implemented with at most* $O(n^2)$ T *gates over* $\{\text{CNOT}, X, T\}$.

5.3.2 Optimization algorithm

Theorem 5.2.8 implies that some element $\mathbf{c} \in C^n_{2^k l}$ minimizing the number of odd coefficients in $\mathbf{a} + \mathbf{c}$ can be found by decoding $\text{Res}_2(\mathbf{a})$ in $\mathcal{RM}(n-k-1,n)^*$ – however, the decoding itself isn't enough to find $\mathbf{c}$. In particular, decoding the binary residue $\text{Res}_2(\mathbf{a})$ produces a minimal residue $\mathbf{w} \in \mathbb{F}_2^{2^n}$ of a codeword $\mathbf{c}$ in $C^n_{2^k l}$. To actually synthesize a Z_m-count minimal circuit, we need to compute $\mathbf{c}$ with $\text{Res}_2(\mathbf{y}) = \mathbf{w}$ and then synthesize the phase function $f_{\mathbf{a}+\mathbf{c}}$. Fortunately, there is an easy way to do this, given the following lemma.

Lemma 5.3.2. *For all* $\mathbf{y} \in \mathbb{F}_2^n$ *with* $|\mathbf{y}| \leq n-k-1$ *we have* $l\mathbf{X}^\mathbf{y} \in C^n_{2^k l}$.

Proof. Consequence of Lemma 5.2.5. In particular, $2^i l\mathbf{X}^\mathbf{y} \in C^n_{2^k l}$ for any $|\mathbf{y}| - i \leq n-k-1$, and hence substituting $i = 0$ we have $l\mathbf{X}^\mathbf{y} \in C^n_{2^k l}$ whenever $|\mathbf{y}| \leq n-k-1$. □

Using Lemma 5.3.2 and the definition of $\mathcal{RM}(r,n)^*$, we can write a decoded word $\mathbf{w}$ as a (binary) sum of monomials of degree at most $n-k-1$, then *reinterpret* this sum (scaled by l) over $\mathbb{Z}_m$. Specifically, if $\mathbf{w} = \bigoplus_{\mathbf{y}\in\mathbb{F}_2^n} a_\mathbf{y}\mathbf{X}^\mathbf{y}$ for some $\{a_\mathbf{y}\} \subseteq \mathbb{F}_2$, then we choose $\mathbf{c} = \sum_{\mathbf{y}\in\mathbb{F}_2^n} a_\mathbf{y} l\mathbf{X}^\mathbf{y}$, which by Lemma 5.3.2 is in $C^n_{2^k l}$, and further note that

$$\text{Res}_2(\mathbf{c}) = \text{Res}_2\left(\sum_{\mathbf{y}\in\mathbb{F}_2^n} a_\mathbf{y} l\mathbf{X}^\mathbf{y}\right) = \bigoplus_{\mathbf{y}\in\mathbb{F}_2^n} \text{Res}_2(a_\mathbf{y} l)\mathbf{X}^\mathbf{y} = \mathbf{w}.$$

Using this fact we develop an algorithm for the optimization of Z_m-count based on Reed-Muller decoding.

Algorithm 5.1 RM-OPTIMIZE(C, $m = 2^k l$)

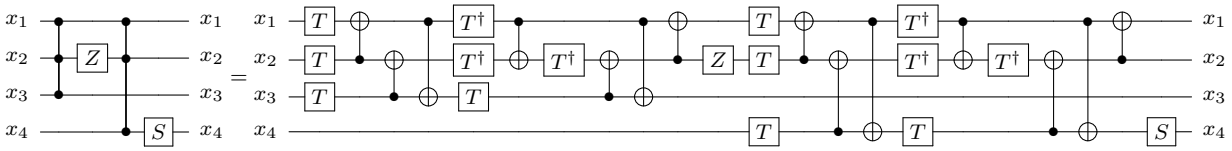
1: Compute sum-over-paths action $(f_{\mathbf{a}}, A_{\mathbf{b}})$ of C
2: $\mathbf{w} \leftarrow$RM-DECODE($n-k-1$, n, $\mathrm{Res}_2(\mathbf{a})$)
3: Write $\mathbf{w}$ as a Boolean polynomial $\mathbf{w} = \bigoplus_{\mathbf{y}\in\mathbb{F}_2^n} a_{\mathbf{y}}\mathbf{X}^{\mathbf{y}}$
4: $\mathbf{c} \leftarrow \sum_{\mathbf{y}\in\mathbb{F}_2^n} a_{\mathbf{y}} l \mathbf{X}^{\mathbf{y}}$
5: **return** CNOT-DIHEDRAL-SYNTH($f_{\mathbf{a}+\mathbf{c}}, A_{\mathbf{b}}$)

Algorithm 5.1 summarizes our algorithm for Z_m-count optimization in $\{\mathrm{CNOT}, X, Z_m\}$ circuits, where $m = 2^k l$ for odd l. The algorithm works by computing the sum-over-paths action of the circuit C as $(f_{\mathbf{a}}, A_{\mathbf{b}})$, where $\mathbf{a}$ gives the coefficients of the phase polynomial. The vector of residues modulo 2 is then decoded as $\mathbf{w}$ in $\mathcal{RM}(n-k-1, n)^*$ using the procedure RM-DECODE($n-k-1$, n, $\mathrm{Res}_2(\mathbf{a})$). A vector $\mathbf{c} \in C^n_{2^k l}$ with $\mathrm{Res}_2(\mathbf{c}) = \mathbf{w}$ is then computed and added to the original set of coefficients, and a circuit is synthesized with the new set of coefficients. In particular, the procedure CNOT-DIHEDRAL-SYNTH takes a sum-over-paths action $(f_{\mathbf{a}}, A_{\mathbf{b}})$ (of order $2m$) and synthesizes a circuit over $\{\mathrm{CNOT}, X, Z_m\}$ implementing

$$|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{a}}(\mathbf{x})} |A\mathbf{x} + \mathbf{b}\rangle.$$

The algorithm is parametric in both the decoder and the synthesis procedure, meaning any variable order Reed-Muller decoder may be used to implement RM-DECODE. If a minimum distance decoder is used, Algorithm 5.1 synthesizes a minimal Z_m-count circuit. Likewise, any synthesis procedure may be used to implement CNOT-DIHEDRAL-SYNTH – for instance, the T-depth minimizing T-par algorithm [AMM14], or the CNOT-minimizing synthesis algorithm we describe in the next chapter.

Example 5.3.3. Consider the circuit below:



By annotating the circuit and computing the phase contributions, we find the phase polynomial (reduced mod 8) for the above circuit is

$$f(\mathbf{x}) = 2x_1 + 6x_2 + 6(x_1 \oplus x_2) + x_3 + 7(x_1 \oplus x_3) + 7(x_2 \oplus x_3) + (x_1 \oplus x_2 \oplus x_3) + 3x_4 + 7(x_1 \oplus x_4) + 7(x_2 \oplus x_4) + (x_1 \oplus x_2 \oplus x_4).$$

Writing the Fourier coefficients as a tuple $\mathbf{a} \in \mathbb{Z}_8^{2^n}$ we get

$$\mathbf{a} = (0, 2, 6, 6, 1, 7, 7, 1, 3, 7, 7, 1, 0, 0, 0, 0),$$

which has 8 odd entries – hence can be implemented directly with 8 T gates, giving a reduction of 6 T gates and corresponding to phase-folding as in Chapter 4.

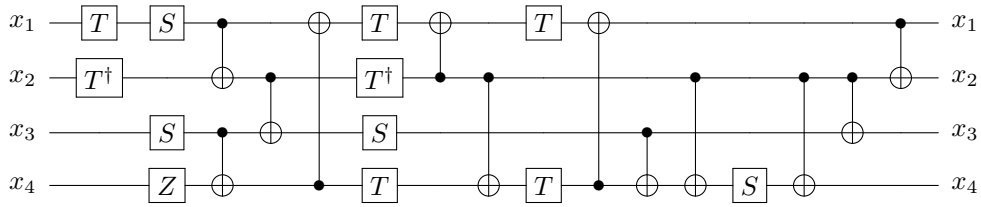
Next we optimize the implementation of f further by decoding

$$\mathrm{Res}_2(\mathbf{a}) = (0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0)$$

in the code $\mathcal{RM}(0,4)^*$. As $\mathcal{RM}(0,4)^*$ is the set of evaluation vectors for degree 0 Boolean polynomials, there are exactly two vectors to choose from, corresponding to the constant 0 and 1 functions. Since the all 1 vector achieves the minimum distance of 7 (excluding the first coordinate) from $\mathrm{Res}_2(\mathbf{a})$, we choose $\mathbf{w}$ to be the all 1 vector – i.e. $\mathbf{w} = \mathbf{1}$. Writing $\mathbf{c} = \mathbf{1}$, by Lemma 5.3.2 we have $\mathbf{c} \in C_3^4$ with $\mathrm{Res}_2(\mathbf{c}) = \mathbf{w}$. Finally we synthesize a circuit for the tuple $\mathbf{a} + \mathbf{c} = (0, 3, 7, 7, 2, 0, 0, 2, 4, 0, 0, 2, 1, 1, 1, 1)$, corresponding to the phase polynomial

$$\begin{aligned} f_{\mathbf{a}+\mathbf{c}}(\mathbf{x}) = 3x_1 + 7x_2 + 7(x_1 \oplus x_2) + 2x_3 + 2(x_1 \oplus x_2 \oplus x_3) + 4x_4 + 2(x_1 \oplus x_2 \oplus x_4) \\ + (x_3 \oplus x_4) + (x_1 \oplus x_3 \oplus x_4) + (x_2 \oplus x_3 \oplus x_4) + (x_1 \oplus x_2 \oplus x_3 \oplus x_4). \end{aligned}$$

The resulting circuit, optimized for T-depth, is shown below. Note that many different circuits with the same T-count are possible, for instance targeting CNOT-minimization instead of T-depth.



The circuit decoding reduces the T-count from 14 (or 8, as just phase-folding would achieve) to 7. Moreover, the number of T gates is equal to the distance from $\mathrm{Res}_2(\mathbf{a})$ to the decoded word $\mathbf{w}$.

Remark 5.3.4. It is interesting to note that the minimal T-depth of the decoded phase polynomial $f_{\mathbf{a}+\mathbf{c}}$ without additional ancillas is 3, while the minimal ancilla-free T-depth of $f_{\mathbf{a}}$ is 2, even though the number of T gates is reduced. Hence reducing the number of Z_m gates may result in an *increase* in other cost functions of the circuit, even in cases where the increase can seem counterintuitive. We will return to this idea in the next chapter when we discuss CNOT-optimal synthesis.

5.3.3 Complexity of phase gate optimization

It was claimed in the beginning of this chapter that the problems of phase gate optimization and minimum-distance Reed-Muller decoding were polynomial-time equivalent. We now prove this fact, and discuss some implications of the equivalence. Note that the inputs to either problem are in general length 2^n vectors (the 2^n Fourier coefficients for optimal synthesis and length 2^n binary vector of Reed-Muller decoding), hence our reductions are *polynomial in* 2^n rather than n.

Theorem (5.1.5)**.** *Let $m = 2^k l$ where l is odd. Then the optimal synthesis problem for $\langle \text{CNOT}, X, Z_m \rangle$ with respect to Z_m-count, and minimum-distance decoding problem for $\mathcal{RM}(n-k-1,n)^*$ are polynomial-time equivalent – that is, there exist polynomial-time reductions from each to the other.*

Proof. For the forward reduction, observe that Algorithm 5.1 finds a Z_m-count optimal circuit in time polynomial in 2^n using an oracle for minimum-distance decoding in $\mathcal{RM}(n-k-1,n)^*$. In particular, the input $\text{Res}_2(\mathbf{a})$ to the Reed-Muller decoder has size polynomial in 2^n and can be computed in polynomial time. Likewise, given $\mathbf{w} \in \mathcal{RM}(n-k-1,n)^*$, $\mathbf{w}$ can be written as a degree at most $n-k-1$ polynomial in polynomial time (in 2^n) via Gaussian elimination. Since $\mathbf{w}$ has degree at most $n-k-1$, it is a sum of polynomially many terms, and hence $\mathbf{c}$ can be computed in polynomial time. Finally, CNOT-dihedral synthesis can be completed in time polynomial in 2^n.

For the opposite reduction, given a vector $\mathbf{w} \in \mathbb{F}_2^{2^n}$ to be decoded synthesize a circuit C with minimal Z_m-count implementing the transformation

$$|\mathbf{x}\rangle \mapsto e^{\frac{2\pi i}{m} f_{\mathbf{w}}(\mathbf{x})} |\mathbf{x}\rangle.$$

The Fourier coefficients $\mathbf{a}$ of C can then be computed in time polynomial in $n \cdot |C|$, which is polynomial in 2^n since any Z_m-gate minimal CNOT-dihedral circuit can be implemented in at most $O(n2^{n-1})$ gates by simply enumerating each parity $\chi_{\mathbf{y}}(\mathbf{x})$. As a consequence of Theorem 5.2.8, $\text{Res}_2(\mathbf{a}) + \mathbf{w} \in \mathcal{RM}(n-k-1,n)^*$ is a minimum-distance decoding of $\mathbf{w}$, and moreover can be computed in time polynomial in 2^n as required. □

This equivalence lends evidence to the difficulty of $Z_{2^k l}$-count optimization when $k \geq 3$, even in the restricted setting we consider here of CNOT-dihedral circuits. In particular, as a function of n, any *sub-exponential* algorithm for exact optimization of T-count over n-qubit $\{\text{CNOT}, T\}$ circuits induces a *linear-time* (in 2^n) minimum-distance decoding algorithm for $\mathcal{RM}(n-4,n)^*$. By contrast, it appears very unlikely that even a polynomial-time

algorithm for minimum-distance decoding the order $n-4$ punctured Reed-Muller code exists; no minimum-distance decoding algorithms in time polynomial in 2^n are currently known for arbitrary order length 2^n binary Reed-Muller codes. While some particular orders of Reed-Muller codes have efficient decoders, e.g., order 1, it was shown by Seroussi and Lempel [SL83] that minimum-distance decoding for $\mathcal{RM}(n-4,n)^*$ is equivalent to the problem of finding a minimal decomposition of a symmetric 3-tensor into symmetric tryads (rank 1 3-tensors), a problem that is widely believed to be computationally hard [SL83].

On the other hand, Theorem 5.1.5 also shows that for $k < 3$, the problem is efficiently solvable. In particular, Seroussi and Lempel gave an efficient algorithm for maximum likelihood decoding of $\mathcal{RM}(n-3,n)^*$ [SL83], and moreover efficient decoders exist for $k < 2$, corresponding to the Hamming code when $k = 2$ and trivial codes when $k = 0$ or 1. Hence we have the following corollary.

Corollary 5.3.5. *The optimal synthesis problem for $\langle \mathrm{CNOT}, X, Z_m \rangle$ with respect to Z_m-count is efficiently solvable whenever $k < 3$ is the highest power of 2 dividing m.*

5.4 Generators of C_m^n

We now prove Lemma 5.2.5. Our proof proceeds by considering the two cases – where the phase polynomial is 0 mod 2^k and 0 mod l – separately, as per the proposition below. In particular, we show that only the trivial phase polynomials are zero mod l for any odd l, while the phase polynomials which are 0 mod 2^k correspond to polynomials of *order* at most $n-k-1$.

Proposition 5.4.1. *Let $m = 2^k l$ for some l coprime with 2. Then $C_m^n = C_{2^k}^n \cap C_l^n$.*

That the above proposition gives a set of generators of C_m^n given generators of $C_{2^k}^n$ and C_l^n is a trivial consequence of the particular presentations we derive below.

5.4.1 Rotations of odd order

We begin by proving that for *odd order* phase rotations, the zero-everywhere code is the trivial code. In particular, there are no non-trivial phase polynomials such that $f_{\mathbf{a}}(\mathbf{x}) = 0$ mod l for all $\mathbf{x} \in \mathbb{F}_2^n$ when l is odd.

Lemma 5.4.2. *Let l be an odd integer. Then*

$$C_l^n = \langle \mathbf{0} \rangle .$$

That is, C_l^n is the trivial subgroup of $\mathbb{Z}_l^{2^n}$.

To prove Lemma 5.4.2, we recall the *multilinear representation* of a phase polynomial. In particular, a function $f : \mathbb{F}_2^n \to \mathbb{Z}_l$ can be written as a multilinear polynomial

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \cdot \mathbf{x}^{\mathbf{y}}.$$

The fact, originally informally noted in Chapter 3, that this representation is *unique* is a consequence of the following lemma, namely that the evaluation vectors of monomials forms a basis for the module $\mathbb{Z}_l^{2^n}$.

Lemma 5.4.3. *The set*[2] $\{\mathbf{X}^{\mathbf{y}} \in \mathbb{F}_2^{2^n} \subset \mathbb{Z}_k^{2^n} \mid \mathbf{y} \in \mathbb{F}_2^n\}$ *is a basis for $\mathbb{Z}_k^{2^n}$ for any k.*

Proof. Informally, given that the monomial evaluation vectors contain a pivot for every entry of $\mathbb{F}_2^{2^n}$ (see, e.g., Figure 5.2), any element of $\mathbb{Z}_k^{2^n}$ can be written as a sum of monomial evaluation vectors via Gaussian elimination.

More formally, the set $\{\mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n\}$ has cardinality 2^n and moreover is linearly independent. Since the (module) rank of $\mathbb{Z}_k^{2^n}$ is 2^n, it is a basis of $\mathbb{Z}_k^{2^n}$. □

The fact that uniqueness follows from Lemma 5.4.3 is a trivial consequence of the fact that any such function can be uniquely represented by its vector of evaluations or truth table – i.e. $f \mapsto \mathbf{f}$ is an isomorphism. The uniqueness of the multilinear representation moreover establishes that the trivial polynomial is the *only* multilinear representation of the zero-everywhere function $f(\mathbf{x}) = 0$.

The final piece needed to establish Lemma 5.4.2 is that a multilinear polynomial over $\mathbb{Z}_l$ for odd l can be rewritten as a Fourier expansion. For this fact, recall that

$$2xy = x + y - (x \oplus y)$$

for any $x, y \in \mathbb{F}_2$. Since 2 is coprime with l it has a multiplicative inverse 2^{-1} in $\mathbb{Z}_l$, hence we can rewrite this identity as

$$xy = 2^{-1}x + 2^{-1}y - 2^{-1}(x \oplus y) \mod l.$$

[2]Since we really work "up to global phase", to be precise the set of monomials excluding the constant monomial is a basis of the isomorphic group $\mathbb{Z}_k^{2^n-1}$.

The equation above can be used to recursively rewrite a given monomial $\mathbf{x}^{\mathbf{y}} = x_{i_1}x_{i_2}\cdots x_{i_{|\mathbf{y}|}}$ over the parity basis – i.e. as a Fourier expansion:

$$\begin{aligned}(x_{i_1}x_{i_2})x_{i_3}\cdots x_{i_{|\mathbf{y}|}} &= (2^{-1}x_{i_1} + 2^{-1}x_{i_2} - 2^{-1}(x_{i_1} \oplus x_{i_2}))x_{i_3}\cdots x_{i_{|\mathbf{y}|}} \mod l \\ &= 2^{-1}x_{i_1}x_{i_3}\cdots x_{i_{|\mathbf{y}|}} + 2^{-1}x_{i_2}x_{i_3}\cdots x_{i_{|\mathbf{y}|}} - 2^{-1}(x_{i_1} \oplus x_{i_2})x_{i_3}\cdots x_{i_{|\mathbf{y}|}} \mod l,\end{aligned}$$

where each term in the second line has degree $|\mathbf{y}| - 1$, hence the recursion is terminating. It follows that this representation is also unique, as the multilinear representation is unique and both representations have 2^n degrees of freedom.

Proposition 5.4.4. *Let l be an odd integer. Then for any function $f : \mathbb{F}_2^n \to \mathbb{Z}_l$, there is a unique $\mathbf{a} \in \mathbb{Z}_l^{2^n}$ such that for all $\mathbf{x} \in \mathbb{F}_2^n$,*

$$f(\mathbf{x}) = f_{\mathbf{a}}(\mathbf{x}).$$

Proof. As per the above, every such function has a unique multilinear representation, and hence a unique Fourier expansion, over $\mathbb{Z}_l$. Taking the tuple $\mathbf{a}$ as the coefficients of the Fourier expansion of f we have $f(\mathbf{x}) = f_{\mathbf{a}}(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{F}_2^n$. □

We can now prove Lemma 5.4.2.

Proof of Lemma 5.4.2. Recall that $C_l^n \subseteq \mathbb{Z}_l^{2^k}$ such that for any $\mathbf{c} \in C_l^n$, $f_{\mathbf{c}}(\mathbf{x}) = 0 \mod l$ for all $\mathbf{x} \in \mathbb{F}_2^n$. Since $f_{\mathbf{0}}(\mathbf{x}) = 0 \mod l$ for all $\mathbf{x}$, by Proposition 5.4.4 this is the unique Fourier expansion of the zero-everywhere function (mod l), hence

$$C_l^n = \{\mathbf{0}\} = \langle \mathbf{0} \rangle .$$

□

Remark 5.4.5. From the perspective of phase gate optimization, Lemma 5.4.2 asserts that for any Z_m gate where m is odd, *phase-folding achieves the optimal Z_m-count for* CNOT*-dihedral circuits of order* $2m$. In particular, phase-folding can be seen to give an optimal circuit implementing a *particular decomposition* of the phase polynomial; for odd m, there are no equivalent phase polynomials except up to equivalences of the phase gate angles mod m, so the circuit obtained is exactly optimal.

5.4.2 Rotations of order 2^k

We now turn our attention to the more interesting case of *even-power orders*. In particular, we prove the following lemma:

Lemma 5.4.6. *Let k be some integer. Then*

$$C_{2^k}^n = \left\langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \right\rangle.$$

As a roadmap, we first show that the tuple $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ can itself be written as the evaluations of a polynomial in n variables. We then derive a formula for the value of $f_{\mathbf{a}}(\mathbf{x})$ as a function of the polynomial form of $\mathbf{a}$, which we then use to show that every zero-everwhere polynomial must arise as the evaluations of an order $n-k-1$ polynomial, where the order is (roughly) the maximum difference between the degree of a monomial and the highest power of 2 dividing its coefficient.

The monomial basis

Our proof relies on a connection between the binary evaluations of polynomials over $\mathbb{Z}_{2^k}$ and the $\mathbb{Z}$-module $\mathbb{Z}_{2^k}^{2^n}$. In particular, consider the set of degree at most $n-1$ monomial (Boolean) evaluation vectors

$$\{\mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| < n\}.$$

This set of vectors, under the natural inclusion of $\mathbb{F}_2$ in $\mathbb{Z}_{2^k}$, generates $\mathbb{Z}_{2^k}^{2^n} / \langle (1, 0, \ldots, 0) \rangle$. Hence every tuple of Fourier coefficients $\mathbf{a} \in \mathbb{Z}_{2^k}^{2^n}$ can be written *up to global phase* as the evaluation vector of a degree less than n polynomial function in $\mathbb{Z}_{2^k}[X]$.

Lemma 5.4.7.

$$\mathbb{Z}_{2^k}^{2^n} / \langle (1, 0, \ldots, 0) \rangle = \langle \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| < n \rangle.$$

Proof. Recall that by Lemma 5.4.3, the set of *all* monomials $\{\mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n\}$ is a basis for $\mathbb{Z}_{2^k}^{2^n}$. It therefore suffices to prove that $\mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n$ is in the span of $\{\mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| < n\}$, up to the first coordinate.

It may be observed that over $\mathbb{F}_2$, the set of *all* monomials up to constants is linearly dependent up to the first coordinate:

$$\bigoplus_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} = (1, 0, \ldots, 0)$$

Since for every non-constant monomial $X^{\mathbf{y}}$, exactly half of the valuations to $X = X_1, \ldots, X_n$ satisfies $X^{\mathbf{y}} = 1$. Further, we see that

$$\bigoplus_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} = \mathrm{Res}_2 \left(\sum_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} \right) = (1, 0, \ldots, 0)$$

and so $\sum_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} = \mathbf{a}$ for some $\mathbf{a} \in \mathbb{Z}_m^{2^n}$ such that $\mathrm{Res}_2(\mathbf{a}) = (1, 0, \ldots, 0)$. Since $\mathbf{a}$ can be written up to the first coordinate over $\{\mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| > 0\}$ we see

$$\sum_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} = \mathbf{a} = \sum_{\mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| > 0} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}.$$

Rearranging we get

$$\alpha_{\mathbf{1}} \mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n = \sum_{\mathbf{y} \in \mathbb{F}_2^n, 0 < |\mathbf{y}| < n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}}$$

Now suppose $\alpha_{\mathbf{1}} = 1 \mod 2$. Then

$$\mathrm{Res}_2 \left(\alpha_{\mathbf{1}} \mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n \right) = \mathrm{Res}_2 \left(\sum_{\mathbf{y} \in \mathbb{F}_2^n, 0 < |\mathbf{y}| < n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n} \mathbf{X}^{\mathbf{y}} \right)$$
$$\mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n = \bigoplus_{\mathbf{y} \in \mathbb{F}_2^n, 0 < |\mathbf{y}| < n} \mathrm{Res}_2(\alpha_{\mathbf{y}}) \mathbf{X}^{\mathbf{y}},$$

which is a contradiction since the non-constant monomials are linearly independent over $\mathbb{F}_2$.

Thus $\alpha_{\mathbf{1}} = 0 \mod 2$ and in particular $(1 + \alpha_{\mathbf{1}})^{-1} \in \mathbb{Z}_{2^k}$, hence we can write

$$(1 + \alpha_{\mathbf{1}}) \mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n = \sum_{\mathbf{y} \in \mathbb{F}_2^n, 0 < |\mathbf{y}| < n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| < n} \mathbf{X}^{\mathbf{y}}$$
$$\mathbf{X}_1 \mathbf{X}_2 \cdots \mathbf{X}_n = (1 + \alpha_{\mathbf{1}})^{-1} \left(\sum_{\mathbf{y} \in \mathbb{F}_2^n, 0 < |\mathbf{y}| < n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| < n} \mathbf{X}^{\mathbf{y}} \right).$$

as required. □

Lemma 5.4.7 tells us that any element $\mathbf{a}$ of $\mathbb{Z}_{2^k}^{2^n} / \langle (1, 0, \ldots, 0) \rangle$ is the vector of evaluations for some pseudo-Boolean polynomial function $f : \mathbb{F}_2^n \to \mathbb{Z}_{2^k}$ where $f(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} \mathbf{x}^{\mathbf{y}}$ and $\deg(f) < n$. In particular,

$$\mathbf{a} = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}.$$

Evaluating $P_{\mathbf{a}}$

The next step in our proof is to give an analytic formula for the value of a phase function $f_{\mathbf{a}}$ applied to a vector $\mathbf{x} \in \mathbb{F}_2^n$ as a function of the degree of the polynomial form of $\mathbf{a}$. Specifically, we show that $f_{\mathbf{a}}(\mathbf{x})$ is equal to a linear combination of the Hamming weights of certain Boolean polynomials arising from the multiplication of a monomial with a linear polynomial.

Consider the value $f_{\mathbf{a}}$ at $\mathbf{x} \in \mathbb{F}_2^n$:

$$f_{\mathbf{a}}(\mathbf{x}) = a_{\mathbf{0}} + \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \chi_{\mathbf{y}}(\mathbf{x})$$

We can rewrite the above equation as an inner product on $\mathbb{Z}_m$, since the value $\chi_{\mathbf{y}}(\mathbf{x})$ is the $\mathbf{y}$th component of the evaluation vector $\chi_{\mathbf{x}}(\mathbf{X}) = x_1\mathbf{X_1} \oplus \cdots \oplus x_n\mathbf{X_n}$.

Example 5.4.8. Consider the evaluation vectors over 3 Boolean variables:

$$\mathbf{X_1} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{X_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{X_3} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Computing the sum of the evaluation vectors multiplied by some $\mathbf{x} = x_1x_2x_3$ we see that

$$x_1\mathbf{X_1} \oplus x_2\mathbf{X_2} \oplus x_2\mathbf{X_3} = \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_1 \oplus x_2 \\ x_3 \\ x_1 \oplus x_3 \\ x_2 \oplus x_3 \\ x_1 \oplus x_2 \oplus x_3 \end{bmatrix}.$$

Now it can be observed that the ith entry of $x_1\mathbf{X_1} \oplus x_2\mathbf{X_2} \oplus x_2\mathbf{X_3}$ is exactly $\chi_i(\mathbf{x})$ – for instance,

$$\chi_{101}(\mathbf{x}) = x_1 \oplus x_3 = (x_1\mathbf{X_1} \oplus x_2\mathbf{X_2} \oplus x_2\mathbf{X_3})_{101}.$$

Formally, we define $\langle \mathbf{a}, \mathbf{b} \rangle$ for $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_m^{2^n}$ as $\sum_{i=1}^{2^n} a_i b_i$. Note that

$$\langle \mathbf{a} + \mathbf{b}, \mathbf{c} \rangle = \langle \mathbf{a}, \mathbf{c} \rangle + \langle \mathbf{b}, \mathbf{c} \rangle$$

for any $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{Z}_m^{2^n}$ since the inner product is linear in either argument. Using this observation, we give an explicit formula for $f_{\mathbf{a}}(\mathbf{x})$ up to global phase as a function of the basis vectors appearing in $\mathbf{a}$:

Lemma 5.4.9. *Let* $\mathbf{a} = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}} \in \mathbb{Z}_m^{2^n}$. *Then*

$$f_{\mathbf{a}}(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} |\mathbf{X}^{\mathbf{y}}(x_1 \mathbf{X_1} \oplus \cdots \oplus x_n \mathbf{X_n})|.$$

Proof. By direct calculation.

$$\begin{aligned} f_{\mathbf{a}}(\mathbf{x}) &= \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \chi_{\mathbf{y}}(\mathbf{x}) \\ &= \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} (x_1 \mathbf{X_1} \oplus x_2 \mathbf{X_2} \oplus x_2 \mathbf{X_3})_{\mathbf{y}} \\ &= \langle \mathbf{a}, x_1 \mathbf{X_1} \oplus x_2 \mathbf{X_2} \oplus x_2 \mathbf{X_3} \rangle \\ &= \sum_{\mathbf{y} \in \mathbb{F}_2^n} \langle \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}, x_1 \mathbf{X_1} \oplus x_2 \mathbf{X_2} \oplus x_2 \mathbf{X_3} \rangle \\ &= \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} |\mathbf{X}^{\mathbf{y}}(x_1 \mathbf{X_1} \oplus \cdots \oplus x_n \mathbf{X_n})|. \end{aligned}$$

□

As the Hamming weight of the evaluation vector of a Boolean function is simply the number of satisfying solutions to that function, the value of $|\mathbf{X}^{\mathbf{y}}(x_1 \mathbf{X_1} \oplus \cdots \oplus x_n \mathbf{X_n})|$ in Lemma 5.4.9 above may be restated as the number of solutions to the equation $X^{\mathbf{y}}(x_1 X_1 \oplus \cdots \oplus x_n X_n) = 1$. While generally the number of solutions to a Boolean function is non-trivial, in this particular case a simple analytic formula of the polynomial's degree suffices.

Lemma 5.4.10. *For any* $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$,

$$|\mathbf{X}^{\mathbf{y}}(x_1 \mathbf{X_1} \oplus \cdots \oplus x_n \mathbf{X_n})| = 2^{n - \deg(X^{\mathbf{y}}(x_1 X_1 \oplus \cdots \oplus x_n X_n))}$$

if $X^{\mathbf{y}}(x_1 X_1 \oplus \cdots \oplus x_n X_n) \neq 0$, *or* 0 *otherwise.*

Proof. Clearly if $X^{\mathbf{y}}(x_1X_1 \oplus \cdots \oplus x_nX_n) = 0$, then $|\mathbf{X}^{\mathbf{y}}(x_1\mathbf{X_1} \oplus \cdots \oplus x_n\mathbf{X_n})| = 0$ as required.

Suppose instead that $X^{\mathbf{y}}(x_1X_1 \oplus \cdots \oplus x_nX_n) \neq 0$. Since $x_1X_1 \oplus \cdots \oplus x_nX_n$ and $X^{\mathbf{y}}$ has degree $|\mathbf{y}|$, their multiplication has degree either $|\mathbf{y}|$ or $|\mathbf{y}| + 1$. We consider the two cases separately.

Consider the degree $|\mathbf{y}|$ case first. Clearly $\mathbf{x} \subseteq \mathbf{y}$ – that is, every variable X_i in the linear combination $x_1X_1 \oplus \cdots \oplus x_nX_n$ is already in $X^{\mathbf{y}}$, hence the degree remains unchanged. Then

$$X^{\mathbf{y}}(x_1X_1 \oplus \cdots \oplus x_nX_n) = |\mathbf{x}|X^{\mathbf{y}} = \begin{cases} 0 & \text{if } |\mathbf{x}| = 0 \mod 2 \\ X^{\mathbf{y}} & \text{otherwise} \end{cases}.$$

Since $X^{\mathbf{y}}(x_1X_1 \oplus \cdots \oplus x_nX_n) \neq 0$, it must be the case that $|\mathbf{x}| = 1 \mod 2$. Moreover, the equation $X^{\mathbf{y}} = 1$ has exactly $2^{n-\deg(X^{\mathbf{y}})}$ solutions, corresponding to $X_i = 1$ if $y_i = 1$.

Now consider the degree $|\mathbf{y}| + 1$ case. We know $\mathbf{x} \nsubseteq \mathbf{y}$. Without loss of generality we can assume $\mathbf{x} \cap \mathbf{y} = \mathbf{0}$ – that is, there are no variables that appear in both $X^{\mathbf{y}}$ and $x_1X_1 \oplus \cdots \oplus x_nX_n$ – as any common variables can be absorbed into $X^{\mathbf{y}}$.

Recall that $x_1X_1 \oplus \cdots \oplus x_nX_n = 1$ for exactly half of the values of all X_i such that $x_i = 1$. Since $X^{\mathbf{y}} = 1$ for the $2^{n-\deg(X^{\mathbf{y}})}$ valuations on $X_1 \ldots, X_n$ where $X_i = 1$ whenever $y_i = 1$ and $y_i = 1$ implies $x_i = 0$, exactly half of those solutions – of which there are $2^{n-\deg(X^{\mathbf{y}})-1}$ – satisfy $x_1X_1 \oplus \cdots \oplus x_nX_n = 1$. Hence

$$|\mathbf{X}^{\mathbf{y}}(x_1\mathbf{X_1} \oplus \cdots \oplus x_n\mathbf{X_n})| = 2^{n-\deg(X^{\mathbf{y}}(x_1X_1 \oplus \cdots \oplus x_nX_n))}$$

as required. □

In general, it is not the case that the number of solutions to $f(\mathbf{x}) = 1$ is $2^{n-\deg(f)}$ for an n-variate Boolean polynomial function f. In particular, consider $f(\mathbf{x}) = 1 \oplus x_1x_2 \cdots x_i$. Since $x_1x_2 \cdots x_i = 1$ has 2^{n-i} solutions, the number of solutions to $f(\mathbf{x}) = 1$ is

$$2^n - 2^{n-i} \neq 2^{n-\deg(f)}.$$

An explicit set of generators

From Lemma 5.4.10 it's immediate that if $\mathbf{a} \in \mathbb{Z}_{2^k}^{2^n}$ may be written over the monomial basis with degree at most $n - k - 1$, then $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^k$ for any $\mathbf{x}$ and so $\mathbf{a} \in C_{2^k}^n$. However, it may be the case that $\mathbf{a}$ is a sum of monomials with degree greater than $n - k - 1$ and yet are still in $C_{2^k}^n$.

For instance, consider $n = 4, k = 3$ and let $\mathbf{a} = 2\mathbf{X}^{\mathbf{y}}$ where $|\mathbf{y}| = 1$. Now for any $\mathbf{x} \in \mathbb{F}_2^4$,

$$\begin{aligned}\log_2 f_{\mathbf{a}}(\mathbf{x}) &\geq 1 + n - |\mathbf{y}| - 1 \\ &= 3,\end{aligned}$$

hence $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^k$, and yet $\mathbf{a}$ has degree $1 > n - k - 1$. In particular, with regards to evaluation on $\mathbf{x} \in \mathbb{F}_2^n$, the term $2\mathbf{X}^{\mathbf{y}}$ *acts as if it were a term of degree* $n - 4$.

We use this intuition to define the *order* of a polynomial in $\mathbb{Z}_m[X]$. Specifically, we say the order of a term $\alpha X^{\mathbf{y}}$, denoted $\mathsf{ord}\,(\alpha X^{\mathbf{y}})$, is the degree of $X^{\mathbf{y}}$ minus the largest k such that $2^k \mid \alpha$. Analytically,

$$\mathsf{ord}\,(\alpha X^{\mathbf{y}}) = |\mathbf{y}| - \lfloor \log_2 \alpha \rfloor.$$

We extend order to polynomials, and moreover evaluation vectors of polynomials, by taking the order of a polynomial as the highest order of each term.

The next lemma, and the main lemma of the section, now establishes that any $\mathbf{a} \in \mathbb{Z}_{2^k}^{2^n}$ with multilinear form of order greater than $n - k - 1$ is not zero-everywhere mod 2^k.

Lemma 5.4.11. *Let* $\mathbf{a} \in \mathbb{Z}_{2^k}^{2^n}$ *have order* $m > n - k - 1$. *Then there exists* $\mathbf{x} \in \mathbb{F}_2^n$ *such that*

$$f_{\mathbf{a}}(\mathbf{x}) \neq 0 \mod 2^k$$

Proof. Suppose to the contrary that $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^k$ for all $\mathbf{x} \in \mathbb{F}_2^n$ and let

$$\mathbf{a} = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}.$$

Since $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^k$ for all $\mathbf{x} \in \mathbb{F}_2^n$ we must also have $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^{n-m-1}$, as $n - m - 1 < n - (n - k - 1) - 1 = k$. Recalling that if $\mathsf{ord}\,(\alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}) = i$, by Lemmas 5.4.9 and 5.4.10

$$f_{\alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}}(\mathbf{x}) = 2^{n-i} \text{ or } 2^{n-i-1}.$$

The latter case occurs exactly when $\mathbf{x} \not\subseteq \mathbf{y}$, and hence we see that $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^{n-m-1}$ implies there are *evenly many terms* $\alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}$ *of maximum order* m *such that* $\mathbf{x} \not\subseteq \mathbf{y}$. Alternatively, for any $\mathbf{x} \in \mathbb{F}_2^n$

$$f_{\mathbf{a}}(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} f_{\alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}}(\mathbf{x}) = \sum_{\substack{\mathbf{y} \in \mathbb{F}_2^n, \mathbf{x} \not\subseteq \mathbf{y} \\ \mathsf{ord}(\alpha_{\mathbf{y}} \mathbf{X}^{\mathbf{y}}) = m}} 2^{n-m-1} \mod 2^{n-m},$$

since every other term evaluates either to 2^m (i.e. if it has order m by $\mathbf{x} \subseteq \mathbf{y}$), or to 2^{n-i} or 2^{n-i-1} where $i < m$. Moreover, we know at least one such term exists, since by Lemma 5.4.7 $\mathbf{a}$ can be written up to global phase with degree at most $n-1$

Our contradiction arises from the fact that there necessarily exists $\mathbf{x} \in \mathbb{F}_2^n$ such that an odd number of terms $\alpha_{\mathbf{y}}\mathbf{X}^{\mathbf{y}}$ with order m such that $\mathbf{x} \not\subseteq \mathbf{y}$. In particular, let

$$S_i = \{\mathbf{y} \in \mathbb{F}_2^n \mid \mathsf{ord}\,(\alpha_{\mathbf{y}}\mathbf{X}^{\mathbf{y}}) = m, y_i = 0\},$$

that is S_i is the set of terms of maximum order which do not contain X_i. Given some $\mathbf{x} \in \mathbb{F}_2^n$, $\bigcup_{i|x_i=1} S_i$ gives the set of terms of maximum order such that $\mathbf{x} \not\subseteq \mathbf{y}$, and hence since $f_{\mathbf{a}}(\mathbf{x}) = 0 \mod 2^{n-m-1}$, it follows that

$$|\cup_{i|x_i=1} S_i| = 0 \mod 2.$$

Now take $\mathbf{y}' \in \mathbb{F}_2^n$ such that $\mathsf{ord}\left(\alpha_{\mathbf{y}'}\mathbf{X}^{\mathbf{y}'}\right) = m$, minimizing $|\mathbf{y}'|$ – that is, $\mathbf{y}'$ is a term of maximum order but with minimum degree. Since $\mathbf{y}'$ has minimal weight, for every other $\mathbf{y}$ such that $\mathsf{ord}\,(\alpha_{\mathbf{y}}\mathbf{X}^{\mathbf{y}}) = m$, there necessarily exists i such that $y_i = 1$ but $y'_i = 0$. Hence

$$\cap_{i|y'_i=0} S_i = \{\mathbf{y}'\}.$$

By the principle of inclusion-exclusion, the cardinality of this set can be written as a sum of cardinalities of unions of S_i – in particular,

$$\begin{aligned}
1 = |\cap_{i|y'_i=0} S_i| &= |\overline{\cup_{i|y'_i=0} \overline{S_i}}| \\
&= 2^n - |\cup_{i|y'_i=0} \overline{S_i}| \\
&= 2^n - \sum_{k=1}^{|\overline{\mathbf{y}}|} (-1)^{k+1} \left(\sum_{i_1,\ldots,i_k} |\overline{S_{i_1}} \cap \cdots \cap \overline{S_{i_k}}| \right) \\
&= 2^n - \sum_{k=1}^{|\overline{\mathbf{y}}|} (-1)^{k+1} \left(\sum_{i_1,\ldots,i_k} 2^n - |S_{i_1} \cup \cdots \cup S_{i_k}| \right)
\end{aligned}$$

However, since $|\cup_{i|x_i=1} S_i| = 0 \mod 2$ for any $\mathbf{x}$, we have $1 = 0 \mod 2$, hence we derive our contradiction.

□

Lemma 5.4.11 shows that $C_{2^k}^n$ is a subset of the evaluation vectors of order at most $n-k-1$ polynomials, and hence sums of the generators in Lemma 5.4.6. Together with the fact that every generator gives a zero-everywhere phase polynomial, we can finally prove the result.

Lemma (5.4.6)**.** *Let k be some integer. Then* $C^n_{2^k} = \langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \rangle$.

Proof. For the forward direction,

$$C^n_{2^k} \subseteq \left\langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \right\rangle,$$

suppose $\mathbf{c} \in C^n_{2^k}$. Then $f_{\mathbf{c}}(\mathbf{x}) = 0 \mod 2^k$ for all $\mathbf{x} \in \mathbb{F}_2^n$, so by Lemma 5.4.11, $\mathbf{c}$ has order at most $n - k - 1$ and can hence be written as a sum of the above generators.

Now consider some $\mathbf{c} = 2^i \mathbf{X}^{\mathbf{y}}$ where $|\mathbf{y}| - i \leq n - k - 1$. By Lemmas 5.4.9 and 5.4.10,

$$P_{\mathbf{c}}(\mathbf{x}) = 2^{i+n-|\mathbf{y}|}$$

for any $\mathbf{x} \in \mathbb{F}_2^n$. Since $i + n - |\mathbf{y}| \geq i + n - (n + i - k - 1) = k - 1$ we have $f_{\mathbf{c}}(\mathbf{x}) = 0 \mod 2^k$ so $\mathbf{c} \in C^n_{2^k}$. Moreover since $C^n_{2^k}$ is a group, we see that

$$\left\langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \right\rangle \subseteq C^n_{2^k}.$$

Hence $C^n_{2^k} = \langle 2^i \mathbf{X}^{\mathbf{y}} \mid \mathbf{y} \in \mathbb{F}_2^n, |\mathbf{y}| - i \leq n - k - 1 \rangle$. □

5.5 Experiments

To evaluate Algorithm 5.1 experimentally, it was implemented in the open source T-par[3] circuit optimization software, which implements a version of the phase-folding algorithm of the previous chapter. Algorithm 5.1 was used to optimize greedily chosen CNOT-dihedral sub-circuits both *before and after* phase-folding. Only the results for before phase folding are shown, as the results after phase folding gave no improvements. The experimental set up was identical to the previous chapter, with an additional timeout of 30 minutes due to the high degree of difficulty of decoding.

We implemented and tested the algorithm with two Reed-Muller decoders – an early majority logic decoder due to Reed [Ree54], and a modern recursive decoder due to Dumer [Dum04]. The former has complexity in $O(2^{2n})$ for an n-qubit circuit while the latter has a significantly lower complexity of $O(2^n)$. While both of these algorithms are exponential in the number of qubits n, we nonetheless obtain reasonable performance for large circuits by storing and operating directly on *run-length encoded* vectors, as most vectors we saw had only a few 0-1 or 1-0 alternations. In order to optimize these large circuits we chose relatively fast decoders over minimum-distance decoders.

[3] https://github.com/meamy/t-par

5.5.1 Evaluation

Table 5.1 reports the T-count of circuits optimized with both phase-folding alone, and with Algorithm 5.1 using either the majority logic or recursive decoder applied to $\{\text{CNOT}, X, T\}$ subcircuits. Instances where the algorithm failed to report a result within the timeout are identified with a dash.

On average, Algorithm 5.1 performed slightly better than just phase-folding with both the majority logic decoder and the recursive decoder. While the recursive decoder produced the best results in some cases, notably the Galois field multipliers, and failed less often, for many benchmarks it reported significantly *increased* T-counts compared to phase-folding. Majority logic decoding by comparison typically produced less T-reduction, though it consistently resulted in circuits with equal or lesser T-count than just phase-folding. Counter-intuitively this appears to result from the recursive decoder actually doing a *better* job optimizing T-count – after the recursive decoder performs significant rewrites on individual CNOT-dihedral subcircuits, less phase gate merging possible. On the other hand, phase-folding before re-synthesis gave no improvement in T-counts. This lends strong evidence to the fact that for *real-world* circuits, as opposed to random circuits, phase-folding alone is generally close to optimal.

5.6 Related work

Quantum fault-tolerance The relationship between Reed-Muller codes and T gates has previously been studied from the perspective of fault-tolerance. Previous works [KLZ96, ZCC11, AJO16] have shown that there exist Quantum Reed-Muller codes which admit transversal implementations of Z_{2^k} gates for any k, and likewise that no such codes exist for odd-order phase gates. Likewise, such results have given rise to magic-state distillation algorithms [BH12, CAB12, LC13] for all levels of the Clifford hierarchy – that is, for implementing Z_{2^k} gates via gate teleportation.

These results are directly related to our results, in that they arise from the same modular equations of phase polynomials. The key difference is that while those results establish *existence* theorems, so as to show that such error-correcting codes exist, we also establish *completeness* results, so that our methods can be shown to be exactly optimal.

Subsequent work Since this work was originally made public [AM16], a series of works [CH17b, CH17a, HC18] has used the basic connection we have described here to perform both

Table 5.1: T-count optimization results. n reports the number of qubits in the circuit. T-counts are recorded for the original circuit, after optimization just by phase-folding, and after optimization by Algorithm 5.1 with either the majority logic or recursive decoder.

Benchmark	n	T-count			
		original	phase-folding	majority	recursive
Grover_5	9	140	52	52	52
Mod 5_4	5	28	16	16	16
VBE-Adder_3	10	70	24	24	24
CSLA-MUX_3	15	70	62	62	58
CSUM-MUX_9	30	196	140	84	76
QCLA-Com_7	24	203	95	94	153
QCLA-Mod_7	26	413	249	238	299
QCLA-Adder_{10}	36	238	162	–	188
Adder_8	24	399	215	213	249
RC-Adder_6	14	77	63	47	47
Mod-Red_{21}	11	119	73	73	73
Mod-Mult_{55}	9	49	37	35	35
Mod-Adder_{1024}	28	1,995	1,011	1,011	1,011
$\text{Mod-Adder}_{1048576}$	58	16,660	7,340	–	–
Cycle 17_3	35	4,739	1,945	1,944	1,982
GF(2^4)-Mult	12	112	68	68	68
GF(2^5)-Mult	15	175	111	111	101
GF(2^6)-Mult	18	252	150	150	144
GF(2^7)-Mult	21	343	217	217	208
GF(2^8)-Mult	24	448	264	264	237
GF(2^9)-Mult	27	567	351	–	301
GF(2^{10})-Mult	30	700	410	–	410
GF(2^{16})-Mult	48	1,792	1,040	–	–
GF(2^{32})-Mult	96	7,168	4,128	–	–
GF(2^{64})-Mult	192	28,672	16,448	–	–
GF(2^{128})-Mult	384	114,688	65,664	–	–
GF(2^{256})-Mult	768	458,752	262,400	–	–
Ham_{15} (low)	17	161	97	97	97
Ham_{15} (med)	17	574	230	230	230
Ham_{15} (high)	20	2,457	1,019	1,019	1,019
HWB_6	7	105	71	75	75
HWB_8	12	5,887	3,551	3,531	3,531
HWB_{10}	16	26,579	15,921	–	15,921
HWB_{12}	20	159,341	85,897	–	–
QFT_4	5	69	67	67	67
$\Lambda_3(X)$	5	28	16	16	16
$\Lambda_3(X)$ (Barenco)	5	21	15	15	15
$\Lambda_4(X)$	7	56	28	28	28
$\Lambda_4(X)$ (Barenco)	7	35	23	23	23
$\Lambda_5(X)$	9	84	40	40	40
$\Lambda_5(X)$ (Barenco)	9	49	31	31	31
$\Lambda_{10}(X)$	19	224	100	100	100
$\Lambda_{10}(X)$ (Barenco)	19	119	71	71	71

state distillation of more complicated – specifically, CNOT-dihedral – states, as well as to perform T-count optimization. Their methods used the connection between $\mathcal{RM}(n-4,n)^*$ decoding and minimal decomposition of symmetric 3-tensors to develop new decoding, and hence optimization algorithms.

In [HC18], Heyfron and Campbell show that on random CNOT-dihedral circuits, their decoders scale according to the upper bound of $O(2^n)$ given in this chapter. To optimize general Clifford+T circuits, rather than apply their decoders to CNOT-dihedral sub-circuits, they move all Hadamard gates to the beginning and end of the circuit at the cost of 1 ancilla per Hadamard, giving a single CNOT-dihedral circuit conjugated by Hadamard gates. By doing so they are able to directly optimize T-count over the entire circuit at once, giving an additional 20% improvement over phase-folding alone. It remains however unclear how much of their optimization is due to their decoder, and how much is a result of this gate cost vs. space trade-off.

Chapter 6

CNOT-count optimization

While the phase gate count, and more specifically R_Z gate count, is particularly important for fault-tolerant quantum computing, in most physical implementations of quantum computing the CNOT gate is the most expensive. Moreover, it forms the backbone of most discrete quantum circuits, as it is typically the only entangling operation and is hence used judiciously in effectively any practically useful quantum circuit. Even in physical implementations where gates commonly have tunable parameters many use the CNOT gate, or a CNOT gate up to single qubit rotations, as the two qubit entangling gate (e.g. [DLF+16]).

In this chapter, we again consider the problem of optimally synthesizing a CNOT-dihedral circuit given a sum-over-paths action

$$|\mathbf{x}\rangle \mapsto e^{2\pi i f(\mathbf{x})}|A\mathbf{x} + \mathbf{b}\rangle$$

where $f : \mathbb{F}_2^n \to \mathbb{R}$ and $A_\mathbf{b} \in \mathrm{GA}(n, \mathbb{F}_2)$. However, rather than from the perspective of R_Z-count, we target CNOT-count as our cost function – the synthesis method we describe here can moreover be combined with the R_Z-count optimization of Chapter 5. As the particular angle of rotations does not matter for the CNOT optimizations we consider here, we return to the more general CNOT-dihedral group of arbitrary order – that is, circuits over $\{\mathrm{CNOT}, X, R_Z(\theta) \mid \theta \in \mathbb{R}\}$.

We introduce the notion of a *parity network* to characterize the CNOT-minimal synthesis problem over CNOT-dihedral circuits. Informally, a parity network for a set $S \subseteq \mathbb{F}_2^n$ is a CNOT circuit in which the parity $\chi_\mathbf{y}(\mathbf{x})$ of the circuit's input state $|\mathbf{x}\rangle$ appears in the annotated circuit for any $\mathbf{y} \in S$. The intuition is that a parity network for $\mathrm{supp}(\widehat{f})$ suffices to implement the phase rotation $e^{2\pi i f(\mathbf{x})}$.

Using this characterization, we prove that synthesizing a CNOT-optimal circuit over $\{\text{CNOT}, X, R_Z\}$ is at least as hard as computing a minimal parity network. We then show that the minimal parity network problem is **NP**-hard in two restricted cases: when all CNOT gates are restricted to the same target bit, and when the m primary inputs are encoded in the state of $n > m$ qubits. The former case provides evidence for the hardness of computing minimal parity networks, while the latter case is useful when combined with phase-folding.

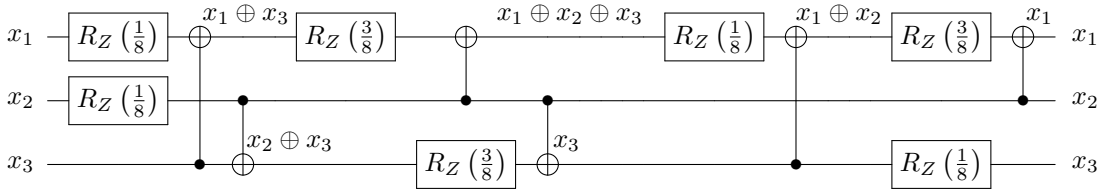
We further devise a new heuristic optimization algorithm for CNOT-dihedral circuits by synthesizing parity networks. The optimization algorithm is inspired by *Gray codes* [Gra53], which cycle through the set of n-bit strings using the exact minimal number of bit flips. Like Gray codes, our algorithm achieves the minimal number of CNOT gates when all 2^n parities are needed.

This work appears in [AAM18] and was presented at Theory of Quantum Computation, Communication & Cryptography 2018 (TQC'18). An implementation appears in Feynman.

6.1 Parity networks

A key observation from the last chapter is that *only parities which appear in the annotated circuit may have non-zero Fourier coefficient*. Otherwise, the parities may appear in any order, parities may appear in the circuit but not in the sum-over-paths form, or the same parity may appear multiple times. Multiple R_Z gates may also be applied with the same incoming parity throughout a circuit, in which case the Fourier coefficient is the sum of all the rotation angles and can be replaced with a single R_Z gate – this effect was previously used to optimize R_Z-counts in the phase-folding algorithm of Chapter 4.

The inverse of the above observation is that *a circuit in which every parity in* $\text{supp}(\widehat{f})$ *appears as an annotation can be modified to implement the phase rotation* $f(\mathbf{x}) = \widehat{f}(\mathbf{0}) + \sum_{\mathbf{y}\in\mathbb{F}_2^n} \widehat{f}(\mathbf{y})\chi_{\mathbf{y}}(\mathbf{x})$ *at no extra* CNOT *cost*. For example, recall again the CCZ circuit



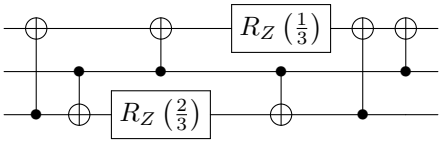
with sum-over-paths form $(f, A_{\mathbf{b}})$ where $A = I, \mathbf{b} = \mathbf{0}$ and

$$f(\mathbf{x}) = \frac{1}{8}\left(x_1 + x_2 + 3(x_1 \oplus x_3) + 3(x_2 \oplus x_3) + (x_1 \oplus x_2 \oplus x_3) + 3(x_1 \oplus x_2) + x_3\right).$$

The above circuit can be modified to give a new circuit with (non-equivalent) sum over paths form $(f', I_{\mathbf{0}})$ for

$$f'(\mathbf{x}) = \frac{2}{3}(x_2 \oplus x_3) + \frac{1}{3}(x_1 \oplus x_2 \oplus x_3)$$

with no additional CNOT gates simply by changing the parameters of the fourth and fifth R_Z gates to $\frac{2}{3}$ and $\frac{1}{3}$, respectively, and removing all other phase gates. The resulting circuit is shown below:



This motivates the definition of a *parity network* below as a CNOT circuit computing a set of parities, which can be used to implement phase rotations with Fourier expansions having support contained in that set.

Definition 6.1.1. A *parity network* for a set $S \subseteq \mathbb{F}_2^n$ is an n-qubit circuit C over CNOT gates where, for each $\mathbf{y} \in S$, the parity $\chi_{\mathbf{y}}(\mathbf{x})$ appears in the annotated circuit.

As all parity networks apply some overall linear transformation of the input, we say a parity network is *pointed at* $A \in \mathrm{GL}(n, \mathbb{F}_2)$ if the overall transformation is A, i.e.,

$$|\mathbf{x}\rangle \mapsto |A\mathbf{x}\rangle.$$

Note that as a parity network consists of just CNOT gates, this basis state transformation is in fact a linear rather than affine one. For convenience we refer to a parity network with the trivial transformation as an *identity parity network*. In the context of synthesizing parity networks, we use the term *pointed parity network* to refer to a parity network applying a specific linear transformation.

We can now formalize the above observations with the following proposition, stating that the problem of finding a minimal size (pointed) parity network is equivalent to finding a CNOT-minimal circuit having a *particular* sum-over-paths action – i.e. not up to equivalent phase functions mod 2π.

Proposition 6.1.2. *There exists a* CNOT*-dihedral circuit with sum-over-paths form* $(f, A_{\mathbf{b}})$ *and* t CNOT *gates if and only if there exists a parity network for* $\mathrm{supp}(\widehat{f})$ *pointed at* A *with* t CNOT *gates.*

Proof. First assume there exists a circuit C with t CNOT gates and sum-over-paths form $(f, A_{\mathbf{b}})$. Then the annotated circuit necessarily contains either the label $\chi_{\mathbf{y}}(\mathbf{x})$ or $1 \oplus \chi_{\mathbf{y}}(\mathbf{x})$ for every $\mathbf{y} \in \text{supp}(\widehat{f})$. As the X gates only affect affine factors, and R_Z gates don't affect labels, the circuit C' obtained by removing all X and R_Z gates contains as a label $\chi_{\mathbf{y}}(\mathbf{x})$ for every $\mathbf{y} \in \text{supp}(\widehat{f})$. Moreover, the overall linear transformation is $|\mathbf{x}\rangle \mapsto |A\mathbf{x}\rangle$, hence C' is a parity network for $\text{supp}(\widehat{f})$ pointed at A with t CNOT gates.

Now assume there exists a length t parity network C for $\text{supp}(\widehat{f})$ pointed at A. Then the circuit C' obtained by, for each $\mathbf{y} \in \text{supp}(\widehat{f})$ inserting $R_Z(\widehat{f}(\mathbf{y}))$ into C where $\chi_{\mathbf{y}}(\mathbf{x})$ appears as a label, has sum-over-paths action

$$|\mathbf{x}\rangle \mapsto e^{2\pi i \sum_{\mathbf{y}} \widehat{f}(\mathbf{y})\chi_{\mathbf{y}}(\mathbf{x})}|A\mathbf{x}\rangle$$

As the additional affine factor $|A\mathbf{x}\rangle \mapsto |A\mathbf{x} + \mathbf{b}\rangle$ is implementable with just X gates – i.e. $X^{\mathbf{b}} = \bigotimes X_i^{b_i}$ – and the remaining global phase $e^{2\pi i \widehat{f}(\mathbf{0})}$ can be implemented with $R_Z(\widehat{f}(\mathbf{0}))XR_Z(\widehat{f}(\mathbf{0}))X$, there exists a CNOT-dihedral circuit with sum-over-paths form $(f, A_{\mathbf{b}})$ and t CNOT gates. □

6.1.1 From CNOT–minimization to parity network synthesis

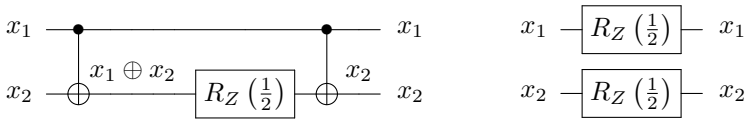
Proposition 6.1.2 implies that the problem of finding a minimal pointed parity network is equivalent to the problem of finding a CNOT-minimal circuit for a particular sum-over-paths form. However, it is not necessarily the case that a CNOT-minimal circuit having a particular sum-over-paths form is a CNOT-minimal circuit implementing a particular unitary matrix. Since for any integer-valued function $g : \mathbb{F}_2^n \to \mathbb{Z}$,

$$e^{2\pi i f(\mathbf{x})} = e^{2\pi i (f(\mathbf{x}) + g(\mathbf{x}))},$$

it may in general be possible to instead synthesize a *different* sum-over-paths form giving the same unitary operator, but with lower CNOT cost. For instance,

$$\frac{1}{2}(x_1 \oplus x_2), \qquad \text{and} \qquad \frac{1}{2}x_1 + \frac{1}{2}x_2$$

differ by an integer-valued function, $k(x_1, x_2) = x_1x_2$, and hence implement the same phase rotation. The left expression, together with the identity basis state transformation, gives rise to a minimal circuit containing 2 CNOT gates, while the expression on the right requires no CNOT gates to implement at the expense of an extra phase gate, shown below:



As we are concerned with the question of minimizing CNOT gates over circuits with equal *unitary* representations, a natural question is how this relates to the question of minimizing CNOT gates over circuits with equal *sum-over-paths* representations. The remainder of this section shows that so long as no rotation gates have angles which are *dyadic fractions* – numbers of the form $\frac{a}{2^b}$ where a and b are integers – the problems coincide.

Recall from Chapter 3 that two (CNOT-dihedral) sum-over-paths forms correspond to equivalent unitaries if and only if their phases are related by an integer-valued function. In particular,

$$[f] = \{f' : \mathbb{F}_2^n \to \mathbb{R} \mid f' = f + g \text{ where } g : \mathbb{F}_2^n \to \mathbb{Z}\},$$

and we say $f \sim f'$ if $f' \in [f]$. Recall as well that every pseudo-Boolean function $f : \mathbb{F}_2^n \to \mathbb{R}$ has a unique Fourier expansion [O'D14]. To study the relationship between Fourier expansions of equivalent functions, it will be important to know their precise form in the case of integer-valued functions.

Proposition 6.1.3. *For any integer-valued function $g : \mathbb{F}_2^n \to \mathbb{Z}$, the Fourier coefficients of g are dyadic fractions.*

Proof. Let $g : \mathbb{F}_2^n \to \mathbb{Z}$ be an integer-valued pseudo-Boolean function. It is known [HR68] that g has a unique representation as an n-ary multilinear polynomial over $\mathbb{Z}$ – that is,

$$g(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \mathbf{x}^{\mathbf{y}}$$

where $\mathbf{x}^{\mathbf{y}} = x_1^{y_1} x_2^{y_2} \cdots x_n^{y_n}$ and $a_{\mathbf{y}} \in \mathbb{Z}$ for all $\mathbf{y}$.

Using the identity $x + y - (x \oplus y) = 2xy$ for $x, y \in \mathbb{F}_2$, we derive an inclusion-exclusion formula for the monomial $\mathbf{x}^{\mathbf{y}}$ which we prove below:

$$2^{|\mathbf{y}|-1} \mathbf{x}^{\mathbf{y}} = \sum_{\mathbf{y}' \subseteq \mathbf{y}} (-1)^{|\mathbf{y}'|-1} \chi_{\mathbf{y}'}(\mathbf{x}). \tag{6.1}$$

Note that binary vectors are viewed as subsets of $\{1, \ldots, n\}$ for convenience. Since $a_{\mathbf{y}} \in \mathbb{Z}$ for all $\mathbf{y}$ and dyadic fractions are closed under addition, we observe that the Fourier coefficients of $g(\mathbf{x})$ are dyadic fractions:

$$g(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} a_{\mathbf{y}} \mathbf{x}^{\mathbf{y}} = \sum_{\mathbf{y} \in \mathbb{F}_2^n} \left(\sum_{\mathbf{y}' \subseteq \mathbf{y}} (-1)^{|\mathbf{y}'|-1} \frac{a_{\mathbf{y}}}{2^{|\mathbf{y}|-1}} \right) \chi_{\mathbf{y}'}(\mathbf{x}).$$

□

It remains to prove Equation (6.1), which we obtain as a simple corollary of the more convenient lemma below.

Lemma 6.1.4. *For any* $\mathbf{x} \in \mathbb{F}_2^n$,

$$2^{n-1} x_1 x_2 \cdots x_n = \sum_{\mathbf{y} \in \mathbb{F}_2^n} (-1)^{|\mathbf{y}|-1} \chi_{\mathbf{y}}(\mathbf{x})$$

Proof. Clearly the formula is satisfied for $n = 1$. Now consider $n = k + 1$ for some k. Using the identity $x + y - (x \oplus y) = 2xy$ for any $x, y \in \mathbb{F}_2$ and basic arithmetic we observe that

$$\begin{aligned} 2^k x_1 x_2 \cdots x_{k+1} &= 2^{k-1} x_1 x_2 \cdots (2 x_k x_{k+1}) \\ &= 2^{k-1} x_1 x_2 \cdots (x_k + x_{k+1} - (x_k \oplus x_{k+1})) \\ &= 2^{k-1} x_1 x_2 \cdots x_k + 2^{k-1} x_1 x_2 \cdots x_{k+1} - 2^{k-1} x_1 x_2 \cdots (x_k \oplus x_{k+1}) \end{aligned}$$

Next we define the length k vectors $\mathbf{x}', \mathbf{x}'', \mathbf{x}''' \in \mathbb{F}_2^n$ as follows:

$$x_i' = x_i, \qquad x_i'' = \begin{cases} x_{k+1} \text{ if } i = k \\ x_i \text{ otherwise} \end{cases}, \qquad x_i''' = \begin{cases} x_k \oplus x_{k+1} \text{ if } i = k \\ x_i \text{ otherwise} \end{cases}$$

By induction we see that

$$\begin{aligned} 2^k x_1 \cdots x_{k+1} &= 2^{k-1} x_1' x_2' \cdots x_k' + 2^{k-1} x_1'' x_2'' \cdots x_k'' - 2^{k-1} x_1''' x_2''' \cdots x_k''' \\ &= \sum_{\mathbf{y} \in \mathbb{F}_2^k} (-1)^{|\mathbf{y}|-1} \left(\chi_{\mathbf{y}}(\mathbf{x}') + \chi_{\mathbf{y}}(\mathbf{x}'') - \chi_{\mathbf{y}}(\mathbf{x}''') \right) \\ &= \sum_{\substack{\mathbf{y} \in \mathbb{F}_2^k, \\ y_k = 0}} (-1)^{|\mathbf{y}|-1} \chi_{\mathbf{y}}(\mathbf{x}') + \sum_{\substack{\mathbf{y} \in \mathbb{F}_2^k, \\ y_k = 1}} (-1)^{|\mathbf{y}|-1} \left(\chi_{\mathbf{y}}(\mathbf{x}') + \chi_{\mathbf{y}}(\mathbf{x}'') - \chi_{\mathbf{y}}(\mathbf{x}''') \right) \\ &= \sum_{\substack{\mathbf{y} \in \mathbb{F}_2^k, \\ y_k = 0}} (-1)^{|\mathbf{y}|-1} \left(\chi_{\mathbf{y}}(\mathbf{x}') - \chi_{\mathbf{y}}(\mathbf{x}') \oplus x_k - \chi_{\mathbf{y}}(\mathbf{x}') \oplus x_{k+1} + \chi_{\mathbf{y}}(\mathbf{x}') \oplus x_k \oplus x_{k+1} \right) \\ &= \sum_{\mathbf{y} \in \mathbb{F}_2^{k+1}} (-1)^{|\mathbf{y}|-1} \chi_{\mathbf{y}}(\mathbf{x}) \end{aligned}$$

□

Corollary 6.1.5 (Equation (6.1))**.** *For any* $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$,

$$2^{|\mathbf{y}|-1} \mathbf{x}^{\mathbf{y}} = \sum_{\mathbf{y}' \subseteq \mathbf{y}} (-1)^{|\mathbf{y}'|-1} \chi_{\mathbf{y}'}(\mathbf{x})$$

Remark 6.1.6. The proof of Proposition 6.1.3 also suffices to prove a more general result, namely that any function from $\mathbb{F}_2^n$ to an Abelian group G in which 2 is a regular element has a unique Fourier expansion over G. This also offers an alternative proof to the fact that the code C_m^n from Chapter 5 is the trivial code whenever m is odd; indeed, 2 is regular in any such group.

We next use the above proposition to show that any pseudo-Boolean function with non-dyadic spectrum has a property of *minimal support* over all equivalent functions. This is important as a parity network for S is also a parity network for any subset of S.

Proposition 6.1.7. *Let $f : \mathbb{F}_2^n \to \mathbb{R}$ be a pseudo-Boolean function having a Fourier spectrum not containing any non-zero dyadic fractions. Then for any $f' \sim f$,*

$$\mathrm{supp}(\widehat{f}) \subseteq \mathrm{supp}(\widehat{f'}).$$

Proof. Consider some pseudo-Boolean function f' such that $f' \sim f$. By definition we have $f' = f + g$ for some function $g : \mathbb{F}_2^n \to \mathbb{Z}$. Expanding $f(\mathbf{x})$ and $g(\mathbf{x})$ with their Fourier expansions we have

$$f'(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbb{F}_2^n} (\widehat{f}(\mathbf{y}) + \widehat{g}(\mathbf{y}))\chi_{\mathbf{y}}(\mathbf{x}).$$

Now since for any $\mathbf{y}$, $\widehat{g}(\mathbf{y}) = \frac{a}{2^b}$, $\widehat{f}(\mathbf{y}) + \widehat{g}(\mathbf{y}) \neq 0$. Thus $\mathrm{supp}(\widehat{f}) \subseteq \mathrm{supp}(\widehat{f'})$ as required. □

We can now prove that the problem of synthesizing a minimal (pointed) parity network is at least as hard as general CNOT-minimization. As a corollary, synthesizing a minimal parity network solves the CNOT-minimization problem whenever the rotation angles, and hence the Fourier coefficients, are not dyadic fractions.

Theorem 6.1.8. *Given $A \in \mathrm{GL}(n, \mathbb{F}_2)$, the problem of finding a minimal parity network for $S \subseteq \mathbb{F}_2^n$ pointed at A reduces (in polynomial time) to the problem of finding a* CNOT*-minimal circuit equivalent to an n-qubit circuit C over $\{\mathrm{CNOT}, X, R_Z\}$.*

Proof. Given $A \in \mathrm{GL}(n, \mathbb{F}_2)$ and $S \subseteq \mathbb{F}_2^n$, define $f : \mathbb{F}_2^n \to \mathbb{R}$ as

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in S} \frac{1}{3}\chi_{\mathbf{y}}(\mathbf{x}).$$

Recall that a circuit C over $\{\text{CNOT}, X, R_Z\}$ implementing the sum-over-paths form $(f, A_{\mathbf{0}})$ can be constructed in polynomial time.

Now let C' be a CNOT-minimal circuit equivalent to C. We know the sum-over-paths form of C' must be $(f', A_{\mathbf{0}})$ for some $f' \sim f$. However, by Proposition 6.1.7, $S = \text{supp}(\widehat{f}) \subseteq \text{supp}(\widehat{f'})$, so by definition, the circuit obtained from C' by removing all X and R_Z gates is a (necessarily minimal) parity network for S pointed at A. □

Remark 6.1.9. As in Chapter 5, in cases when the Fourier coefficients contain dyadic fractions, it may in general be possible to further minimize CNOT-count by optimizing *over all equivalent phase functions*. However, in contrast to the Z_m-gate optimization problem, the Fourier spectrum does not correspond directly to the size of a minimal parity network – e.g., $\frac{1}{2}(x_1 \oplus x_2)$ has smaller support than $\frac{1}{2}x_1 + \frac{1}{2}x_2$ but a larger minimal parity network as shown earlier – which appears to make the problem of minimizing CNOTs size over all equivalent functions more difficult.

6.2 Complexity of parity network minimization

We now turn to the question of the complexity of computing minimal parity networks. We study two cases in particular where the problem can be shown to be **NP**-complete – the fixed-target case, and with encoded inputs (i.e. with ancillae). At the end of the section we discuss the case of synthesizing a minimal parity network with arbitrary targets and no ancillae.

6.2.1 Fixed-target minimal parity network

We call the problem of synthesizing a minimal parity network in which every CNOT gate has the same target the *fixed-target minimal parity network problem*. Formally, we define the decision problem MPNP_{FT} below:

PROBLEM:	Fixed-target minimal parity network (MPNP_{FT})
INSTANCE:	A set of strings $S \subseteq \mathbb{F}_2^n$, and a positive integer k.
QUESTION:	Does there exist an n-qubit circuit C over CNOT gates of length at most k such that C is a parity network for S?

Remark 6.2.1. In general not every set of strings S admits an ancilla-free fixed-target parity network, as the value of the target bit necessarily appears in every parity calculation of a fixed-target CNOT circuit. It follows that an (ancilla-free) parity network for S is

synthesizable if and only if there exists an index i such that for every $\mathbf{y} \in S$, $y_i = 1$. However, a fixed-target parity network may always be synthesized by adding a single ancillary bit initialized to the state $|0\rangle$. In particular, given a set $S \subseteq \mathbb{F}_2^n$ and $A \in \mathrm{GL}(n, \mathbb{F}_2)$, we may construct

$$S' = \{(\mathbf{y}, 1) \mid \mathbf{y} \in S\},$$

where we recall that $(\mathbf{y}, 1)$ denotes the length $n+1$ string obtained by concatenating $\mathbf{y}$ with 1. It may then be observed that a fixed-target parity network for S' is always synthesizable, and in particular forms a parity network for S when the $(n+1)$th bit is initialized to $|0\rangle$.

To show that the fixed-target minimal parity network problem is **NP**-complete, we introduce the *Hamming salesman problem* (HTSP) [EKP85]. Recall that the n-dimensional hypercube is the graph with vertices $\mathbf{x} \in \mathbb{F}_2^n$ and edges between $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^n$ if $\mathbf{x}$ and $\mathbf{y}$ differ in one coordinate (i.e. have Hamming distance 1).

PROBLEM:	Hamming salesman (HTSP)
INSTANCE:	A set of strings $S \subseteq \mathbb{F}_2^n$, and a positive integer k.
QUESTION:	Does there exist a path in the n-dimensional hypercube of length at most k starting at $\mathbf{0}$ and going through each vertex $\mathbf{y} \in S$?

An equivalent (from a complexity standpoint) version of the Hamming salesman problem exists where a cycle rather than path is found. Intuitively, the Hamming salesman problem is to find a sequence of at most k bit-flips iterating through every string in some set S starting from the initial string $00\ldots0$. In the case when $S = \mathbb{F}_2^n$ the minimal number of bit flips is known to be 2^n, corresponding to one bit flip per string; this is the well known *Gray code*, a total ordering on $\mathbb{F}_2^n$ where each subsequent string differs by exactly one bit. We will come back to this connection later in Section 6.3 when designing a synthesis algorithm.

Ernvall, Katajainen and Penttonen [EKP85] show that the Hamming salesman problem is in fact **NP**-complete, hence we can use a reduction from HTSP to prove **NP**-completeness of $\mathrm{MSP}_{\mathrm{FT}}$.

Theorem 6.2.2. *$MPNP_{FT}$ is **NP**-complete.*

Proof. Clearly $\mathrm{MPNP}_{\mathrm{FT}}$ is in **NP**, as the state of each bit as a parity of the input values at each state in a CNOT circuit is polynomial-time computable [AMM14], and hence a parity network can be efficiently verified. Since HTSP is **NP**-complete [EKP85] it then suffices to show **NP**-hardness by reducing the Hamming salesman problem to the fixed-target minimal parity network problem.

Given an instance $(S \subseteq \mathbb{F}_2^n, k)$ of HTSP, we construct an instance $(S' \subseteq \mathbb{F}_2^{n+1}, k')$ of MPNP_{FT} with size polynomial in $|S| \cdot n$ as follows:

$$S' = \{(\mathbf{x}, 1) \mid \mathbf{x} \in S\}, \qquad k' = k.$$

Suppose there exists a fixed-target parity network C for S' with length at most k. Without loss of generality we may assume that the fixed target is the $(n+1)$th bit, as if some $i \neq n+1$ is the target index, then for all $\mathbf{y} \in S'$, $y_i = 1 = y_{n+1}$ and hence swapping bits i and $n+1$ yields a parity network for S'. We can then construct a length k hypercube path through each vertex $\mathbf{y} \in S$ with starting point $\mathbf{0}$ by mapping C to a sequence of bit flips on each CNOT's control bit. Indeed, by noting that

$$\text{CNOT}|x_i\rangle|x_{n+1} \oplus \chi_{\mathbf{y}}(\mathbf{x})\rangle = |x_i\rangle|x_{n+1} \oplus \chi_{\mathbf{y} \oplus e_i}(\mathbf{x})\rangle,$$

where e_i is the ith elementary vector, each CNOT gate in C with control i has the affect of flipping the ith bit of $\mathbf{y}$. By the definition of a parity network, for every $\mathbf{y} \in S$, the parity

$$x_{n+1} \oplus \chi_{\mathbf{y}}(\mathbf{x})$$

appears as an annotation in the circuit, in particular on the $(n+1)$th bit which had initial state $x_{n+1} \oplus \chi_{\mathbf{0}}(\mathbf{x})$, hence the sequence of bit flips passes through each vertex $\mathbf{y} \in S$ starting from $\mathbf{0}$.

Likewise, if there exists a length k tour through each $\mathbf{y} \in S$, given by a sequence of bit flips, the circuit defined by mapping each bit flip on i to a CNOT with control i and target $n+1$ is a length k parity network for S' and A'. □

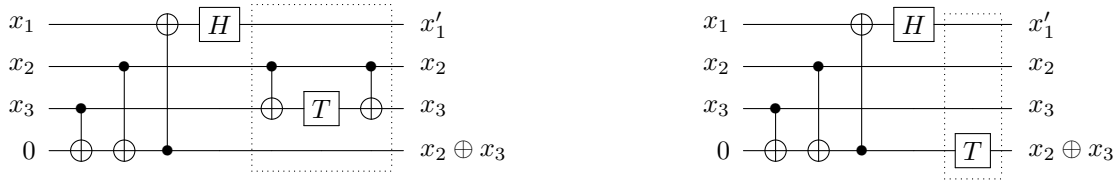
As the minimum k for which a parity network exists is at most $(n-1) \cdot |S|$, the optimization version of MPNP_{FT} is also in **NP**, and hence is **NP**-complete.

Corollary 6.2.3. *The problem of finding a minimal fixed-target parity network is **NP**-complete.*

It may be observed that the proof of Theorem 6.2.2 can be modified to show that the problem of finding a minimal *pointed* parity network with fixed CNOT targets. In particular, taking A to be the identity transformation gives a reduction from the cycle version of HTSP.

6.2.2 Minimal parity network with encoded inputs

Up to this point, we have discussed only the optimization of pure CNOT-dihedral circuits. However, our main interest in such optimization problems is to optimally synthesize CNOT-dihedral *sub-circuits* of larger circuits. In this case, the inputs to the sub-circuit may generally be contained in some subset of $\mathbb{F}_2^n$. Knowing this more precise information about the input space of the circuit may allow shorter parity networks to be synthesized. For instance, the boxed CNOT-dihedral sub-circuit below on the left performs a phase rotation of $\frac{1}{8}(x_2 \oplus x_3)$ – by noting that the ancilla begins the sub-circuit already in the state $x_2 \oplus x_3$ we can remove both CNOT gates, as shown by the equivalent circuit on the right.



We now consider the problem of synthesizing minimal parity networks when some of the inputs are linear combinations of others. Formally, given a linear transformation $E \in \mathrm{L}(\mathbb{F}_2^n, \mathbb{F}_2^m)$ where $m > n$, a string $\mathbf{w} \in \mathbb{F}_2^m$ is an *encoding* of $\mathbf{x} \in \mathbb{F}_2^n$ if $E\mathbf{x} = \mathbf{w}$. We ignore affine factors in the input space since, as noted before, they do not affect CNOT-counts. The *minimal parity network with encoded inputs problem* ($\mathrm{MPNP_E}$) is then to find a parity network for a given set $S \subseteq \mathbb{F}_2^n$ as a function of the primary inputs $\mathbf{x} \in \mathbb{F}_2^n$, but with the initial state $|E\mathbf{x}\rangle$ rather than $|\mathbf{x}\rangle$.

Problem:	Minimal parity network with encoded inputs ($\mathrm{MPNP_E}$)
Instance:	A set of strings $S \subseteq \mathbb{F}_2^n$, a linear transformation $E \in \mathbb{F}_2^{m \times n}$, and a positive integer k.
Question:	Does there exist an m-qubit circuit C over CNOT gates of length at most k such that C is a parity network for some set $S' \subseteq \mathbb{F}_2^m$ where for any $\mathbf{y} \in S$ there exists $\mathbf{w} \in S'$ such that $E^T\mathbf{w} = \mathbf{y}$?

It can be observed that a parity network for some set S' as above is equivalent to a parity network for S starting from the initial state $|E\mathbf{x}\rangle$ for any $\mathbf{x} \in \mathbb{F}_2^n$. In particular, for any $\mathbf{w} \in S'$ and $\mathbf{x} \in \mathbb{F}_2^n$,

$$\chi_{\mathbf{w}}(E\mathbf{x}) = \mathbf{w}^T E\mathbf{x} = \chi_{E^T\mathbf{w}}(\mathbf{x}) = \chi_{\mathbf{y}}(\mathbf{x}).$$

In general, there may be many such S', as for example

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \mathbf{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

has two solutions: $\mathbf{w} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T$ and $\mathbf{w} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$, corresponding to different ways of computing $x_1 \oplus x_2$ from the input state $|E\mathbf{x}\rangle = |x_1 x_2 (x_1 \oplus x_2)\rangle$.

MPNP_{E} is again **NP**-complete, which we prove by a reduction from the well known **NP**-complete *Maximum-likelihood Decoding Problem* (MLDP).

PROBLEM:	Maximum-likelihood decoding (MLDP)
INSTANCE:	A linear transformation $H \in \mathbb{F}_2^{m \times n}$, a vector $\mathbf{y} \in \mathbb{F}_2^n$, and a positive integer k.
QUESTION:	Does there exist a vector $\mathbf{w} \in \mathbb{F}_2^m$ of weight at most k such that $H\mathbf{w} = \mathbf{y}$?

In the case when H is the parity check matrix of a code C and $\mathbf{y}$ is the syndrome of some vector $\mathbf{z}$, finding the minimum such $\mathbf{w}$ gives the minimum weight vector in the coset of $\mathbf{z} + C$, corresponding to a minimum distance decoding of $\mathbf{z}$. Berlekamp, McEliece, and van Tilborg [BMvT06] proved that the Maximum-likelihood decoding problem is **NP**-complete, and so we may reduce it to the minimal parity network with encoded inputs problem to show **NP**-completeness.

Theorem 6.2.4. *$MPNP_E$ is **NP**-complete.*

Proof. As noted in the proof of Theorem 6.2.2, MPNP_{E} is clearly in **NP** since a parity network can be verified in polynomial time. To establish **NP**-hardness we give a reduction from MLDP.

Given an instance $(H, \mathbf{y}, k)$ of MLDP we construct an instance (S, k') of MPNP_{E} as follows:

$$S = \{\mathbf{y}\}, \qquad E = H^T, \qquad k' = k - 1.$$

Suppose there exists a vector $\mathbf{w} \in \mathbb{F}_2^m$ of weight at most k such that $H\mathbf{w} = \mathbf{y}$. Then we know $S' = \{\mathbf{w}\}$ satisfies the requirement that for any $\mathbf{y} \in S$ there exists $\mathbf{w} \in S'$ such that $E^T\mathbf{w} = H\mathbf{w} = \mathbf{y}$. Moreover, the parity computation $\chi_{\mathbf{w}}$ can be computed with $|\mathbf{w}| - 1 \leq k'$ CNOT gates, hence there exists a parity network of length at most k' for S'

On the other hand, suppose there exists a length $\leq k'$ parity network for some set S' where there exists $\mathbf{w} \in S'$ such that $E^T\mathbf{w} = H\mathbf{w} = \mathbf{y}$. By noting that

$$\text{CNOT}|\chi_{\mathbf{y}}(\mathbf{x})\rangle|\chi_{\mathbf{z}}(\mathbf{x})\rangle = |\chi_{\mathbf{y}}(\mathbf{x})\rangle|\chi_{\mathbf{y}\oplus\mathbf{z}}(\mathbf{x})\rangle,$$

$|\mathbf{y} \oplus \mathbf{z}| \leq |\mathbf{y}| + |\mathbf{z}|$. As each bit starts in some state $x_i = \chi_{e_i}(\mathbf{x})$ we see that the size of the parity in any bit at any point in the parity network is at most $k' + 1 \leq k$, and so $|\mathbf{w}| \leq k$ as required. □

Corollary 6.2.5. *The problem of finding a minimal parity network with encoded inputs is* ***NP****-hard.*

As in the fixed-target case, the problem of finding a minimal pointed parity network with encoded inputs is also **NP**-complete, by virtue of the fact that a minimal identity parity network for a singleton set $S = \{\mathbf{y}\}$ necessarily has the form $C \fatsemi C'$ where both C and $(C')^\dagger$ are parity networks for S. Recall that the inverse circuit $(C')^\dagger$ has the same length as C'.

6.2.3 Discussion

While the case of parity network synthesis with encoded inputs is the most relevant to the usage (CNOT-dihedral sub-circuit synthesis) in this thesis, Theorem 6.2.4 relies on the hardness of finding a minimal sum of linearly dependent vectors, or minimum distance decoding. It would appear that the problem becomes easier when using unencoded inputs, as each vector and hence parity may be expressed uniquely – i.e. minimally – over the inputs.

We leave determining the complexity of the synthesizing a minimal parity network without encoded inputs as an open problem. We do however conjecture that the identity version – i.e. synthesizing a minimal identity parity network – is at least as hard as synthesizing a minimal fixed-target identity parity network. In particular, it appears that whenever one bit appears in every parity there exists a minimal identity parity network that targets only that bit, and hence these two problems coincide.

6.3 A heuristic synthesis algorithm

We now present an efficient, heuristic algorithm for synthesizing parity networks. The algorithm is inspired by Gray codes, which iterate through all 2^n elements of $\mathbb{F}_2^n$ minimally with one bit flip per string. The situation is distinct for synthesizing parity networks as the bits which can be flipped depend on the state of all n bits, so our method works by trying to identify subsets of S which can be efficiently iterated with a Gray code-like construction on a fixed target. In the limit where $S = \mathbb{F}_2^n$, the algorithm gives a minimal size parity

network for S. Again we focus just on the problem of synthesizing a parity network up to some arbitrary overall linear transformation.

The algorithm GRAY-SYNTH, is presented in pseudo-code in Algorithm 6.1. Recall that $E_{i,j}$ denotes the elementary $\mathbb{F}_2$-matrix adding row i to row j. Given a set of binary strings S, the algorithm synthesizes a parity network for S by repeatedly choosing an index i to expand and then effectively recurring on the co-factors S_0 and S_1, consisting of the strings $\mathbf{y} \in S$ with $y_i = 0$ or 1, respectively. As a subset S is recursively expanded, CNOT gates are applied so that a designated *target* bit contains the (partial) parity $\chi_{\mathbf{y}}(\mathbf{x})$ where $y_i = 1$ if and only if $y'_i = 1$ for all $\mathbf{y}' \in S$ – if S is a singleton $\{\mathbf{y}'\}$, then $\mathbf{y} = \mathbf{y}'$, hence the target bit contains the value $\chi_{\mathbf{y}'}(\mathbf{x})$ as desired. Notably, rather than *uncomputing* this sequence of CNOT gates when a subset S is finished being synthesized, the algorithm maintains the invariant that the remaining parities to be computed are expressed over the current state of the bits. This allows the algorithm to avoid the "backtracking" inherent in uncomputing-based methods.

More precisely, the invariant of Algorithm 6.1 described above is expressed in the following lemma.

Lemma 6.3.1. *Let C be a* CNOT *circuit and $S \subseteq \mathbb{F}_2^n$. For any positive integer i we let $C_{\leq i}$ denote the first i gates of C, c_i and t_i be the control and target of the ith* CNOT *gate in C. If we define*

$$\begin{aligned} A_0 &= I & \mathbf{y}_0 &= \mathbf{y} \\ A_i &= E_{c_i,t_i} A_{i-1} & \mathbf{y}_i &= E_{t_i,c_i} \mathbf{y}_{i-1} \end{aligned}$$

for every $\mathbf{y} \in S$, then it follows that for any $\mathbf{x} \in \mathbb{F}_2^n$, $U_{C \leq i}|\mathbf{x}\rangle = |A_i\mathbf{x}\rangle$ and

$$\chi_{\mathbf{y}_i}(A_i\mathbf{x}) = \chi_{\mathbf{y}}(\mathbf{x}).$$

Proof. The fact that $U_{C \leq i}|\mathbf{x}\rangle = |A_i\mathbf{x}\rangle$ follows simply from the fact that $\mathrm{CNOT}_{i,j}|\mathbf{x}\rangle = |E_{i,j}\mathbf{x}\rangle$. For the latter fact, clearly $\chi_{\mathbf{y}_i}(A_i\mathbf{x}) = \chi_{\mathbf{y}}(\mathbf{x})$ by definition. Moreover, recall that

$$\chi_{\mathbf{y}}(A\mathbf{x}) = \mathbf{y}^T A\mathbf{x} = \chi_{A^T\mathbf{y}}(\mathbf{x})$$

and hence by induction,

$$\begin{aligned} \chi_{\mathbf{y}_i}(A_i\mathbf{x}) &= \chi_{(A_i)^T\mathbf{y}_i}(\mathbf{x}) \\ &= \chi_{(A_{i-1})^T E_{t_i,c_i} E_{t_i,c_i} \mathbf{y}_{i-1}}(\mathbf{x}) \\ &= \chi_{(A_{i-1})^T \mathbf{y}_{i-1}}(\mathbf{x}) \\ &= \chi_{\mathbf{y}}(\mathbf{x}). \end{aligned}$$

□

Algorithm 6.1 Algorithm for synthesizing a parity network

function GRAY-SYNTH($S \subseteq \mathbb{F}_2^n$)
    New empty circuit C
    New empty stack Q
    Q.push(S, $\{1, \ldots, n\}$, ϵ)
    while Q non-empty **do**
        (S, I, i) $\leftarrow$ Q.pop
        if $S = \emptyset$ **or** $I = \emptyset$ **then return**
        else if $i \in \mathbb{N}$ **then**
            while $\exists j \neq i \in \{1, \ldots, n\}$ s.t. $\mathbf{y}_j = 1$ for all $\mathbf{y} \in S$ **do**
                $C \leftarrow C \fatsemi \mathrm{CNOT}_{j,i}$
                for all (S', I', i')$\in Q \cup$(S, I, i) **do**
                    for all $\mathbf{y} \in S'$ **do**
                        $\mathbf{y} \leftarrow E_{i,j}\mathbf{y}$
                    end for
                end for
            end while
        end if
        $j \leftarrow \arg\max_{j \in I} \max_{x \in \mathbb{F}_2} |\{\mathbf{y} \in S \mid y_j = x\}|$
        $S_0 \leftarrow \{\mathbf{y} \in S \mid y_j = 0\}$
        $S_1 \leftarrow \{\mathbf{y} \in S \mid y_j = 1\}$
        if $i \in \{\epsilon\}$ **then**
            Q.push(S_1, $I \setminus \{j\}$, j)
        else
            Q.push(S_1, $I \setminus \{j\}$, i)
        end if
        Q.push(S_0, $I \setminus \{j\}$, i)
    end while
    return C
end function

It is clear to see that each $\mathbf{y}_i$ for $\mathbf{y} \in S$ in Lemma 6.3.1 is the value of $\mathbf{y}$ after i iterations. To see then that the output is in fact a parity network for S, it suffices to observe that whenever $S = \{\mathbf{y}_i\}$ and $I = \emptyset$, $|\mathbf{y}_i| = 1$ and thus by Lemma 6.3.1, some bit is in the state $|\chi_{\mathbf{y}_i}(A_i\mathbf{x})\rangle = |\chi_{\mathbf{y}}(\mathbf{x})\rangle$ after the ith CNOT gate. While the fact that $|\mathbf{y}_i| = 1$ is assured by lines 9-16 in this case, the non-zero elements of the target strings are actually zero-ed out earlier, as the algorithm expands each coordinate. In particular, the first time the "1" branch is taken when expanding a set, corresponding to the first 1 seen over the indices previously examined, the target bit i is set – taking further "1" branches result in the row j being flipped to 0 with a single CNOT. In this way the algorithm makes use of the redundancy in Fourier spectrum S.

In practice, a parity network implementing some particular basis state transformation is typically needed. We take the approach of synthesizing a pointed parity network by first synthesizing a regular parity network, then implementing the remaining linear transformation – i.e. AA_i^{-1} where A_i is the linear transformation implemented by the network. In our implementation we use the Patel-Markov-Hayes algorithm [PMH08] which gives asymptotically optimal CNOT count. To achieve the necessary affine factor $\mathbf{b}$ in the output, the circuit $X^{\mathbf{b}} = X_1^{b_1} \otimes \cdots \otimes X_n^{b_n}$ is appended to the end.

While the correctness of Algorithm 6.1 is independent of the choice of index j to expand in line 18, in practice it has a large impact on the size of the resulting parity network. We chose j so as to maximize the size of the largest subset, S_0 or S_1, i.e. $j = \arg\max_{j \in I} \max_{x \in \mathbb{F}_2} |\{\mathbf{y} \in S \mid y_j = x\}|$. The intuition behind this choice is that as a subset S of $\mathbb{F}_2^n$ with m bits fixed approaches $|S| = 2^{n-m}$, the minimal parity network for S approaches one CNOT per string, corresponding to the Gray code in the limit. We also ran experiments with other methods of choosing j; we found that $j = \arg\max_{j \in I} \max_{x \in \mathbb{F}_2} |\{\mathbf{y} \in S \mid y_j = x\}|$ gave the best results on average.

6.3.1 Examples

Example 6.3.2. To illustrate Algorithm 6.1, we demonstrate the use of GRAY-SYNTH to synthesize a circuit over $\{\text{CNOT}, X, R_Z\}$ implementing the operator $U : |\mathbf{x}\rangle \mapsto e^{2\pi i f(\mathbf{x})}|\mathbf{x}\rangle$ where

$$f(\mathbf{x}) = \frac{1}{8}\left[(x_2 \oplus x_3) + x_1 + (x_1 \oplus x_4) + (x_1 \oplus x_2 \oplus x_3) + (x_1 \oplus x_2 \oplus x_4) + (x_1 \oplus x_2)\right].$$

$$\left(\begin{array}{|cccccc|}\hline 0&1&1&1&1&1\\ 1&0&0&1&1&1\\ 1&0&0&1&0&0\\ 0&0&1&0&1&0\\ \hline\end{array}\right)$$

x_1 — x_1
x_2 — x_2
x_3 — x_3
x_4 — x_4

Starting with the initial set S written as a matrix on the left, we choose a bit maximizing the number of 0's or 1's in S. As $j = 1$ in this case, we construct the cofactors S_0 and S_1 on the values in the first row, and recurse on S_0 as indicated by the box in the diagram below.

$$\left(\begin{array}{|c|ccccc|}\hline 0&1&1&1&1&1\\ \hline 1&0&0&1&1&1\\ 1&0&0&1&0&0\\ 0&0&1&0&1&0\\ \hline\end{array}\right)$$

x_1 — x_1
x_2 — x_2
x_3 — x_3
x_4 — x_4

The algorithm next selects row 2 and immediately descends into the 1-cofactor, since $S_0 = \emptyset$. Again, the algorithm selects row 3, and since both rows 2 and 3 have the value 1, a CNOT is applied with bit 2 as the target and 3 as the control. The remaining vectors are updated by multiplying with $E_{2,3}$ – the modified entries are shown in red.

$$\left(\begin{array}{|c|ccccc|}\hline 0&1&1&1&1&1\\ \hline 1&0&0&1&1&1\\ 1&0&0&1&0&0\\ 0&0&1&0&1&0\\ \hline\end{array}\right)$$

x_1 — x_1
x_2 — x_2
x_3 — x_3
x_4 — x_4

$$\left(\begin{array}{|c|ccccc|}\hline 0&1&1&1&1&1\\ \hline 1&0&0&1&1&1\\ 1&0&0&1&0&0\\ 0&0&1&0&1&0\\ \hline\end{array}\right)\rightarrow\left(\begin{array}{|c|ccccc|}\hline 0&1&1&1&1&1\\ \hline 1&0&0&1&1&1\\ 0&0&0&0&1&1\\ 0&0&1&0&1&0\\ \hline\end{array}\right)$$

x_1 —— x_1
x_2 —⊕— $x_2 \oplus x_3$
x_3 —●— x_3
x_4 —— x_4

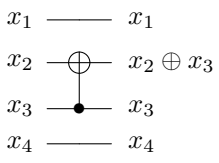
As the final row has the value 0, we're finished with this column and may continue with the remaining ones.

$$\left(\begin{array}{|c|ccccc|}\hline 0&1&1&1&1&1\\ \hline 1&0&0&1&1&1\\ 0&0&0&0&1&1\\ 0&0&1&0&1&0\\ \hline\end{array}\right)$$

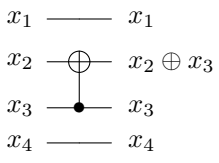
x_1 —— x_1
x_2 —⊕— $x_2 \oplus x_3$
x_3 —●— x_3
x_4 —— x_4

Again the algorithm chooses row 2 maximizing the number of entries which are the same for the remaining columns, and recurses on the 0-cofactor as shown below.

$$\left(\begin{array}{c|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right)$$

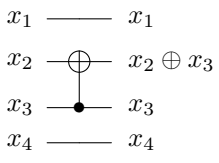


$$\left(\begin{array}{c|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right)$$

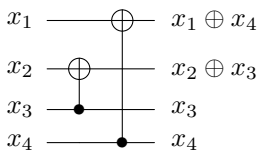


In expanding the last row, we first examine the 0-cofactor and find nothing to do, then the 1-cofactor, at which point we need to apply a CNOT with target bit 1 and control 4.

$$\left(\begin{array}{c|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right)$$

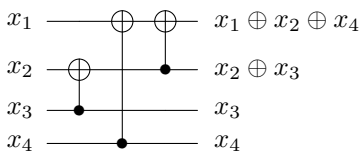


$$\left(\begin{array}{ccc|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right)$$



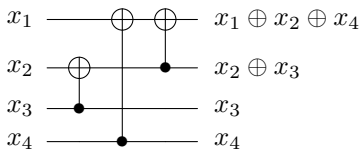
Now backtracking and entering the 1-cofactor for the remaining columns, we find we need to apply a CNOT between bits 2 and 1 to zero out the row.

$$\left(\begin{array}{ccc|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right)$$

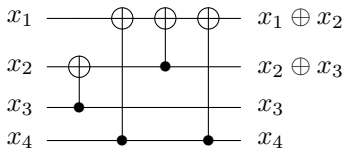


Continuing on we recurse on the 0-cofactor of row 3 and apply a CNOT with target bit 1, control 4, before backtracking to the 1-cofactor.

$$\left(\begin{array}{ccc|ccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right)$$

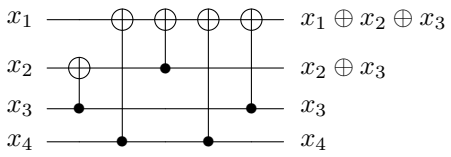


$$\left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&1&1\\ 0&0&0&1&0&1 \end{array}\right) \to \left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&1&1\\ 0&0&0&0&1&0 \end{array}\right)$$

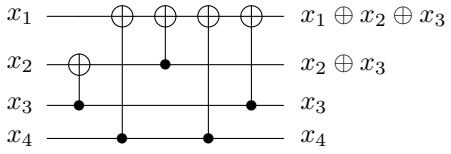


For the remaining two columns we first zero out row 3 by applying a CNOT gate between bits 3 and 1, then finally descend into the cofactors on the last row.

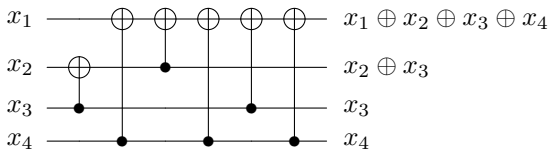
$$\left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&1&1\\ 0&0&0&0&1&0 \end{array}\right) \to \left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&1&0 \end{array}\right)$$



$$\left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&1&0 \end{array}\right)$$



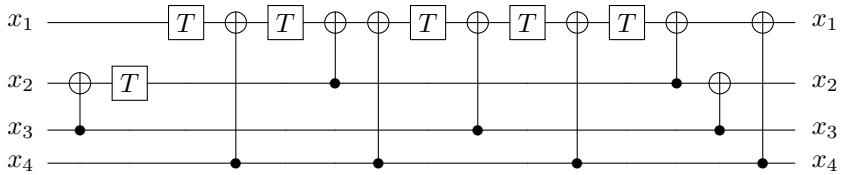
$$\left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&1&0 \end{array}\right) \to \left(\begin{array}{cccccc} 0&1&1&1&1&1\\ 1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{array}\right)$$



The overall linear transformation applied is

$$A = \begin{bmatrix} 1&1&0&1\\ 0&1&1&0\\ 0&0&1&0\\ 0&0&0&1 \end{bmatrix}$$

so the algorithm completes by appending a circuit computing A^{-1}. Inserting $T = R_Z(1/8)$ gates in the relevant positions, we get the following circuit computing $|\mathbf{x}\rangle \mapsto e^{2\pi i f(\mathbf{x})}|\mathbf{x}\rangle$:



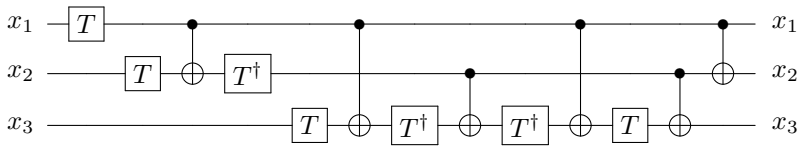


Figure 6.1: Circuit implementing the doubly-controlled Z gate CCZ synthesized with Algorithm 6.1. The CNOT-minimal Fourier expansion in this case gives $S = \mathbb{F}_2^3 \setminus \{\mathbf{0}\}$, and $A = I$.

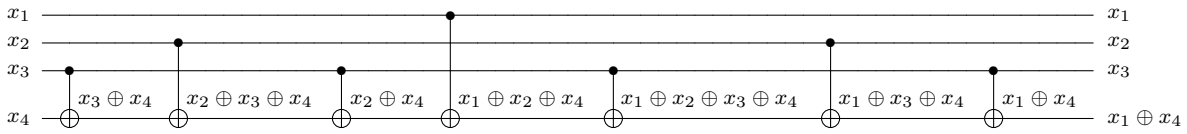


Figure 6.2: Annotated parity network for the set $S = \{(\mathbf{y}, 1) \mid \mathbf{y} \in \mathbb{F}_2^3\}$. Note that the parity network corresponds exactly to the Gray code on $\mathbb{F}_2^3$.

Example 6.3.3. Figure 6.1 shows a circuit implementing the doubly-controlled Z gate, corresponding to an identity parity network for $\mathbb{F}_2^3 \setminus \{000\}$, synthesized with Algorithm 6.1 followed by the Patel-Markov-Hayes algorithm. In this case both the parity network and the identity parity network are minimal, as verified by brute force search – further, the circuit synthesized by Algorithm 6.1 reproduces exactly the minimal circuit for $\mathbb{F}_2^3 \setminus \{000\}$ from [WGMAG14]. In general, for any n the identity parity network for $\mathbb{F}_2^n \setminus \{\mathbf{0}\}$ synthesized in this manner has the same structure, using $2^n - 2$ CNOT gates, compared to 2^n bit flips for the Gray code.

Example 6.3.4. If instead of $S = \mathbb{F}_2^n \setminus \{\mathbf{0}\}$ we have $S \simeq \mathbb{F}_2^m$ for some $m < n$, Algorithm 6.1 instead gives a circuit corresponding directly to the Gray code. In particular, Figure 6.2 shows a parity network for $S = \{(\mathbf{y}, 1) \mid \mathbf{y} \in \mathbb{F}_2^3\} \simeq \mathbb{F}_2^3$ synthesized with Algorithm 6.1, where $S \simeq \mathbb{F}_2^3$. In this case it can be observed that the controls of the CNOT gates are exactly the bits flipped in a Gray code for $\mathbb{F}_2^3$. Further, it may be noted that this is a minimal size parity network for S, and is in fact a fixed-target parity network.

6.3.2 Synthesis with encoded inputs

Given an encoder $E \in \mathrm{L}(\mathbb{F}_2^n, \mathbb{F}_2^m)$, we can use Algorithm 6.1 to synthesize a parity network for some set S with encoded inputs as follows. Recall that a parity network for $S \subseteq \mathbb{F}_2^n$ with

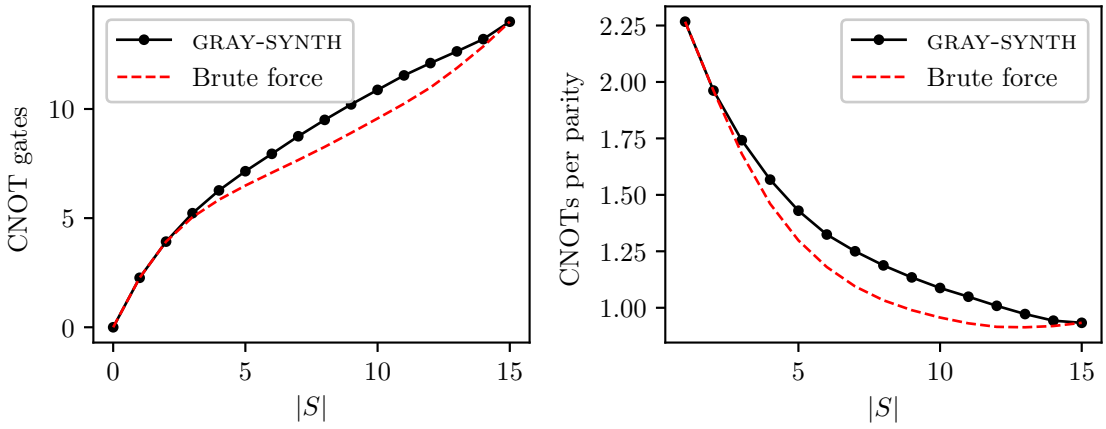


Figure 6.3: Average CNOT counts of parity networks computed by Algorithm 6.1 and brute force minimization.

inputs encoded by E corresponds to a parity network for some set $S' \subseteq \mathbb{F}_2^m$ such that for any $\mathbf{y} \in S$, there exists $\mathbf{w} \in S'$ where $E^T\mathbf{w} = \mathbf{y}$. While finding the minimal such $\mathbf{w}$ would require solving the **NP**-hard Maximum-likelihood decoding problem, we can efficiently compute *some* $\mathbf{w}$ using a generalized inverse of E^T, as in Chapter 4.

It is worth noting that it may be possible to perform additional optimization by optimizing the set S' with a generalized inverse. In particular, it is known [CM09] that the set of *all* solutions to the linear system $A\mathbf{x} = \mathbf{y}$ is given by $\{A^g\mathbf{y} + (I - A^gA)\mathbf{w} \mid \mathbf{w} \in \mathbb{F}_2^m\}$, which may be possible to optimize with classical techniques. We tried brute-force optimizing the set S' for some small instances and found negligible effects on overall CNOT counts, though we leave it as an open question as to whether scalable sub-optimal methods reduce parity network sizes in large benchmarks.

6.4 Evaluation

To evaluate the performance of Algorithm 6.1, it was implemented in Feynman as a CNOT-dihedral synthesis algorithm. The experimental set up was the same as in chapters 4 and 5.

We generated all 4-bit minimal parity networks by brute force search and compared them with the parity networks generated with Algorithm 6.1. Figure 6.3 graphs the results,

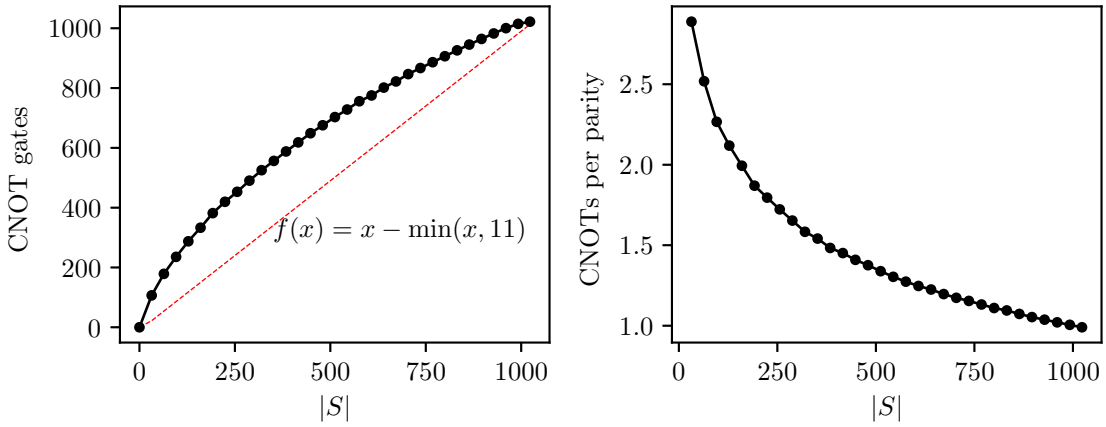


Figure 6.4: Average CNOT counts of parity networks synthesized with Algorithm 6.1 for sets of parities on 10 bits.

with CNOT cost averaged over sets S of parities with the same size. The results show that our algorithm synthesizes optimal or near-optimal networks for small and large sets S, and diverges slightly for sets of parities containing around half of the possible parities. The divergence peaks at $|S| = 8$, exactly half of the 2^4 parities, with Algorithm 6.1 coming within 15% of the minimal number of CNOT gates on average. On examining the structure of optimal parity networks for sets on which Algorithm 6.1 performed poorly, it appears that the optimal parity networks save on CNOT cost by making more judicious use of *shortcuts* – leaving qubits in particular states to flip between distant parities in other bits quickly. In general we found that the optimal results in these cases *were not* achievable just by using the GRAY-SYNTH algorithm with different index expansion orders. An effective synthesis algorithm may then be to combine Algorithm 6.1 for small and large sets with a different heuristic for sets S of size close to 2^{n-1}.

To further examine the scaling of our algorithm, we generated random sets of parities on n bits and used Algorithm 6.1 to synthesize parity networks. Figure 6.4 graphs the results for 10 bits, averaged across 50 randomly generated sets of cardinality $32i$ for each i, against the theoretical lower bound of $|S| - \min(|S|, n + 1)$ corresponding to one CNOT per parity of weight at least 2. The results show a similar curve to the 4 qubit case, with the number of CNOT gates per parity scaled, as expected.

6.4.1 Benchmarks

To evaluate the performance of Algorithm 6.1 on practical quantum circuits, we used it to optimize the CNOT-dihedral subcircuits generated by the phase-folding algorithm of Chapter 4 on the same suite of benchmarks. To assign phase terms to different CNOT-dihedral subcircuits, we performed experiments with two simple strategies – either applying each phase term at the earliest possible location, or the latest. We found that the former "greedy" strategy worked best in almost all cases, and report only those results. We reproduce T counts for comparison with other methods.

Table 6.1 reports the results of our experiments. On average, Algorithm 6.1 resulted in an 22% reduction of CNOT gates, with 43% reduction in the best case. In reality there may be more CNOT reduction on average, as the algorithm performed relatively poorly on the Galois field multipliers, which comprise over a quarter of the benchmarks. Further, only 4 benchmarks took over a second to complete, lending evidence to the scalability of our method. One benchmark, CSLA-MUX_3, did observe an *increase* in CNOT gates of 23% – this appears to be due to our sub-optimal method of generating a pointed parity network from a parity network, combined with the fact that very few T gates cancel (that is, the support of the Fourier expansions synthesized have total size close to the number of T gates in the original circuit). It may be possible to reduce the overhead in this case, and further reduce CNOT counts in the other benchmarks, by synthesizing pointed parity networks directly, rather than synthesizing a parity network followed by a linear permutation.

We compared Algorithm 6.1 against a recent heuristic optimization algorithm by Nam, Ross, Su, Childs and Maslov [NRS+18]. When available the "light" optimization results reported in [NRS+18] are given, as their software is not open-source. Algorithm 6.1 typically results in similar CNOT counts to their circuit optimizer, with Algorithm 6.1 reporting better CNOT counts on some circuits (e.g., VE-Adder_3, CSUM-MUX_9) and worse on others (e.g., CSLA-MUX_3, QCLA-Mod_7). Additionally, it may be noted that in the case of the Galois field multipliers, the reductions Nam *et al.* achieve are from using base circuits with fewer CNOT gates. As their optimizations rely largely on special-purpose synthesis and local rewrites, the techniques are complementary and so it may be possible to combine both to further reduce CNOT counts.

Nam *et al.* also report on the results of a "heavy" optimization algorithm which generally performs slightly better than Algorithm 6.1, with the exception of the VBE-adder_3 and Mod 5_4 benchmarks. This optimization however does not scale to the largest of our benchmark circuits, such as GF(2^{64})-Mult, as a result of the use of local rule-based rewrites to optimize CNOT-dihedral subcircuits. An interesting question is whether first performing re-synthesis with Algorithm 6.1 would reduce run-times for the "heavy" local rewrites.

Table 6.1: CNOT-count optimization results. Original gives the original circuit statistics, Nam *et al.* (L) reports the light optimization results from [NRS+18] (where available), and GRAY-SYNTH gives the results using phase-folding with Algorithm 6.1 for CNOT-dihedral resynthesis. The % reduction in CNOT-count over the base circuit is reported in the last column.

Benchmark	n	Original		Nam *et al.* (L)			GRAY-SYNTH			
		CNOT	T	Time (s)	CNOT	T	Time (s)	CNOT	T	% Red.
Grover_5	9	336	336	–	–	–	0.001	226	154	32.7
Mod 5_4	5	32	28	< 0.001	28	16	0.001	26	16	18.8
VBE-Adder_3	10	80	70	< 0.001	50	24	0.004	46	24	42.5
CSLA-MUX_3	15	90	70	< 0.001	76	64	0.073	111	62	-23.3
CSUM-MUX_9	30	196	196	< 0.001	168	84	0.095	148	84	24.5
QCLA-Com_7	24	215	203	0.001	132	95	0.097	136	94	36.7
QCLA-Mod_7	26	441	413	0.004	302	237	0.145	360	237	18.4
QCLA-Adder_10	36	267	238	0.002	195	162	0.112	214	162	19.9
Adder_8	24	466	399	0.004	331	215	0.165	359	215	23.0
RC-Adder_6	14	104	77	< 0.001	73	47	0.080	71	47	31.7
Mod-Red_21	11	122	119	< 0.001	81	73	0.091	86	73	29.5
Mod-Mult_55	9	55	49	< 0.001	40	35	0.004	40	35	27.3
Mod-Adder_1024	28	2,005	1,995	–	–	–	0.739	1,390	1,011	30.7
Mod-Adder_1048576	58	16,680	16,660	–	–	–	12.272	11,080	7,339	33.6
Cycle 17_3	35	4,532	4,739	–	–	–	2.618	2,968	1,955	37.4
GF(2^4)-Mult	12	115	112	0.001	99	68	0.041	106	68	7.8
GF(2^5)-Mult	15	179	175	0.001	154	115	0.038	163	111	8.9
GF(2^6)-Mult	18	257	252	0.003	221	150	0.055	235	150	8.6
GF(2^7)-Mult	21	349	343	0.004	300	217	0.450	319	217	8.6
GF(2^8)-Mult	24	469	448	0.006	405	264	0.066	428	264	8.7
GF(2^9)-Mult	27	575	567	0.010	494	351	0.076	526	351	8.5
GF(2^{10})-Mult	30	709	700	0.009	609	410	0.081	648	410	8.6
GF(2^{16})-Mult	48	1,837	1,792	0.065	1,581	1,040	0.363	1,691	1,040	7.9
GF(2^{32})-Mult	96	7,292	7,168	1.834	6,299	4,128	5.571	6,636	4,128	9.0
GF(2^{64})-Mult	192	28,861	28,672	58.341	24,765	16,448	114.310	25,934	16,448	10.1
GF(2^{128})-Mult	384	115,069	114,688	1,744.746	98,685	65,664	1,745.724	102,490	65,664	10.9
GF(2^{256})-Mult	768	459,517	458,752	–	–	–	–	–	–	–
Ham_15 (low)	17	259	161	–	–	–	0.043	208	97	19.7
Ham_15 (med)	17	616	574	–	–	–	0.089	357	242	42.0
Ham_15 (high)	20	2,500	2,457	–	–	–	0.376	1,502	1,021	39.9
HWB_6	7	131	105	–	–	–	0.029	110	75	16.0
HWB_8	12	7,508	5,425	–	–	–	1.706	6,861	3,531	8.6
HWB_10	16	36,087	26,579	–	–	–	54.258	32,175	15,921	10.8
HWB_12	20	204,174	159,341	–	–	–	2,849.475	175,805	85,897	13.9
QFT_4	5	48	69	–	–	–	0.005	48	67	0.0
$\Lambda_3(X)$	5	21	21	< 0.001	14	15	< 0.001	14	15	33.3
$\Lambda_3(X)$ (Barenco)	5	28	28	< 0.001	20	16	< 0.001	18	16	35.7
$\Lambda_4(X)$	7	35	35	< 0.001	22	23	0.001	22	23	37.1
$\Lambda_4(X)$ (Barenco)	7	56	56	< 0.001	40	28	< 0.001	36	28	35.7
$\Lambda_5(X)$	9	49	49	< 0.001	30	31	0.003	30	31	38.8
$\Lambda_5(X)$ (Barenco)	9	84	84	< 0.001	60	40	< 0.001	54	40	35.7
$\Lambda_{10}(X)$	19	119	119	< 0.001	70	71	0.071	70	71	41.2
$\Lambda_{10}(X)$ (Barenco)	19	224	224	0.001	160	100	0.029	144	100	35.7
Total										22.0

6.5 Related work

CNOT circuits Previous work regarding CNOT optimization has largely focused on strictly reversible circuits. Iwama, Kambayashi and Yamashita [IKY02] gave some transformation rules which they use to normalize and optimize CNOT-based circuits. More specific to CNOT circuits, Patel, Markov and Hayes [PMH08] gave an algorithm for synthesizing linear reversible circuits which produces circuits of asymptotically optimal size. Their method modifies Gaussian elimination by prioritizing rows which are close in Hamming distance and gives circuits of size at most $O(n^2/\log n)$, coinciding with the known lower bound of $\Theta(n^2/\log n)$ on the worst-case size of CNOT circuits [SPMH02]. We use their algorithm to perform linear reversible synthesis in our implementation.

Clifford+R_Z circuits In the realm of pure quantum circuits, Shende and Markov [SM09] studied the CNOT cost of Toffoli gates. They proved that 6 CNOT gates is minimal for the Toffoli gate, and gave a lower bound of $2n$ CNOT gates for the n qubit Toffoli gate. More recently, Welch, Greenbaum, Mostame and Aspuru-Guzik [WGMAG14] studied the construction of efficient circuits for diagonal unitaries. They used similar insights to the ones we use here, notably the use of the 2^n Walsh functions as a basis for n-qubit diagonal operators which correspond to the parity functions in the Fourier expansions we use to describe CNOT-dihedral circuits. While their main objective was to optimize circuits by constructing approximations of the operator which use fewer Walsh functions, they give a construction of an optimal circuit computing all 2^n Walsh functions. They further give CNOT identities which they use to optimize circuits when not all Walsh functions are used, but give no experimental data as to the effectiveness of these optimizations. In contrast, we present and test an algorithm which directly synthesizes an efficient circuit for a specific set of Walsh functions, rather than construct a circuit for the full set and optimize later.

Part III

Verification

Chapter 7

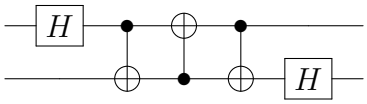
Functional verification

Up to this point, the path integral model of quantum circuits has been used to perform circuit optimization and synthesis. The shift from *qubits and gates* to *paths and phases*, in particular, exposes gates which are physically distant in the circuit but act on the same computational paths. Effectively, the shift in viewpoint automatically "mods out" certain commutation rules, leaving just the computational logic.

A natural question is whether this same effect is helpful for the problem of *verification*. Recall that the identity circuit HH, equal to the identity, has sum-over-paths action

$$HH : |x\rangle \mapsto \frac{1}{2} \sum_{\mathbf{y} \in \mathbb{F}_2^2} e^{\pi i (xy_1 + y_1 y_2)} |y_2\rangle,$$

and in particular $\sum_{\mathbf{y}^2 \in \mathbb{F}_2} e^{\pi i (xy_1 + y_1 y_2)} |y_2\rangle = 2|x\rangle$. We see a similar effect whereby physically separated Hadamard gates which nevertheless act *on the same paths* can be reduced with the above equation. For instance, consider the circuit below:

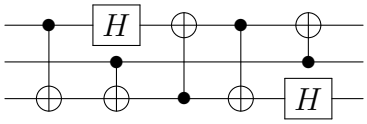


Computing the circuit's sum-over-paths action we see that

$$|x_1\rangle|x_2\rangle \mapsto \frac{1}{2} \sum_{\mathbf{y} \in \mathbb{F}_2^2} e^{\pi i (x_1 y_1 + y_1 y_2)} |x_2\rangle|y_2\rangle,$$

which by writing the right-hand side as $|x_2\rangle \otimes (\frac{1}{2} \sum_{\mathbf{y} \in \mathbb{F}_2^2} e^{\pi i (x_1 y_1 + y_1 y_2)} |y_2\rangle)$, we can use the above equation to rewrite algebraically as $|x_2\rangle|x_1\rangle$. Hence the circuit implements a swap of

two basis states; in an equational *circuit* theory on the other hand, the most natural proof would require rewriting the three CNOT gates as a swap gate, then commuting one of the Hadamard gates through the swap and cancelling. While the "circuit" proof is relatively simple in this case, in the circuit below implementing a swap between the first and third qubits, it is not immediately clear how such an equational proof might proceed.



In comparison, the sum-over-paths action is again

$$|x_1\rangle|x_2\rangle|x_3\rangle \mapsto \frac{1}{2}\sum_{\mathbf{y}\in\mathbb{F}_2^2} e^{\pi i(x_1y_1+y_1y_2)}|x_3\rangle|x_2\rangle|y_2\rangle,$$

which can be rewritten to $|x_3\rangle|x_2\rangle|x_1\rangle$ via the above path integral equation.

In this chapter, we use this insight to address the *functional verification* problem for quantum circuits over the Clifford+$R_k = R_Z(2\pi/2^k)$ gate set. We develop a formal semantic model of such circuits as path integrals, on top of which we build a calculus of rewrite rules which identify and reduce interfering sets of computational paths. We consider this gate set in particular as it will be more convenient to prove certain properties about the size of path integrals – the rewrite system we develop works in principle for circuits over Clifford+R_Z. Recall also that most general quantum algorithms, such as the Quantum Fourier Transform (see, e.g., [NC00]) and those algorithms using it, including Shor's algorithm, are given over the Clifford+R_k gate set.

The model we formalize in this chapter doubles as a natural *specification language* for pure quantum circuits, hence our use of the term *functional* verification – verification of the precise input-output relation of a circuit in relation to a (human-readable) specification. In particular, with some syntactic sugar path integrals correspond directly to the mathematical definition of circuits typically given in textbooks such as [NC00,KLM07], and moreover allow the direct use of classical (functional) programs in specifications of quantum circuits. To evaluate the suitability of path integrals for specification, we perform case studies verifying quantum algorithms against specifications written directly as path integrals.

This work appears in [Amy18] and won the best student paper award at Quantum Physics & Logic 2018 (QPL'18). The verification algorithm is implemented in Feynman and has been used to verify most of the experimental results in chapters 4 and 6.

7.1 The path integral model

Recall from Chapter 3 that we describe a path integral informally as a collection of *paths* Π and amplitude function $\phi : \Pi \to \mathbb{C}$, representing the operator

$$|\mathbf{x}\rangle \mapsto \sum_{\pi:\mathbf{x}\to\mathbf{x}'\in\Pi_{\mathbf{x}}} \phi(\pi)|\mathbf{x}'\rangle.$$

Moreover, it was noted that for Clifford+R_Z circuits, the path integral can be represented by a collection of path variables, a pseudo-Boolean phase function and an affine basis state transformation. In this section, we develop a *concrete*, *computable* representation with a few additions to allow the specification of algorithms with ancillas, as well as our rewrite rules which necessarily take us out of the strict Clifford+R_Z path integral representation.

First recall that phase group for Clifford+R_k circuits is isomorphic to the group of *dyadic fractions* $\mathbb{D} = \{\frac{a}{2^b} | a, b \in \mathbb{Z}\}$, and hence the phase function over such circuits can be represented by a pseudo-Boolean function into $\mathbb{D}$. Moreover, every pseudo-Boolean function $f : \mathbb{F}_2^n \to \mathbb{D}$ has a *unique* presentation as a multilinear polynomial over variables $X = X_1, X_2, \ldots, X_n$. We denote by $\mathbb{D}_M[X] = \mathbb{D}[X]/\langle X_1^2 - X_1, \ldots, X_n^2 - X_n\rangle$ the space of multilinear dyadic polynomials in the variables of X.

In this chapter, we define (concrete) path integrals over the Clifford+R_k gate set with ancillas as a tuple consisting of an input space $S \subseteq \mathbb{F}_2^n$, phase polynomial in n input and m path variables $P \in \mathbb{D}_M[\mathbf{x}, \mathbf{y}]$, and an output function $f : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^n$, represented as (Boolean) polynomial functions.

Definition 7.1.1 (path integral)**.** An n-qubit *path integral* $\xi = (S, P, f)$ consists of

- an *input signature* $S \subseteq \mathbb{F}_2^n$,
- a *phase polynomial* $P \in \mathbb{D}_M[X, Y]$ over input variables $X = X_1, X_2, \ldots, X_n$ and *path variables* $Y = Y_1, Y_2, \ldots, Y_m$, and
- an *output signature* $f : (\mathbf{x}, \mathbf{y}) \mapsto \begin{bmatrix} f_1(\mathbf{x}, \mathbf{y}) \\ f_2(\mathbf{x}, \mathbf{y}) \\ \vdots \\ f_n(\mathbf{x}, \mathbf{y}) \end{bmatrix}$ where each $f_i \in \mathbb{F}_2[X, Y]$.

The *associated operator* of a path integral is the partial linear map U_ξ where for any $\mathbf{x} \in S$,

$$U_\xi : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}\in\mathbb{F}_2^m} e^{2\pi i P(\mathbf{x},\mathbf{y})} |f(\mathbf{x}, \mathbf{y})\rangle.$$

Remark 7.1.2. In contrast to previous chapters, we use P and f to denote the phase polynomial and output functions, respectively, as the phase polynomial is now strictly speaking a polynomial, and the output function f is no longer linear, and instead an arbitrary Boolean function.

A path variable is *internal* if it does not appear in the output signature. We typically write a path integral informally by the action of its associated operator; when writing a path integral by the action of its associated operator, we use Boolean variables and constants to define the input signature. For example, $|x\rangle|0\rangle$ refers to an arbitrary vector in the input signature $S = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$.

Example 7.1.3. Path integral specifications of common quantum gates and circuits are listed below:

$$\begin{aligned}
T &: |x\rangle \mapsto e^{2\pi i \frac{x}{8}} |x\rangle \\
H &: |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \mathbb{F}_2} e^{2\pi i \frac{xy}{2}} |y\rangle \\
\mathsf{Toffoli}_n &: |x_1 x_2 \cdots x_n\rangle \mapsto |x_1 x_2 \cdots (x_n \oplus \textstyle\prod_{i=1}^{n-1} x_i)\rangle \\
\mathsf{Adder}_n &: |\mathbf{x}\rangle|\mathbf{y}\rangle|\mathbf{0}\rangle \mapsto |\mathbf{x}\rangle|\mathbf{y}\rangle|\mathbf{x}+\mathbf{y}\rangle \\
\mathsf{QFT}_n &: |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{\mathbf{y} \in \mathbb{F}_2^n} e^{2\pi i \frac{[\mathbf{x}\cdot\mathbf{y}]}{2^n}} |\mathbf{y}\rangle
\end{aligned}$$

Addition and multiplication of Boolean vectors are interpreted as integer operations at the bit level. In the QFT above, $[\mathbf{x} \cdot \mathbf{y}]$ denotes the integer value of $\mathbf{x} \cdot \mathbf{y}$. For any classical function f, we can lift the polynomial representation of f to a quantum operator via the path integral $|\mathbf{x}\rangle|\mathbf{0}\rangle \mapsto |\mathbf{x}\rangle|f(\mathbf{x})\rangle$. Note that the polynomial representation of a classical function may grow exponentially large, as in the case of addition. A practical implementation of path integrals as a specification language should include a classical sub-language as syntactic sugar for operations on Boolean polynomials.

As a unitary or partial isometry may admit many distinct path integral representations, we define an equivalence between path integrals with the same associated operator.

Definition 7.1.4 (equivalence)**.** Two path integrals ξ_1, ξ_2 are *equivalent*, denoted $\xi_1 \equiv \xi_2$, if and only if their associated operators are equal – that is, $U_{\xi_1} = U_{\xi_2}$.

Non-isometric path integrals are possible in this model, as for instance $|x\rangle \mapsto |0\rangle$ is a syntactically valid path integral. As we are concerned only with the unitary circuit model

and by extension isometric path integrals, we define a notion of well-formedness for path integrals.

Definition 7.1.5 (well-formed)**.** A path integral is well-formed if its associated operator is a (partial) isometry.

In practice, well-formedness is only an issue when writing path integrals directly as specifications, and our verification methods work even when a path integral is not guaranteed to be well-formed.

7.1.1 Composing path integrals

The abstract path integrals of Chapter 3 admit both vertical and horizontal compositions, defined by adding their path variables and composing their phase and output functions. We can define vertical composition for the concrete path integrals we consider here similarly, so that

$$U_{(S,P,f)\otimes(S',P',f')} : |\mathbf{x}\rangle|\mathbf{x}'\rangle \mapsto \sum_{\mathbf{y}\in\mathbb{F}_2^m} \sum_{\mathbf{y}'\in\mathbb{F}_2^{m'}} e^{2\pi i(P(\mathbf{x},\mathbf{y})+P'(\mathbf{x}',\mathbf{y}'))}|f(\mathbf{x},\mathbf{y})\rangle|f'(\mathbf{x}',\mathbf{y}')\rangle.$$

More concretely, we need to reify $P(\mathbf{x},\mathbf{y})+P'(\mathbf{x}',\mathbf{y}')$ and $|f(\mathbf{x},\mathbf{y})\rangle|f'(\mathbf{x}',\mathbf{y}')\rangle$ as polynomials in $n+n'$ input and $m+m'$ path variables, which we can do by shifting the input and path variables by n and m respectively in P' and f':

$$(S,P,f)\otimes(S',P',f') = (S\times S', P + P'\uparrow^{n,m}, (f, f'\uparrow^{n,m})),$$

where for and polynomial P in X and Y, $P\uparrow^{n,m} = P[X_i \leftarrow X_{i+n}][Y_i \leftarrow Y_{i+m}]$ and $P[X\leftarrow P']$ denotes the substitution of X with P' in P.

More care must be taken for horizontal composition, as in general not every such composition of partial isometries is well formed. For instance, composing the path integral $|x\rangle \mapsto |x\rangle$ with $|0\rangle \mapsto |0\rangle$ effectively post-selects on $x=0$. For this reason we require that only *compatible* signatures are composed; in particular, the path integrals (S,P,f) and (S',P',f') are compatible if and only if

$$\{f(\mathbf{x},\mathbf{y}) \mid \mathbf{x}\in S, \mathbf{y}\in\mathbb{F}_2^m\} \subseteq S'.$$

Determining compatibility is non-trivial in general, as it is at least as hard as Boolean satisfiability. In practice, we only ever composes partial isometries with unitaries, and

hence all compositions we consider are trivially compatible. A more complete account of compositions of morphisms in categories of partial isometries can be found in [HB09].

Given two compatible path integrals, (S, P, f) and (S', P', f') their horizontal composition $(S, P, f) \fatsemi (S', P', f')$ may be defined by substituting the input variables X_i of the latter with the outputs f_i of the former. As the phase and output polynomials are defined over different rings ($\mathbb{D}$ and $\mathbb{F}_2$, respectively), when substituting a variable with a Boolean polynomial in the phase we first need to lift it into a *functionally equivalent* polynomial over $\mathbb{D}$. For instance, for all $x, y \in \mathbb{F}_2$, $\frac{1}{4}(x \oplus y) = \frac{1}{4}x + \frac{1}{4}y - \frac{1}{2}xy$. We define the *lifting* of a Boolean polynomial P to a polynomial $\overline{P} \in \mathbb{D}_M[X]$ recursively by

$$\begin{aligned} \overline{X^\alpha} &= X^\alpha, \\ \overline{P \oplus X^\alpha} &= \overline{P} + \overline{X^\alpha} - 2\overline{PX^\alpha}, \end{aligned}$$

where $X^\alpha = X_1^{\alpha_1} X_2^{\alpha_2} \cdots X_n^{\alpha_n}$ for $\alpha \in \mathbb{F}_2^n$ is a multi-index, the first equation uses the inclusion of $\mathbb{F}_2$ in $\mathbb{D}$, and the recursion is terminating since each polynomial on the right hand side has strictly fewer terms. It can be easily verified that the lifting of a Boolean polynomial preserves its action on elements of $\mathbb{F}_2$.

Lemma 7.1.6. *For any n-variable Boolean polynomial P and all $\mathbf{x} \in \mathbb{F}_2^n$, $\overline{P}(\mathbf{x}) = \overline{P(\mathbf{x})}$. That is, for any $\mathbf{x} \in \mathbb{F}_2^n$ the following diagram commutes:*

$$\begin{array}{ccc} \mathbb{F}_2[X] & \xrightarrow{P \mapsto P(\mathbf{x})} & \mathbb{F}_2 \\ \Big\downarrow{\scriptstyle \overline{\cdot}} & & \Big\downarrow{\scriptstyle \overline{\cdot}} \\ \mathbb{D}_M[X] & \xrightarrow{P \mapsto P(\mathbf{x})} & \mathbb{D} \end{array}$$

Proof. By induction. Given $\alpha \in \mathbb{F}_2^n$, clearly for all $\mathbf{x} \in \mathbb{F}_2^n$ we have $\mathbf{x}^\alpha = \mathbf{x}^\alpha$. Moreover, for any Boolean P on n variables and $\alpha \in \mathbb{F}_2^n$,

$$\begin{aligned} (\overline{P \oplus X^\alpha})(\mathbf{x}) &= \overline{P}(\mathbf{x}) + (\overline{X^\alpha})(\mathbf{x}) - 2(\overline{PX^\alpha})(\mathbf{x}) \\ &= \overline{P(\mathbf{x})} + \overline{(X^\alpha)(\mathbf{x})} - 2\overline{(PX^\alpha)(\mathbf{x})} \\ &= \overline{(P \oplus X^\alpha)(\mathbf{x})}. \end{aligned}$$

The last equality follows from the fact that $\overline{P(\mathbf{x})}, \overline{(X^\alpha)(\mathbf{x})} \in \{0, 1\}$ and $\overline{(PX^\alpha)(\mathbf{x})} = 1$ if and only if $\overline{P(\mathbf{x})} = 1 = \overline{(X^\alpha)(\mathbf{x})}$. □

We can now define the functional composition of (compatible) path integrals as

$$(S, P, f) \fatsemi (S', P', f') = (S, P + P' \uparrow^{0,m} [X_i \leftarrow \overline{f_i}], f' \uparrow^{0,m} [X_i \leftarrow f_i]).$$

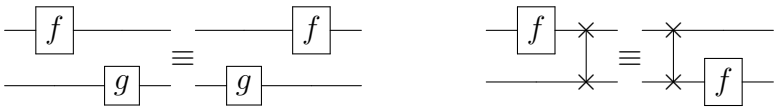
By Lemma 7.1.6, $(P' \uparrow^{0,m} [X_i \leftarrow \overline{f_i}])(\mathbf{x}, \mathbf{y}, \mathbf{y}') = P'(\overline{f(\mathbf{x}, \mathbf{y})}, \mathbf{y}')$, hence

$$U_{(S,P,f)\fatsemi(S',P',f')} : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^{m+m'}}} \sum_{\mathbf{y}\in\mathbb{F}_2^m} \sum_{\mathbf{y}'\in\mathbb{F}_2^{m'}} e^{2\pi i\left(P(\mathbf{x},\mathbf{y})+P'(\overline{f(\mathbf{x},\mathbf{y})},\mathbf{y}')\right)} |f'(f(\mathbf{x}, \mathbf{y}), \mathbf{y}')\rangle.$$

Proposition 7.1.7. *For any well-formed path integrals ξ, ξ', the compositions $\xi \otimes \xi'$ and (if ξ and ξ' are compatible) $\xi \fatsemi \xi'$ are well formed. Moreover,*

$$U_{\xi\otimes\xi'} = U_\xi \otimes U_{\xi'}, \qquad \text{and} \qquad U_{\xi\fatsemi\xi'} = U_{\xi'} U_\xi.$$

Remark 7.1.8. A useful property of path integrals, for the purposes of verification, is that they unify structurally equivalent circuits without the use of *string diagrams* (e.g., [CD11]), which can be difficult to reason about in automated ways [BGK+16]. By this we mean that circuits which are equivalent up to symmetric monoidal laws are *strictly equal* in the path integral picture. For instance, the bifunctoriality law and the naturality of SWAP, stated respectively as the equivalences



are both strict equality in the path integral model. While much progress has been made towards computational methods for diagrammatic reasoning [DL13, BGK+16, CDKW16, GD17], our framework allows us to use standard algebraic tools (e.g., rewriting) *without explicitly managing structural laws.*

Path integrals further unify many *semantic* equivalences of quantum circuits – for instance, the merging of phase gates applied to the same paths as in Chapter 4. In contrast, matrix representations unify *all* equivalences between unitaries, at the expense of exponential space. As was the case in chapters 5 and 6 for CNOT-dihedral circuits, path integrals hence provide an intermediary model, where many equivalences are "modded out" while still remaining efficiently representable.

7.1.2 The path integral semantics

As path integrals admit both a symmetric tensor product and functional composition, we can give a compositional path integral semantics for circuits in $\langle \mathcal{G} \rangle$ by giving path integral

interpretations for each gate in $\mathcal{G}$. As in previous chapters, we use the group laws to "push" all †'s inwards onto gates, since there is no known method of efficiently inverting a path integral. Below we give a path integral interpretation to the Clifford+R_k basis $\{H, \text{CNOT}, R_k\}$[1] for $k > 0$, and their inverses.

Definition 7.1.9. (Clifford+R_k path integral)
The path integral interpretation of an n-qubit circuit C over $\{H, \text{CNOT}, R_k\}$, denoted $[\![C]\!]_p$, is defined as follows:

$$[\![H]\!]_p = (\mathbb{F}_2, \frac{1}{2}X_1Y_1, Y_1)$$
$$[\![R_k]\!]_p = (\mathbb{F}_2, \frac{1}{2^k}X_1, X_1)$$
$$[\![R_k^\dagger]\!]_p = (\mathbb{F}_2, -\frac{1}{2^k}X_1, X_1)$$
$$[\![\text{CNOT}]\!]_p = (\mathbb{F}_2^2, 0, (X_1, X_1 \oplus X_2))$$

The path integral definitions for the $\{H, \text{CNOT}, R_k\}$ basis given in Definition 7.1.9 are somewhat obtuse – a more informal but intuitive definition is given by the sum-over-paths actions of each gate below.

$$H : |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \{0,1\}} e^{2\pi i \frac{xy}{2}} |y\rangle$$
$$R_k : |x\rangle \mapsto e^{2\pi i \frac{x}{2^k}} |x\rangle$$
$$R_k^\dagger : |x\rangle \mapsto e^{2\pi i \frac{-x}{2^k}} |x\rangle$$
$$\text{CNOT} : |x_1 x_2\rangle \mapsto |x_1(x_1 \oplus x_2)\rangle.$$

As a consequence of the above and Proposition 7.1.7 we have the following proposition:

Proposition 7.1.10. *For any circuit C over $\{H, \text{CNOT}, R_k\}$, we have $U_{[\![C]\!]_p} = [\![C]\!]$.*

7.1.3 Computational efficiency

As a composition of linear Boolean functions, it can trivially be observed that each of the outputs of the path integral interpretation of a Clifford+R_k circuit is linear. Moreover, its phase polynomial has degree at most k. To show this, recall the definition of the dyadic *order* of a polynomial (Chapter 5, Section 5.4):

[1] We do not include the X gate in this chapter, as the effect on efficiency is negligible.

Definition 7.1.11. The *order* of a term $\frac{a}{2^b}X^\alpha$ where a is co-prime to 2 and $\alpha \in \mathbb{F}_2^n$ is $b + |\alpha| - 1$. The order of a polynomial $P \in \mathbb{D}_M[X]$, denoted $\mathsf{ord}\,(P)$, is the maximum order of all terms in P.

The definition above is equivalent to the definition in Chapter 5 – however, this version is more intuitive for the dyadic polynomials we consider here.

Example 7.1.12.

$$\mathsf{ord}\left(\frac{1}{2}\right) = 0, \qquad \mathsf{ord}\left(\frac{1}{2}X_1 + \frac{1}{2}X_2\right) = 1, \qquad \mathsf{ord}\left(\frac{1}{2^3}X_2 + \frac{1}{2}X_1X_2X_3\right) = 3$$

An important fact, shown below, is that order is non-increasing with respect to substitution of linear Boolean polynomials.

Lemma 7.1.13. *Let $P \in \mathbb{D}_M[X]$, and let $Q \in \mathbb{F}_2[X]$ be a linear polynomial. Then for any X_i,*

$$\mathsf{ord}\left(P[X_i \leftarrow \overline{Q}]\right) \leq \mathsf{ord}\,(P)$$

Proof. Suppose $Q = \sum_{j\in S} X_j$ for some set S. It is easy to verify that

$$\overline{\sum_{j\in S} X_j} = \sum_{S'\subseteq S} (-2)^{|S'|-1} \prod_{j\in S'} X_j.$$

Substituting $\overline{Q}$ in for X_i we see that for any term $\frac{a}{2^b}X^\alpha$ in P such that $\alpha_i = 1$,

$$\begin{aligned}
\mathsf{ord}\left(\frac{a}{2^b}X^\alpha[X_i \leftarrow \overline{Q}]\right) &= \max_{S'\subseteq S} \mathsf{ord}\left(\frac{a2^{|S'|-1}}{2^b}X^\alpha[X_i \leftarrow \prod_{j\in S'} X_j]\right) \\
&\leq \max_{S'\subseteq S} b - |S'| + |\alpha| + |S'| - 1 \\
&= \mathsf{ord}\left(\frac{a}{2^b}X^\alpha\right).
\end{aligned}$$

□

Since the output function of a Clifford+R_k path integral is strictly linear, by Lemma 7.1.13 composing Clifford+R_k path integrals does not increase the order of the phase polynomial. Moreover, each gate over $\{H, \mathrm{CNOT}, R_k\}$ has phase polynomial of order at most k and denominator at most 2^k, hence we obtain the following results.

Proposition 7.1.14. *The phase polynomial of a (canonical) Clifford+R_k path integral has degree at most k.*

Corollary 7.1.15. *The path integral interpretation of an n-qubit Clifford+R_k circuit C has size polynomial in the volume of C and can be computed in polynomial time.*

On representations of the phase While the representation of the phase as a multilinear polynomial is indeed polynomial in the size of the circuit, at higher levels of the Clifford hierarchy (i.e. large k) the degree of the polynomial can become prohibitively large. Even for the standard Clifford+T gate set, the path integral of a circuit requires space cubic in the volume of the circuit. In practice this makes verification of some larger circuits difficult.

By contrast, representing the phase polynomial with by its Fourier expansion was previously noted to use *linear* space in the circuit volume. This however complicates the process of verification as the Fourier expansion is not necessarily unique. A possible compromise would be to store the Fourier expansion normally, and generate the multilinear form for small subsets on demand.

7.2 A calculus for path integrals

The verification question we're generally concerned with is that of functional verification – *given a circuit C and specification ξ, is $[\![C]\!]_p \equiv \xi$?*. From an automated perspective it is simpler to instead check that the *miter* [YM10] $[\![C^\dagger]\!]_p \circ \xi$ is the identity transformation, but in either case we need a method of efficiently establishing equivalence. To that end, in this section we present a system of reduction rules for path integrals. A key feature of the calculus is that the reduction rules strictly decrease the number of path variables, producing a (not necessarily unique) normal form in polynomial time.

7.2.1 Motivation

Our calculus operates by reducing the number of paths when sets of paths interfere in recognizable ways which we call *interference patterns*. As an illustration, recall once more the identity circuit HH. Computing its canonical path integral we get

$$HH : |x\rangle \mapsto \frac{1}{\sqrt{2^2}} \sum_{y_1,y_2 \in \mathbb{F}_2} e^{2\pi i \frac{xy_1+y_1y_2}{2}} |y_2\rangle.$$

To see that the above path integral is equal to the identity, we can first expand the exponential sum on the right by the values of the internal path variable y_1:

$$\frac{1}{\sqrt{2^2}} \sum_{y_1,y_2 \in \mathbb{F}_2} e^{2\pi i \frac{xy_1+y_1y_2}{2}} |y_2\rangle = \frac{1}{\sqrt{2^2}} \sum_{y_2 \in \mathbb{F}_2} (1 + e^{2\pi i \frac{x+y_2}{2}}) |y_2\rangle$$

Since $e^{\pi}i = -1$, it can be observed that if $x + y_2 = 0 \mod 2$, the two paths corresponding to $y_1 = 0$ and $y_2 = 1$ *constructively* interfere, whereas if $x + y_2 = 1 \mod 2$ they *destructively* interfere. As $\mathbb{F}_2 = x \oplus \mathbb{F}_2 = \{x, 1 \oplus x\}$ for any $x \in \mathbb{Z}$, we can rewrite the sum over $x \oplus \mathbb{F}_2$ and explicitly calculate the interference on either path:

$$\begin{aligned}\frac{1}{\sqrt{2^2}} \sum_{y_2 \in x \oplus \mathbb{F}_2} (1 + e^{2\pi i \frac{x+y_2}{2}})|y_2\rangle &= \frac{1}{2}(1 + e^{2\pi i \frac{x+x}{2}})|x\rangle + \frac{1}{2}(1 + e^{2\pi i \frac{x+1+x}{2}})|1 \oplus x\rangle \\ &= \frac{2}{2}|x\rangle + \frac{0}{2}|1 \oplus x\rangle \\ &= |x\rangle\end{aligned}$$

The reasoning above applies to any situation where an internal path variable y_i only appears with coefficients taken from the Boolean subgroup $\{0, \frac{1}{2}\}$ of $\mathbb{D}/\mathbb{Z}$, as the two branches of y_i are identical, except that $y_i = 1$ path picks up a multiplicative factor of -1 whenever the quotient of $P(\mathbf{x}, \mathbf{y})/y_i$ is odd. Specifically, it can be shown that

$$\frac{1}{\sqrt{2^{m+1}}} \sum_{y_0 \in \mathbb{F}_2} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i \left(\frac{1}{2} y_0 Q(\mathbf{x},\mathbf{y}) + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x}, \mathbf{y})\rangle = \frac{1}{\sqrt{2^{m-1}}} \sum_{\mathbf{y} \in \mathbb{F}_2^m, Q(\mathbf{x},\mathbf{y}) = 0 \mod 2} e^{2\pi i R(\mathbf{x},\mathbf{y})} |f(\mathbf{x}, \mathbf{y})\rangle$$

Note that the polynomial Q is integer-valued and hence can be reduced $\mod 2$ to a Boolean-polynomial – otherwise, the $y_0 = 1$ path could pick up values not in $\{1, -1\}$. In practice, we only perform such reductions when the restricted sum can be reified by solving $Q(\mathbf{x}, \mathbf{y}) = 0 \mod 2$ for some y_i, as we did above with $y_2 = x$.

7.2.2 Reduction rules

Figure 7.1 gives the rules of our calculus. We write $\xi \longrightarrow \xi'$ to denote that ξ reduces to ξ', and denote by $\longrightarrow^*$ the transitive closure of $\longrightarrow$. In all rules, Y_i is an internal path variable, and the quotient $Q \in \mathbb{F}_2[X, Y]$ means Q is an integer valued polynomial reduced $\mod 2$, since for all $\mathbf{x}, \mathbf{y}$ and any integer valued polynomial $Q \in \mathbb{D}_M[X, Y]$,

$$e^{2\pi i \left(\frac{1}{2} Y_i Q(\mathbf{x},\mathbf{y})\right)} = e^{2\pi i \left(\frac{1}{2} Y_i (Q(\mathbf{x},\mathbf{y}) \mod 2)\right)}.$$

The rules were developed by translating known circuit identities into path integrals, then minimizing the identities to obtain simple interference patterns which 1) strictly reduce the number of path variables, and 2) can be efficiently matched. We found that most common Clifford+T equalities reduce to a small set of rules – in particular, the [HH] rule derived

$$\frac{P = \frac{1}{4}Y_i + \frac{1}{2}Y_iQ + R, \qquad Q \in \mathbb{F}_2[X,Y] \quad Y_i \text{ does not appear in } Q,\ R \text{ or } f}{(S,P,f) \longrightarrow (S, \frac{1}{8} - \frac{1}{4}\overline{Q} + R, f)} \qquad [\omega]$$

$$\frac{P = \frac{1}{2}Y_i(Y_j + Q) + R, \qquad Q \in \mathbb{F}_2[X,Y] \quad Y_i \text{ does not appear in } Q,\ R \text{ or } f \quad Y_j \text{ does not appear in } Q}{(S,P,f) \longrightarrow (S, R[Y_j \leftarrow \overline{Q}], f[Y_j \leftarrow Q])} \qquad [\mathsf{HH}]$$

Figure 7.1: Path integral reduction rules

from the equality $HH = I$ as described above is sufficient for the vast majority of path integral reductions. The $[\omega]$ rule arises from the identity $(SH)^3 = e^{\frac{2\pi i}{8}} I = \omega I$. It can be observed that $[\omega]$ eliminates one variable (Y_i), while [HH] eliminates two path variables (Y_i and Y_j).

Proposition 7.2.1 (Correctness)**.** *If $\xi \longrightarrow^* \xi''$, then $\xi \equiv \xi'$.*

Proof. We verify each rewrite rule by direct calculation on the right-hand side of the associated operator. Recall that by Lemma 7.1.6, $\overline{Q}(\mathbf{x},\mathbf{y}) = \overline{Q(\mathbf{x},\mathbf{y})}$.

$$\begin{aligned}
[\omega]: \quad & \frac{1}{\sqrt{2^{m+1}}} \sum_{y_i \in \mathbb{F}_2} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{4}y_i + \frac{1}{2}y_iQ(\mathbf{x},\mathbf{y}) + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle \\
&= \frac{1}{\sqrt{2^{m+1}}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} \left(1 + e^{2\pi i\left(\frac{1}{4} + \frac{1}{2}Q(\mathbf{x},\mathbf{y})\right)}\right) e^{2\pi i R(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle \\
&= \begin{cases} \frac{1}{\sqrt{2^{m+1}}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} (1+i)\, e^{2\pi i R(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle & \text{if } \overline{Q(\mathbf{x},\mathbf{y})} = 0 \\ \frac{1}{\sqrt{2^{m+1}}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} (1-i)\, e^{2\pi i R(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle & \text{if } \overline{Q(\mathbf{x},\mathbf{y})} = 1 \end{cases} \\
&= \begin{cases} \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{8} + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle & \text{if } \overline{Q(\mathbf{x},\mathbf{y})} = 0 \\ \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{8} - \frac{1}{4} + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle & \text{if } \overline{Q(\mathbf{x},\mathbf{y})} = 1 \end{cases} \\
&= \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{8} - \frac{1}{4}\overline{Q}(\mathbf{x},\mathbf{y}) + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle
\end{aligned}$$

[HH]: $$\frac{1}{\sqrt{2^{m+2}}} \sum_{y_i,y_j \in \mathbb{F}_2} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i \left(\frac{1}{2} y_i (y_j + Q(\mathbf{x},\mathbf{y})) + R(\mathbf{x},\mathbf{y},y_j)\right)} |f(\mathbf{x},\mathbf{y},y_j)\rangle$$

$$= \frac{1}{\sqrt{2^{m+2}}} \sum_{y_j \in \mathbb{F}_2} \sum_{\mathbf{y} \in \mathbb{F}_2^m} \left(1 + e^{\pi i (y_i + \overline{Q(\mathbf{x},\mathbf{y})})}\right) e^{\pi i R(\mathbf{x},\mathbf{y},y_j)} |f(\mathbf{x},\mathbf{y},y_j)\rangle$$

$$= \frac{1}{\sqrt{2^{m+2}}} \sum_{y_j \in \mathbb{F}_2} \sum_{\mathbf{y} \in \mathbb{F}_2^m} \left(1 + e^{\pi i y_i}\right) e^{\pi i R(\mathbf{x},\mathbf{y},y_j + \overline{Q(\mathbf{x},\mathbf{y})})} |f(\mathbf{x},\mathbf{y},y_j + Q(\mathbf{x},\mathbf{y}))\rangle$$

$$= \frac{2}{\sqrt{2^{m+2}}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i R(\mathbf{x},\mathbf{y},\overline{Q}(\mathbf{x},\mathbf{y}))} |f(\mathbf{x},\mathbf{y},Q(\mathbf{x},\mathbf{y}))\rangle$$

$$= \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{F}_2^m} e^{2\pi i \left(R[y_j \leftarrow \overline{Q}]\right)(\mathbf{x},\mathbf{y})} |\left(f[y_j \leftarrow Q]\right)(\mathbf{x},\mathbf{y})\rangle$$

□

It is a trivial fact that the calculus is terminating, as every rule reduces the number of path variables. Moreover, each rewrite rule can be matched against in polynomial time, hence every path integral reduces to a normal form in polynomial time.

Proposition 7.2.2 (Strong normalization)**.** *Every sequence of rewrites terminates with an irreducible path integral. The sequence is linear in the number of path variables m and for an n-qubit path integral takes time polynomial in n and m.*

7.2.3 Examples

To illustrate our rewrite system, we give examples below. We write path integrals by their associated operators for clarity.

Example 7.2.3. Recall that the standard implementation of the Toffoli gate over Clifford+T has the sum-over-paths action

$$\mathsf{Toffoli}_3 : |x_1x_2x_3\rangle \mapsto \frac{1}{\sqrt{2^2}} \sum_{y_1,y_2 \in \mathbb{F}_2} e^{2\pi i \frac{1}{2}(x_3y_1 + x_1x_2y_1 + y_1y_2)} |x_1x_2y_2\rangle.$$

We can verify that this is equivalent to the functional specification

$$|x_1x_2x_3\rangle \mapsto |x_1x_2(x_3 \oplus x_1x_2)\rangle$$

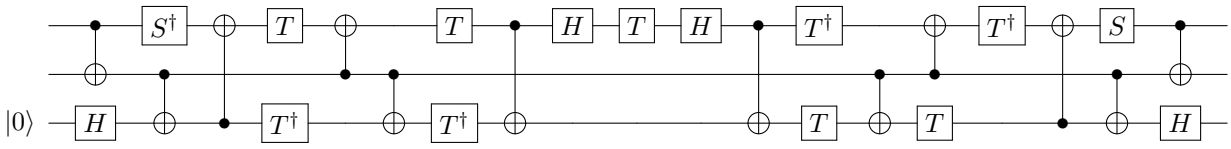
with the following sequence of reductions and algebraic manipulations:

$$\begin{aligned}
|x_1x_2x_3\rangle &\mapsto \frac{1}{\sqrt{2^2}} \sum_{y_1,y_2\in\mathbb{F}_2} e^{2\pi i\frac{1}{2}(x_3y_1+x_1x_2y_1+y_1y_2)}|x_1x_2y_2\rangle \\
&\mapsto \frac{1}{\sqrt{2^2}} \sum_{y_1,y_2\in\mathbb{F}_2} e^{2\pi i\frac{1}{2}y_1(y_2+x_3+x_1x_2)}|x_1x_2y_2\rangle \\
&\mapsto |x_1x_2(x_3\oplus x_1x_2)\rangle \qquad \text{[HH]}
\end{aligned}$$

Example 7.2.4. The controlled-T gate can be specified as the path integral

$$\mathsf{controlled\text{-}T} : |x_1x_2\rangle \mapsto e^{2\pi i\frac{x_1x_2}{8}}|x_1x_2\rangle.$$

An implementation of the controlled-T gate over Clifford+T is given below:



Computing the canonical path integral as a partial isometry with the third qubit in the initial state $|0\rangle$ we get

$$\begin{aligned}
|x_1x_2\rangle|0\rangle &\mapsto \frac{1}{\sqrt{2^4}} \sum_{\mathbf{y}\in\mathbb{F}_2^4} e^{2\pi i\frac{1}{8}(4x_1x_2y_1+4x_1y_2+4y_1y_2+y_2+4y_2y_3+4x_1x_2y_3+4x_1y_4+4y_3y_4+4x_1x_2)}|x_1x_2y_4\rangle \\
&\mapsto \frac{1}{\sqrt{2^4}} \sum_{\mathbf{y}\in\mathbb{F}_2^4} e^{2\pi i\left(\frac{1}{2}y_1(y_2+x_1x_2)+\frac{1}{8}(4x_1y_2+y_2+4y_2y_3+4x_1x_2y_3+4x_1y_4+4y_3y_4+4x_1x_2)\right)}|x_1x_2y_4\rangle \\
&\mapsto \frac{1}{\sqrt{2^2}} \sum_{y_3,y_4\in\mathbb{F}_2} e^{2\pi i\frac{1}{8}(4x_1x_2+x_1x_2+4x_1x_2y_3+4x_1x_2y_3+4x_1y_4+4y_3y_4+4x_1x_2)}|x_1x_2y_4\rangle \qquad \text{[HH]} \\
&\mapsto \frac{1}{\sqrt{2^2}} \sum_{y_3,y_4\in\mathbb{F}_2} e^{2\pi i\left(\frac{1}{2}y_3y_4+\frac{1}{8}(x_1y_4+x_1x_2)\right)}|x_1x_2y_4\rangle \\
&\mapsto e^{2\pi i\frac{x_1x_2}{8}}|x_1x_2\rangle|0\rangle \qquad \text{[HH]}
\end{aligned}$$

Hence the above circuit implements the controlled-T gate with a 0-initialized ancilla, and moreover returns the ancilla to its original state.

If we had tried to verify the above circuit with an arbitrary state $|x_3\rangle$ for the third qubit, it would have failed since $|x_1x_2\rangle|1\rangle \not\mapsto e^{2\pi i\frac{x_1x_2}{8}}|x_1x_2\rangle|1\rangle$. This illustrates the necessity of modelling the semantics of qubit initialization for practical verification tasks.

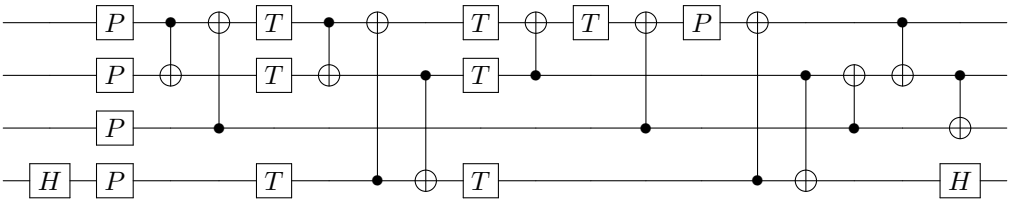
Example 7.2.5. To show the use of the $[\omega]$ rule, we reduce the circuit $(SH)^3$ to the ω constant.

$$\begin{aligned}
(\mathsf{SH})^3 : |x\rangle &\mapsto \frac{1}{\sqrt{2}^3} \sum_{y_1,y_2,y_3 \in \mathbb{F}_2} e^{2\pi i \frac{1}{8}(4xy_1+6y_1+4y_1y_2+6y_2+4y_2y_3+6y_3+1)} |y_3\rangle \\
&\mapsto \frac{1}{\sqrt{2}^3} \sum_{y_1,y_2,y_3 \in \mathbb{F}_2} e^{2\pi i \left(\frac{1}{2}(\frac{1}{2}y_1+y_1(y_2 \oplus 1 \oplus x))+\frac{1}{8}(6y_2+4y_2y_3+6y_3+1)\right)} |y_3\rangle \\
&\mapsto \frac{1}{\sqrt{2}^2} \sum_{y_2,y_3 \in \mathbb{F}_2} e^{2\pi i \frac{1}{8}(1-2(y2+1+x-2y_2-2x-2y_2x+4y_2x)+6y_2+4y_2y_3+6y_3+1)} |y_3\rangle && [\omega] \\
&\mapsto \frac{1}{\sqrt{2}^2} \sum_{y_2,y_3 \in \mathbb{F}_2} e^{2\pi i \frac{1}{8}(2x+4y_2x+4y_2y_3+6y_3)} |y_3\rangle \\
&\mapsto e^{2\pi i \frac{1}{8}(2x+6x)} |x\rangle && [\mathsf{HH}] \\
&\mapsto \omega |x\rangle.
\end{aligned}$$

Example 7.2.6. The one-bit full adder has the reversible path integral specification

$$|x_1x_2x_3x_4\rangle \mapsto |x_1(x_1 \oplus x_2)(x_1 \oplus x_2 \oplus x_3)(x_1x_2 \oplus x_1x_3 \oplus x_2x_3 \oplus x_4)\rangle.$$

The implementation below over Clifford+T was obtained by using the Reed-Muller decoding method of Chapter 5 to reduce the number of T gates from the standard implementation using two Toffoli gates.



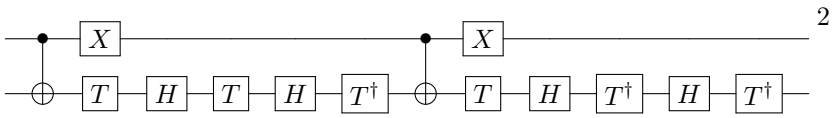
We can verify that this circuit implements the one-bit adder specification as follows:

$$\begin{aligned}
|x_1x_2x_3x_4\rangle &\mapsto \frac{1}{\sqrt{2}^2} \sum_{y_1,y_2 \in \mathbb{F}_2} e^{2\pi i \frac{1}{2}(y_1y_2+y_1x_1x_2+y_1x_1x_3+y_1x_2x_3+y_1x_4)} |x_1(x_1 \oplus x_2)(x_1 \oplus x_2 \oplus x_3)y_2\rangle \\
&\mapsto \frac{1}{\sqrt{2}^2} \sum_{y_1,y_2 \in \mathbb{F}_2} e^{2\pi i \frac{1}{2}y_1(y_2+x_1x_2+x_1x_3+x_2x_3+x_4)} |x_1(x_1 \oplus x_2)(x_1 \oplus x_2 \oplus x_3)y_2\rangle \\
&\mapsto |x_1(x_1 \oplus x_2)(x_1 \oplus x_2 \oplus x_3)(x_1x_2 \oplus x_1x_3 \oplus x_2x_3 \oplus x_4)\rangle && [\mathsf{HH}, \mathsf{Elim}]
\end{aligned}$$

In this case, verification works because of the normalization of the phase polynomial due to its presentation as a multilinear polynomial, rather than the Fourier expansion.

7.3 Completeness

While our calculus computes a normal form in polynomial time, the normal forms are not necessarily unique[2] and hence our reduction system is incomplete. For instance, the Clifford+T identity



from [SB16] gives the irreducible path integral $|x_1x_2\rangle \mapsto \frac{1}{\sqrt{2^8}} \sum_{\mathbf{y}\in\mathbb{F}_2^8} e^{2\pi i \frac{1}{8}P(\mathbf{x},\mathbf{y})}|x_1y_8\rangle$ with phase polynomial

$$\begin{aligned}P(\mathbf{x},\mathbf{y}) = 2 &+ 6x_1x_2 + x_2 + y_1 + 4y_1(x_1 + x_2 + y_2) + 6y_2 + 4y_2y_3 + 2y_2x_1 + 3y_3 + 4y_3(x_1 + y_4)\\ &+ 4y_4y_5 + 6y_4x_1 + y_5 + 4y_5(x_1 + y_6) + 6y_6 + 4y_6y_7 + 2y_6x_1 + 3y_7 + 4y_7(x_1 + y_8) + 7y_8.\end{aligned}$$

A complete verification procedure could proceed by explicitly expanding the values of remaining variables in the path integral after all possible reductions have been made, and then checking equivalence to the identity transformation. In practice we found that this is generally not necessary, as our calculus, along with some additional observations, is sufficient to prove or disprove equivalence for the majority of circuits. Moreover, these heuristics combined with path integral reductions give a complete, polynomial-time procedure for determining equivalence of Clifford group circuits.

7.3.1 Heuristics

Isometry restrictions Our first heuristic reduces the number of path variables in a *well-formed* path integral when checking equivalence. Specifically, we denote by $\xi|_{f(\mathbf{x},\mathbf{y})=\mathbf{x}}$ the restriction of ξ to solutions $\mathbf{x} \in \mathbb{F}_2^n, \mathbf{y} \in \mathbb{F}_2^m$ such that $f(\mathbf{x},\mathbf{y}) = \mathbf{x}$, which we can write as the restricted sum

$$|\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}\in\mathbb{F}_2^m, f(\mathbf{x},\mathbf{y})=\mathbf{x}} e^{2\pi i P(\mathbf{x},\mathbf{y})}|\mathbf{x}\rangle.$$

Effectively, the sum $\frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}\in\mathbb{F}_2^m, f(\mathbf{x},\mathbf{y})=\mathbf{x}} e^{2\pi i P(\mathbf{x},\mathbf{y})}$ gives the amplitude of the basis state $|\mathbf{x}\rangle$ in the output for a given input state $|\mathbf{x}\rangle$. If the path integral ξ is well-formed (i.e. normalized), then this sum will be equal to 1 exactly if ξ is the identity transformation. We sum this up in the lemma below:

[2]Uniqueness would imply that equivalence checking of reversible Boolean circuits is in **P**. As this problem is **co-NP**-complete, uniqueness of our normal forms would indeed imply **P** = **co-NP**.

Lemma 7.3.1. *Suppose* $U_\xi : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}\in\mathbb{F}_2^m} e^{2\pi i P(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle$ *is a well-formed path integral. Then* ξ *implements the identity transformation if and only if* $\xi|_{f(\mathbf{x},\mathbf{y})=\mathbf{x}}$ *implements the identity transformation.*

Note that Lemma 7.3.1 doesn't hold if ξ is not well-formed, as U_ξ may not be an isometry and so it may be that $U_\xi|\mathbf{x}\rangle = \alpha|\mathbf{x}\rangle + \beta|\psi\rangle$ for some residual state $|\psi\rangle$. To reify the restricted path integral $\xi|_{f(\mathbf{x},\mathbf{y})=\mathbf{x}}$ we find path variable substitutions which give $f_i = X_i$ – in particular, if for some index i we have $f_i = Y_i \oplus Q$ where Y_i doesn't appear in Q, we can substitute $X_i + Q$ for Y_i to get $f_i = X_i$ and remove Y_i from the sum. Any restrictions which can't be reified are simply ignored. In practice this results in a significant simplification for some circuits, instantly removing up to n path variables.

Remark 7.3.2. Restricting a path integral to solutions of the form $f(\mathbf{x},\mathbf{y}) = \mathbf{x}$ may break normalization. In particular, restricting

$$|x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y\in\mathbb{F}_2} |y\rangle$$

to solutions $y = x$ gives the sum-over-paths action

$$|x\rangle \mapsto \frac{1}{\sqrt{2}}|x\rangle,$$

which has no path variables but a transition amplitude of $\frac{1}{\sqrt{2}}$ and hence is *not* the identity. For this reason when using isometry restrictions a normalization factor is required along with the path integral to ensure the output amplitude of the $|\mathbf{x}\rangle$ state is 1.

Disproving equivalence As the reduction rules of Figure 7.1 only suffice to prove *positive* results (i.e. equivalence), when no more reductions are possible we attempt to apply an observation that was found to be effective for proving that a path integral ξ is *not* the identity. In particular, recall that

$$\frac{1}{\sqrt{2^{m+1}}} \sum_{y_0\in\mathbb{F}_2} \sum_{\mathbf{y}\in\mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{2}y_0 Q(\mathbf{x},\mathbf{y}) + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle = 0$$

if $Q(\mathbf{x},\mathbf{y}) = 1 \mod 2$. If Q is a non-zero Boolean polynomial in *only* input variables X_i, then there necessarily exists a solution $\mathbf{x} \in \mathbb{Z}^n$ such that $Q(\mathbf{x}) = 1 \mod 2$, and in particular

$$\sum_{\mathbf{y}\in\mathbb{F}_2^m} e^{2\pi i\left(\frac{1}{2}y_0 Q(\mathbf{x},\mathbf{y}) + R(\mathbf{x},\mathbf{y})\right)} |f(\mathbf{x},\mathbf{y})\rangle = \frac{1}{\sqrt{2^{m+1}}} \sum_{\mathbf{y}\in\mathbb{F}_2^m} (1-1) e^{2\pi i R(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle = 0.$$

We sum this up in the following lemma.

Lemma 7.3.3. *Suppose $\xi = (S, \frac{1}{2}Y_iQ + R, f)$ where Y_i does not appear in Q, R or f and Q is a non-zero integer-valued polynomial not referencing any path variables. Then $U_\xi \neq I$.*

Hence we can use a variant of [HH] where Q references only input variables to prove *negative* results – i.e. that a path integral is *not* the identity.

7.3.2 Clifford completeness

We can now show that together with the previous heuristics, our path integral reductions are complete for deciding equivalence of Clifford group circuits. Recall that over the Clifford group, the path integral interpretation of a circuit has phase polynomial of order at most 2. Our proof of completeness rests on the fact that progress can always be made if the phase polynomial is at most second-order.

Lemma 7.3.4 (Clifford progress & preservation)**.** *If $\xi = (S, P, f)$ is a path integral where $\mathsf{ord}(P) \leq 2$ and no path variables appear in f. Then either there exists $\xi' = (S', P', f')$ such that $\xi \longrightarrow \xi'$ where $\mathsf{ord}(P') \leq 2$ and no path variables appear in f', or $U_\xi \neq I$.*

Proof. Since P is at most second-order, we can write $P = Y_iQ + R$ for some internal path variable Y_i and polynomials Q, R. Moreover, Q is at most first-order and has the form

$$a\frac{1}{4} + b\frac{1}{2}Q'$$

for some $a, b \in \mathbb{F}_2$ and linear Boolean polynomial Q'. We have 2 cases to consider, corresponding to the [HH] and [ω] rules respectively.

Case 1: $a = 0, b = 1$**.** If the polynomial Q' contains a path variable Y_j, then $Q' = Y_j + Q''$ and $\xi \longrightarrow_{[\mathsf{HH}]} \xi'$. Further, by Lemma 7.1.13, $\mathsf{ord}\left(R[Y_j \leftarrow \overline{Q''}]\right) \leq \mathsf{ord}(R) \leq 2$ and ξ' has only internal paths since $Y_j \notin f$.

If on the other hand Q' only contains input variables, by Lemma 7.3.3 $U_\xi \neq I$.

Case 2: $a = 1$**.** The sum matches the left hand side of [ω], hence $\xi \longrightarrow_{[\omega]} \xi'$. Further, by Lemma 7.1.13

$$\mathsf{ord}(P') = \mathsf{ord}\left(\frac{1}{8} - \frac{1}{4}\overline{Q''} + R\right) = \max\left\{\mathsf{ord}\left(\frac{1}{8}\right), \mathsf{ord}\left(\frac{1}{4}\overline{Q''}\right), \mathsf{ord}(R)\right\} = 2.$$

□

Corollary 7.3.5. *If $C \in \langle H, \mathrm{CNOT}, S \rangle$, then $[\![C]\!] = I$ can be decided in time polynomial in the volume of C.*

Proof. Since $[\![C]\!]_p$ is well-formed, by Lemma 7.3.1 it suffices to check whether $[\![C]\!]_p|_{f(\mathbf{x},\mathbf{y})=\mathbf{x}}$ is the identity. Further, as $f(\mathbf{x}, \mathbf{y})$ is linear, we can compute via Gaussian elimination a solution $\mathbf{y}$ so that $f(\mathbf{x}, \mathbf{y}) = \mathbf{x}$ for any $\mathbf{x}$ – if no such solution exists, $[\![C]\!] \neq I$. Since each f_i is linear, $\mathsf{ord}\,(P[y_i \leftarrow f_i]) \leq \mathsf{ord}\,(P) \leq 2$, hence by Lemma 7.3.4 and Proposition 7.2.2, either $[\![C]\!]_p|_{f(\mathbf{x},\mathbf{y})=\mathbf{x}}$ reduces to the identity in polynomial-time or ξ' is not the identity. □

7.4 Case studies

To evaluate the performance of our reduction system for the task of verification, we performed two sets of experiments – equivalence checking of the optimized benchmark circuits from chapters 4 and 6, and the verification of circuits directly against functional specifications given as path integrals. Again, the experiments were performed in Debian Linux running on a quad-core 64-bit Intel Core i7 2.40 GHz processor and 8 GB RAM.

7.4.1 Translation validation

Translation validation is an important tool for verifying that the transformations a compiler performs do not change the semantics of an input program. While it is generally desirable to prove that a compiler operates correctly on *all* input programs, as we discuss in the next chapter, in many cases this is infeasible since the best optimizations are typically difficult to formally verify.

We used our algorithm to verify the benchmark optimization results of chapters 4 and 6. We give the results of the verification of the GRAY-SYNTH algorithm of Chapter 6 in Table 7.1, as it includes phase-folding and is the harder optimization problem. All but 7 of the benchmark circuits were successfully verified, with the remaining 7 benchmarks running out of memory with a 6 GB limit. The high memory usage may be mitigated in the future by switching to a linear-space representation of the phase polynomial. The largest (completed) benchmark GF(2^{32}), containing 96 bits, 252 path variables and over 25,000 gates completed in under 10 minutes, with the remainder all taking under a minute.

To test the algorithm's ability to disprove equivalence, we also performed the verification of the optimized benchmark circuits after removing a randomly selected gate. Again, all but 7 benchmarks were proven to be not equivalent, with the negative verification results taking about the same amount of time as positive results.

Table 7.1: Optimization validation results. n lists the number of qubits, m gives the number of path variables, and Clifford and T give the number of respective gates. Times for positive and negative verification measure the time to prove equivalence or non-equivalence, respectively. Benchmarks with no timing results ran out of memory.

Benchmark	n	m	Clifford	T	Time (s)	
					Positive	Negative
Grover_5	9	200	1,515	490	0.973	0.988
Mod 5_4	5	12	66	44	0.005	0.028
VBE-Adder_3	10	20	167	94	0.026	0.028
CSLA-MUX_3	15	40	289	132	0.099	0.055
CSUM-MUX_9	30	56	638	280	0.270	0.270
QCLA-Com_7	24	74	1237	297	0.530	0.543
QCLA-Mod_7	26	164	1641	650	9.446	10.517
QCLA-Adder_10	36	100	627	400	0.674	0.683
Adder_8	24	160	1419	614	1.968	2.018
RC-Adder_6	14	44	322	124	0.080	0.090
Mod-Red_21	11	60	392	192	0.110	0.119
Mod-Mult_55	9	28	180	84	0.028	0.009
Mod-Adder_1024	28	660	4,363	3,006	21.362	21.588
Mod-Adder_1048576	58	4,832	40,318	23,999	–	–
Cycle 17_3	35	1,366	9,172	6,694	–	–
GF(2^4)-Mult	12	28	263	180	0.063	0.061
GF(2^5)-Mult	15	36	393	286	0.143	0.141
GF(2^6)-Mult	18	44	559	402	0.279	0.291
GF(2^7)-Mult	21	52	731	560	0.501	0.527
GF(2^8)-Mult	24	60	975	712	0.837	0.881
GF(2^9)-Mult	27	68	1179	918	1.304	1.369
GF(2^{10})-Mult	30	76	1,475	1,110	1.958	0.327
GF(2^{16})-Mult	48	124	3,694	2,832	16.028	17.539
GF(2^{32})-Mult	96	252	14,259	11,296	430.883	436.521
GF(2^{64})-Mult	192	508	55,408	45,120	–	–
GF(2^{128})-Mult	384	1,020	231,318	180,352	–	–
GF(2^{256})-Mult	–	–	–	–	–	–
Hamming_15 (low)	17	76	612	158	0.367	0.168
Hamming_15 (med)	17	184	1,251	762	1.390	1.430
Hamming_15 (high)	20	716	5,332	3,462	24.360	24.303
HWB_6	7	52	369	180	0.200	0.207
HWB_8	12	2,282	17,583	8,895	–	–
HWB_10	16	10,480	88,230	42,500	–	–
HWB_12	20	56,824	500,974	245,238	–	–
QFT_4	5	84	218	136	0.084	0.089
$\Lambda_3(X)$	5	12	52	36	0.004	0.011
$\Lambda_3(X)$ (Barenco)	5	12	66	44	0.007	0.046
$\Lambda_4(X)$	7	20	87	58	0.009	0.008
$\Lambda_4(X)$ (Barenco)	7	20	127	84	0.014	0.024
$\Lambda_5(X)$	9	18	112	80	0.015	0.017
$\Lambda_5(X)$ (Barenco)	9	28	160	124	0.030	0.031
$\Lambda_{10}(X)$	19	68	297	190	0.110	0.111
$\Lambda_{10}(X)$ (Barenco)	19	68	493	324	0.219	0.210

Table 7.2: Results of verifying formally specified quantum algorithms.

Algorithm	n	m	Clifford	T	Time (s)	
					Positive	Negative
$\mathsf{Toffoli}_{50}$	97	190	855	665	1.084	1.064
$\mathsf{Toffoli}_{100}$	197	390	1,755	1,365	5.566	5.275
Maslov_{50}	74	192	481	384	0.801	0.778
Maslov_{100}	149	392	981	784	3.987	3.983
Adder_{8}	40	56	334	196	0.142	0.143
Adder_{16}	80	120	710	420	25.527	92.607
QFT_{16}	16	16	256	–	1.250	1.335
QFT_{31}	31	31	961	–	16.929	15.295
Hidden $\mathsf{Shift}_{20,4}$	20	60	5,254	56	1.067	0.862
Hidden $\mathsf{Shift}_{40,5}$	40	120	6,466	70	3.383	2.826
Hidden $\mathsf{Shift}_{60,10}$	60	180	12,784	140	13.217	12.351
Symbolic $\mathsf{Shift}_{20,4}$	40	60	5,296	56	1.859	1.849
Symbolic $\mathsf{Shift}_{40,5}$	80	120	6,638	70	6.953	7.905
Symbolic $\mathsf{Shift}_{60,10}$	120	180	12,804	140	35.583	29.614

7.4.2 Verifying quantum algorithms

To evaluate path integrals as a tool for *functional specification* as well as verification, we implemented and verified several quantum algorithms (both without and with errors) directly against specifications given as path integrals. Table 7.2 reports the results of our experiments, and we describe the algorithms and implementations below.

Reversible functions We implemented and verified a number of known algorithms for reversible functions. In particular, we performed verifications of Clifford+T implementations of the generalized Toffoli and (out-of-place) addition functions,

$$\mathsf{Toffoli}_n : |x_1 x_2 \dots x_n\rangle \mapsto |x_1 x_2 \dots (x_n \oplus x_1 x_2 \dots x_{n-1})\rangle,$$
$$\mathsf{Adder}_n : |\mathbf{x}\rangle|\mathbf{y}\rangle|\mathbf{0}\rangle \mapsto |\mathbf{x}\rangle|\mathbf{y}\rangle|\mathbf{x}+\mathbf{y}\rangle.$$

The translation of the above specifications to path integrals required translating the outputs into Boolean polynomials. For the $\mathsf{Toffoli}_n$ algorithm, this translation was trivial; for Adder_n, a set of polynomials giving the bits of $\mathbf{x}+\mathbf{y}$ was generated by performing binary addition on symbolic vectors. In comparison to writing a reversible circuit or reversible addition program, this translation – being strictly classical in nature and implementing a familiar algorithm – was (empirically) easier to code, avoiding the difficulty of space management

and reversible cleanup. Moreover, being a classical program this translation can be tested or otherwise verified by known methods.

For the circuit implementations, two versions of the n-bit Toffoli algorithm were used – the standard decomposition into $2(n-3)+1$ Toffoli gates and $n-3$ ancillas, and the Maslov decomposition [Mas16] using *relative phase* Toffolis and $\lceil \frac{n-3}{2} \rceil$ ancillas. For either implementation we were able to verify up to 100 bit Toffoli gates in just seconds.

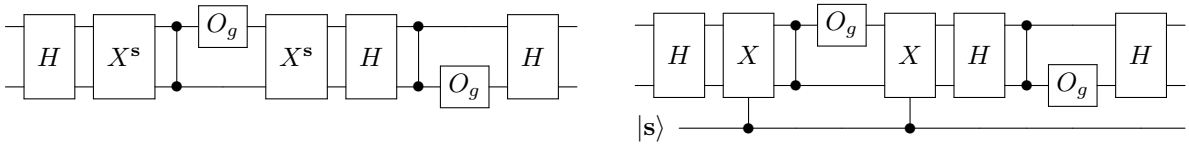
For the addition circuit, we used a standard out-of-place ripple-carry adder which uses $n-1$ ancilla bits to store intermediate carry values and an additional n bit register to store the output, before copying out and uncomputing. The resulting circuit uses $5n-1$ bits of space for an n bit adder, and $4(n-1)$ Toffoli gates, which are then expanded to the Clifford+T gate set. In this case, the size of the bitwise expansion of $\mathbf{x}+\mathbf{y}$ made it difficult to push to implementation sizes (e.g., 32 bits), though smaller sizes such as 16 bits were verifiable within a minute. *Relational* techniques – e.g., representing the outputs of a path integral as "primed" variables along with a set of equations over the input and output variables – may help to push verification of such functions to larger sizes.

The quantum Fourier transform To test our verification method against circuits using higher-order phase gates, we verified an implementation of the quantum Fourier transform. We use a circuit from [KLM07] together with a final qubit permutation correction and verified it against the specification

$$\mathsf{QFT}_n : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{\mathbf{y} \in \mathbb{F}_2^n} e^{2\pi i \frac{[\mathbf{x}\cdot\mathbf{y}]}{2^n}} |\mathbf{y}\rangle.$$

The phase polynomial $[\mathbf{x}\cdot\mathbf{y}]$ was generated in the obvious way – by setting $[\mathbf{x}] = x_1 + 2x_2 + \ldots + 2^{n-1}x_n$ and multiplying the polynomials. In this case our implementation was able to verify implementations up to 31 bits in size, after which integer overflow occurs due to our implementation of dyadic arithmetic. Given that the 31 bit implementation took only 16 seconds to verify, it appears that with better methods for handling dyadic arithmetic much larger sizes of the QFT are likely verifiable.

The quantum hidden shift algorithm To test our framework on more general quantum algorithms, we implemented a version of the quantum hidden shift algorithm [RÏ10] which has been previously used to test quantum simulation algorithms [BG16]. In particular, given oracles $O_{f'} : |\mathbf{x}\rangle \mapsto f(\mathbf{x}+\mathbf{s})|\mathbf{x}\rangle$ and $O_{\tilde{f}} : |\mathbf{x}\rangle \mapsto \tilde{f}(\mathbf{x})|\mathbf{x}\rangle$ for the shifted and dual bent functions $f', \tilde{f} : \mathbb{F}_2^n \to \{-1,+1\}$ respectively, the circuit $H^{\otimes n} O_{\tilde{f}} H^{\otimes n} O_{f'} H^{\otimes n}$ is known [RÏ10] to implement the mapping $|\mathbf{0}\rangle \mapsto |\mathbf{s}\rangle$.



(a) Hidden shift with a fixed shift $\mathbf{s}$.

(b) Hidden shift with a symbolic shift.

Figure 7.2: Circuits for the Quantum Hidden Shift algorithm.

Following [BG16], we generated random instances of Maiorana McFarland bent functions by setting $f'(\mathbf{x}, \mathbf{y}) = f((\mathbf{x}, \mathbf{y}) + \mathbf{s}) = (-1)^{g(\mathbf{x})+\mathbf{xy}}$ with dual $\tilde{f}(\mathbf{x}, \mathbf{y}) = (-1)^{g(\mathbf{y})+\mathbf{xy}}$ for a random $\frac{n}{2}$ bit Boolean function g of degree 3. The circuit for f is generated by, for a given number of alternations A, alternating between selecting 200 random Z and controlled-Z gates, then a random doubly controlled-Z gate, expanded out to Clifford+T. We implemented two versions of the algorithm, one where a concrete shift is given by a randomly generated Boolean vector, and another where the shift is supplied symbolically via a quantum register. In the former case we verify the circuit for a given shift $\mathbf{s}$ against the specification $|\mathbf{0}\rangle \mapsto |\mathbf{s}\rangle$, and in the latter case we verify the specification $|\mathbf{0}\rangle|\mathbf{s}\rangle \mapsto |\mathbf{s}\rangle|\mathbf{s}\rangle$. Figure 7.2 shows both circuits.

Our verification algorithm found a bug in our first implementation, which was a direct implementation of the circuit given in [BG16]. After reimplementing the circuit based on [RÏ0], we were able to verify both versions of the hidden shift algorithm for sizes exceeding those simulated in [BG16] with only a fraction of the time (seconds versus hours [BG16]). Our calculus further finds the correct output $|\mathbf{s}\rangle$ or $|\mathbf{s}\rangle|\mathbf{s}\rangle$ even without providing the specification, effectively simulating the algorithm rather than verifying it. Moreover, our implementation is deterministic compared to theirs which is probabilistic and only samples the output distribution, rather than compute it outright. It is interesting to note that their algorithm also uses a similar technique of effectively evaluating the circuit's phase polynomial – however, by including the T gate phases directly in the polynomial and solving *around* them, rather than pushing them into state preparations, we save a massive amount of time for this algorithm. An interesting question for future research is to determine whether there are quantum algorithms which can be simulated more efficiently by their methods.

7.5 Related work

Equivalence checking Most of the previous work on *functional* verification – as opposed to *property checking* – of quantum circuits has been from the perspective of strict equivalence checking. Such works typically consider a circuit or program to be the specification of an algorithm, and check that another circuit or program has the same semantics.

Some of the earliest work in this vein used SAT solvers to verify reversible circuits of up to a hundred qubits and thousands of gates [WGMD09]. While this work is only directly applicable to reversible circuits, Yamashita and Markov [YM10] later combined it with a method of identifying classical transformations in quantum circuits to first rewrite quantum circuits as reversible ones. Their method was able to verify circuits with up to 128 qubits *but only after local simplifications*, which due to the structure of their test circuits effectively reduces the verification to at most a few gates. However, their work effectively use classical verification techniques on classical circuits to achieve good results, and hence are only applicable for verifying classical oracles before being expanded down to a quantum circuit.

Diagrammatic calculi More recent work has used diagrammatic calculi such as the ZX-calculus [CD11] to check equivalence of quantum circuits by reducing diagrams to the identity. With the help of semi-automated proof assistants such as Quantomatic [KZ15] and Globular [BKV18], correctness proofs of the Steane code and a particular colour code have been developed [DL13, GD17]. However, due to the nature of diagrammatic reasoning, automating this process has proved challenging [BGK+16], and moreover such proofs frequently get "stuck" in local minima, where the diagram has to be expanded before it can be further contracted. In contrast, our method has a very natural notion of the complexity of an operator – the number of path variables – and all circuits we have examined admit proofs which strictly reduce this parameter.

Formal proof Moving away from strict equivalence checking, recent work [RPZ17] has formally proven the correctness of quantum circuits with respect to specifications as superoperators using the Coq proof assistant. Their result shows classical proof assistants can reasonably be used to proof correctness of quantum programs, though due to the nature of the linear-algebraic model, their work was limited to small circuits, or circuits with very simple recursive structure. Anecdotally [Pay18], *specifying* the operations in the superoperator model proved to be difficult and error-prone, providing evidence that effective methods of specifying quantum algorithms are needed for practical verification.

Chapter 8

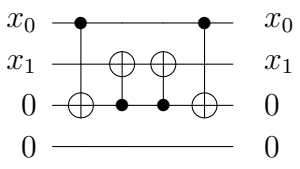
Verified compilation

In this chapter, we shift focus from formally verifying that a single circuit is correct to verifying that *all compiled circuits* are correct. In contrast to the last chapter where the correct *specification* of a circuit's unitary action was one of the main questions, we now take a *program* as the specification of a circuit – a program which itself may have bugs and should be verified to certify the top-to-bottom correctness of a circuit. As per McCarthy and Painter [MP67], this can be reframed as formally verifying that the *compiler*, which takes a specification (e.g., a program) and compiles a circuit, is itself correct; that is, the compiler always produces a circuit which correctly implements the program.

We implement and formally prove the correctness of a reversible circuit compiler called ReVerC in the proof assistant F* [SHK+16]. The compiler itself builds on the Revs language and compiler [PRS15], which compiled a subset of the classical, irreversible F# language to reversible circuits, optimizing for space-efficiency by performing *eager cleanup*. Their method makes use of a dependency graph to determine which bits may be eligible to be cleaned eagerly *without requiring re-computation*, freeing up space without using additional gates. However, like most real-world compilers Revs has bugs, as illustrated by the F# program below to the left, compiled to the incorrect circuit on the right.

```
let f (a : bool array) =
    let b = Array.zeroCreate 2
    b.[0] <- a.[0]
    a.[1] <- a.[1] <> b.[0]
    b
let a = Array 2
let b = f a
a
```

$\longrightarrow$



$$
\begin{aligned}
&\textbf{Var } x, \quad \textbf{Bool } b \in \{0,1\} = \{0,1\}, \quad \textbf{Nat } i,j \in \mathbb{N}, \quad \textbf{Loc } l \in \mathbb{N} \\
&\textbf{Val } v ::= \mathsf{unit} \mid l \mid \mathsf{reg}\ l_1 \ldots l_n \mid \lambda x.t \\
&\textbf{Term } t ::= \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \mid \lambda x.t \mid (t_1\ t_2) \mid t_1; t_2 \mid x \mid t_1 \leftarrow t_2 \mid b \mid t_1 \oplus t_2 \mid t_1 \wedge t_2 \\
&\qquad\qquad \mid \mathsf{clean}\ t \mid \mathsf{assert}\ t \mid \mathsf{reg}\ t_1 \ldots t_n \mid t.[i] \mid t.[i..j] \mid \mathsf{append}\ t_1\, t_2 \mid \mathsf{rotate}\ i\ t
\end{aligned}
$$

Figure 8.1: Syntax of Revs.

The main objective of this line of research was to build a similarly optimizing compiler which is also verified without too much loss in optimization. As formally verified compilers are significantly more constrained in terms of difficulty of implementation, and in the optimizations possible, some loss in optimization is expected (see, e.g., [Ler06] and the unsound optimizations in the GNU C Compiler [GCC16]). Our compiler nonetheless achieves similar bit counts while also being certifiably correct, up to the correctness of the F* proof checker. In addition to formal verification of the compiler, our implementation provides an assertion checker which can be used to formally verify the source program itself, allowing end-to-end verification.

This work appears in [ARS17], and is implemented in ReVerC.

8.1 Languages

In this section we give a formal definition of Revs, as well as the intermediate and target languages of the compiler.

8.1.1 The Source

The abstract syntax of Revs is presented in Figure 8.1. The core of the language is a simple imperative language over Boolean and array (register) types. The language is further extended with ML-style functional features, namely first-class functions and *let* definitions, and a reversible domain-specific construct *clean* which asserts that its argument evaluates to 0 and frees a bit.

```
1  fun a b ->
2    let carry_ex a b c = (a ∧ (b ⊕ c)) ⊕ (b ∧ c)
3    let result = Array.zeroCreate(n)
4    let mutable carry = false

6    result.[0] ← a.[0] ⊕ b.[0]
7    for i in 1 .. n-1 do
8      carry ← carry_ex a.[i-1] b.[i-1] carry
9      result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
10   result
```

Figure 8.2: Implementation of an n-bit adder.

In addition to the basic syntax of Figure 8.1 we add the following derived operations:

$$\neg t \triangleq 1 \oplus t, \qquad t_1 \vee t_2 \triangleq (t_1 \wedge t_2) \oplus (t_1 \oplus t_2),$$
$$\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \triangleq (t_1 \wedge t_2) \oplus (\neg t_1 \wedge t_3),$$
$$\text{for } x \text{ in } i..j \text{ do } t \triangleq t[x \mapsto i]; \cdots ; t[x \mapsto j].$$

Note that Revs has no *dynamic* control – i.e. control flow dependent on run-time values. In particular, every Revs program can be transformed into a straight-line program. While reversible instruction set architectures which allow control exist (e.g., [Vie95]), as our focus is on *quantum* reversible circuits which are purely combinational, we can only compile programs which can be statically unrolled.

The ReVerC compiler uses F# as a meta-language to generate Revs code with particular register sizes and indices, possibly computed by some more complex program. Writing an F# program that generates Revs code is similar in effect to writing in a hardware description language [Cla01]. We use F#'s *quotations* mechanism to achieve this by writing Revs programs in quotations `<@. . . @>`. Note again that unlike QRAM-based languages such as Quipper, our strictly combinational target architecture doesn't allow computations in the meta-language to depend on computations within Revs.

Example 8.1.1. Figure 8.2 gives an example of a carry-ripple adder written in Revs. Naïvely compiling this implementation would result in a new bit being allocated for every carry bit, as the assignment on line 8 is irreversible (note that `carry_ex 1 1 0` = `1` = `carry_ex 1 1 1`, hence the value of `c` can not be uniquely computed given `a`, `b` and the output). ReVerC reduces this space usage by automatically cleaning the old carry bit, allowing it to be reused.

```
let s0 a =
  let a2 = rot 2 a
  let a13 = rot 13 a
  let a22 = rot 22 a
  let t =  Array.zeroCreate 32
  for i in 0 .. 31 do
    t.[i] ← a2.[i] ⊕ a13.[i] ⊕ a22.[i]
  t
let s1 a =
  let a6 = rot 6 a
  let a11 = rot 11 a
  let a25 = rot 25 a
  let mutable t =  Array.zeroCreate 32
  for i in 0 .. 31 do
    t.[i] ← a6.[i] ⊕ a11.[i] ⊕ a25.[i]
  t
let ma a b c =
  let t =  Array.zeroCreate 32
  for i in 0 .. 31 do
    t.[i] ← (a.[i] ∧ (b.[i] ⊕ c.[i]))
            ⊕ (b.[i] ∧ c.[i])
  t
let ch e f g =
  let t =  Array.zeroCreate 32
  for i in 0 .. 31 do
    t.[i] ← e.[i] ∧ f.[i] ∧ g.[i]
  t
fun k w x →
  let hash x =
    let a = x.[0..31], b = x.[32..63]
      c = x.[64..95], d = x.[96..127],
      e = x.[128..159], f = x.[160..191],
      g = x.[192..223], h = x.[224..255]
    (%modAdd 32) (ch e f g) h
    (%modAdd 32) (s0 a) h
    (%modAdd 32) w h
    (%modAdd 32) k h
    (%modAdd 32) h d
    (%modAdd 32) (ma a b c) h
    (%modAdd 32) (s1 e) h
  for i in 0 .. n - 1 do
    hash (rot 32*i x)
  x
```

Figure 8.3: Revs implementation of the SHA-256 algorithm with n rounds, using a meta-function modAdd.

Example 8.1.2. The 256-bit Secure Hash Algorithm 2 (SHA-256) is a cryptographic hash algorithm which performs a series of modular additions and functions on 32-bit chunks of a 256-bit hash value, before rotating the register and repeating for a number of rounds. In the Revs implementation, shown in Figure 8.3, this is achieved by a function `hash` which takes a length 256 register, then makes calls to modular adders with different 32-bit slices. At the end of the round, the entire register is rotated 32 bits with the `rotate` command.

Semantics We designed the semantics of Revs with two goals in mind:

1. keep the semantics as close to the original implementation as possible, and
2. simplify the task of formal verification.

The result is a somewhat non-standard semantics that is nonetheless intuitive for the programmer. Moreover, the particular semantics naturally enforces a style of programming that results in efficient circuits and allows common design patterns to be optimized.

The big-step semantics of Revs is presented in Figure 8.4 as a relation $\Rightarrow \subseteq \text{Config} \times \text{Config}$ on configuration-pairs – pairs of terms and Boolean-valued stores. A key feature of the semantics is that Boolean, or bit values, are always allocated on the store. Specifically, Boolean constants and expressions are modelled by allocating a new location on the store to hold its value – as a result all Boolean values, including constants, are mutable.

The allocation of Boolean values on the store serves two main purposes: to give the programmer fine-grain control over how many bits are allocated, and to provide a simple and efficient model of *registers* – i.e. arrays of bits. Specifically, registers are modelled as static length lists of bits. This allows the programmer to perform array-like operations such as bit modifications ($t_1.[i] \leftarrow t_2$) as well as list-like operations such as slicing ($t.[i..j]$) and concatenation ($\mathsf{append}\ t_1\, t_2$) without copying out entire registers. We found that these were the most common access patterns for arrays of bits in low-level bitwise code (e.g. arithmetic and cryptographic implementations).

The semantics of $\oplus$ (Boolean XOR) and $\wedge$ (Boolean AND) are also notable in that they first reduce both arguments to locations, *then* retrieve their value. This results in statements whose value may not be immediately apparent – e.g., $x \oplus (x \leftarrow y; y)$, which under these semantics will always evaluate to 0. The benefit of this definition is that it allows the compiler to perform important optimizations without a significant burden on the programmer.

Store $\sigma : \mathbb{N} \rightharpoonup \{0, 1\}$

Config $c ::= \langle t, \sigma \rangle$

$$[\textsc{let}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle v_1, \sigma' \rangle \qquad \langle t_2[x \mapsto v_1], \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle}{\langle \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2, \sigma \rangle \Rightarrow \langle v_2, \sigma'' \rangle}$$

$$[\textsc{refl}]\ \frac{}{\langle v, \sigma \rangle \Rightarrow \langle v, \sigma \rangle} \qquad [\textsc{bexp}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle \quad l_3 \notin \mathsf{dom}(\sigma'')}{\langle t_1 \star t_2, \sigma \rangle \Rightarrow \langle l_3, \sigma''[l_3 \mapsto \sigma''(l_1) \star \sigma''(l_2)] \rangle}$$

$$[\textsc{bool}]\ \frac{b \in \{0, 1\} \quad l \notin \mathsf{dom}(\sigma)}{\langle b, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto b] \rangle} \qquad [\textsc{app}]\ \frac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle \lambda x.t_1', \sigma' \rangle \langle t_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \\ \langle t_1'[x \mapsto v_2], \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle\end{array}}{\langle (t_1\ t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle}$$

$$[\textsc{seq}]\ \frac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle \\ \langle t_2, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle\end{array}}{\langle t_1; t_2, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle} \qquad [\textsc{assn}]\ \frac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \\ \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle\end{array}}{\langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma''[l_1 \mapsto \sigma''(l_2)] \rangle}$$

$$[\textsc{append}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_m, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle \mathsf{reg}\ l_{m+1} \ldots l_n, \sigma'' \rangle}{\langle \mathsf{append}\ t_1\ t_2, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_n, \sigma'' \rangle}$$

$$[\textsc{index}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_n, \sigma' \rangle \qquad 1 \leq i \leq n}{\langle t.[i], \sigma \rangle \Rightarrow \langle l_i, \sigma' \rangle} \qquad \begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma_1 \rangle \\ \langle t_2, \sigma \rangle \Rightarrow \langle l_2, \sigma_2 \rangle \\ \vdots \\ \langle t_n, \sigma \rangle \Rightarrow \langle l_n, \sigma_n \rangle\end{array}$$

$$[\textsc{slice}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_n, \sigma' \rangle \quad 1 \leq i \leq j \leq n}{\langle t.[i..j], \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_i \ldots l_j, \sigma' \rangle} \qquad [\textsc{reg}]\ \frac{}{\langle \mathsf{reg}\ t_1 \ldots t_n, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_n, \sigma_n \rangle}$$

$$[\textsc{rotate}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \ldots l_n, \sigma' \rangle \quad 1 < i < n}{\langle \mathsf{rotate}\ t\ i, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_i \ldots l_{i-1}, \sigma' \rangle}$$

$$[\textsc{clean}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = 0}{\langle \mathsf{clean}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \big|_{\mathsf{dom}(\sigma') \setminus \{l\}} \rangle} \qquad [\textsc{assert}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = 1}{\langle \mathsf{assert}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle}$$

Figure 8.4: Operational semantics of Revs.

8.1.2 Boolean expressions

Our compiler uses XOR-AND Boolean expressions – single output classical circuits over XOR and AND gates – as an intermediate language. Compilation from Boolean expressions into reversible circuits forms the main "code generation" step of our compiler.

A Boolean expression is defined as an expression over Boolean constants, variable indices, and logical $\oplus$ and $\wedge$ operators. Explicitly, we define

$$\textbf{BExp}\ \ B ::= 0 \mid 1 \mid i \in \mathbb{N} \mid B_1 \oplus B_2 \mid B_1 \wedge B_2.$$

Note that we use the symbols $0, 1, \oplus$ and $\wedge$ interchangeably with their interpretation in $\{0, 1\}$. We use $\mathsf{vars}(B)$ to refer to the set of free variables in B.

We interpret a Boolean expression as a function from (total) Boolean-valued states to Booleans. In particular, we define $\textbf{State} = \mathbb{N} \to \{0, 1\}$ and denote the semantics of a Boolean expression by $[\![B]\!] : \textbf{State} \to \{0, 1\}$. The formal definition of $[\![B]\!]$ is given by the obvious homomorphism into $\mathbb{F}_2$.

8.1.3 Target architecture

ReVerC compiles to quantum circuits over $\{X, \text{CNOT}, \text{TOF}\}$, which were earlier noted as being reversible computing and hence are suitable for implementing classical functions and oracles within quantum computations.

As we are working with strictly reversible circuits in this chapter, we use a slightly different semantic model. To differentiate the *reversible circuit language* we target here from the (unitary) circuit model used previous, we define

$$\textbf{Circ}\ C ::= - \mid \text{NOT}\ i \mid \text{CNOT}\ i\ j \mid \text{Toffoli}\ i\ j\ k \mid C_1 \fatsemi C_2.$$

Recall that all but the last bit in each gate is a *control*, and the final bit is denoted as the *target*. In analogy to more classical programming concepts, the target of a gate is the only bit *modified* or mutated by the gate. We use $\mathsf{use}(C)$, $\mathsf{mod}(C)$ and $\mathsf{control}(C)$ to denote the set of bit indices that are used in, modified by, or used as a control in the circuit C, respectively. A circuit is *well-formed* if no gate contains more than one reference to a bit – i.e., the bits used in each controlled-NOT or Toffoli gate are distinct.

Similar to Boolean expressions, a circuit is interpreted as a function from states (maps from indices to Boolean values) to states, given by applying each gate which updates the

previous state in order. The formal definition of the semantics of a reversible circuit C, given by $[\![C]\!] : \mathbf{State} \to \mathbf{State}$, is straightforward:

$$\begin{aligned}
[\![-]\!]s &= s \\
[\![\text{NOT } i]\!]s &= s[i \mapsto \neg s(i)] \\
[\![\text{CNOT } i\ j]\!]s &= s[j \mapsto s(i) \oplus s(j)] \\
[\![\text{Toffoli } i\ j\ k]\!]s &= s[k \mapsto (s(i) \wedge s(j)) \oplus s(k)] \\
[\![C_1 \fatsemi C_2]\!]s &= ([\![C_2]\!] \circ [\![C_1]\!])s
\end{aligned}$$

We use $s[x \mapsto y]$ to denote the function that maps x to y, and all other inputs z to $s(z)$; by an abuse of notation we use $[x \mapsto y]$ to denote other substitutions as well.

8.2 Compilation

In this section we discuss the implementation of ReVerC. The compiler consists of around 4000 lines of code in a common subset of F* and F#, with a front-end to evaluate and translate F# quotations into Revs expressions.

8.2.1 Boolean expression compilation

The core of ReVerC's code generation is a compiler from Boolean expressions into reversible circuits. We use a modification of the method employed in Revs.

As a Boolean expression is already in the form of an irreversible classical circuit, the main job of the compiler is to allocate ancillas to store sub-expressions whenever necessary. ReVerC does this by maintaining a (mutable) heap of ancillas $\xi \in \mathbf{AncHeap}$ called an *ancilla heap*, which keeps track of the currently available (zero-valued) ancillary bits. Cleaned ancillas (ancillas returned to the zero state) may be pushed back onto the heap, and allocations return previously used ancillas if any are available, hence not using any extra space.

The function compile-BExp, shown in pseudo-code below, takes a Boolean expression B and a target bit i and then generates a reversible circuit computing $i \oplus B$. Note that ancillas are only allocated to store sub-expressions of $\wedge$ expressions, since $i \oplus (B_1 \oplus B_2) = (i \oplus B_1) \oplus B_2$ and so we compile $i \oplus (B_1 \oplus B_2)$ by first computing $i' = i \oplus B_1$, followed by $i' \oplus B_2$.

function COMPILE-BEXP(B, i, ξ)
  if $B = 0$ **then** $-$
  else if $B = 1$ **then** NOT i
  else if $B = j$ **then** CNOT j i
  else if $B = B_1 \oplus B_2$ **then** COMPILE-BEXP(B_1, i, ξ) ⨟ COMPILE-BEXP(B_2, i, ξ)
  else // $B = B_1 \wedge B_2$
    $a_1 \leftarrow$ pop-min(ξ); $C \leftarrow$ COMPILE-BEXP(B_1, a_1, ξ);
    $a_2 \leftarrow$ pop-min(ξ); $C' \leftarrow$ COMPILE-BEXP(B_2, a_2, ξ);
    C ⨟ C' ⨟ Toffoli a_1 a_2 i
  end if
end function

Cleanup

The definition of COMPILE-BEXP above leaves many garbage bits that take up space and need to be cleaned before they can be re-used. To reclaim those bits, we clean temporary expressions after every call to COMPILE-BEXP.

To facilitate the cleanup – or *uncomputing* – of a circuit, we define the *restricted inverse* $\mathsf{uncompute}(C, A)$ of C with respect to a set of bits $A \subset \mathbb{N}$ by reversing the gates of C, and removing any gates with a target in A. For instance:

$$\mathsf{uncompute}(\text{CNOT } i\ j, A) = \begin{cases} - & \text{if } j \in A \\ \text{CNOT } i\ j & \text{otherwise} \end{cases}$$

The other cases are defined similarly. Note that since uncompute produces a subsequence of the original circuit C, no ancillary bits are used.

The restricted inverse allows the temporary values of a reversible computation to be uncomputed without affecting any of the target bits. In particular, if $C =$ COMPILE-BEXP(B, i), then the circuit C ⨟ $\mathsf{uncompute}(C, \{i\})$ maps a state s to $s[i \mapsto [\![B]\!]s \oplus s(i)]$, allowing any newly allocated ancillas to be pushed back onto the heap. Intuitively, since no bits contained in the set A are modified, the restricted inverse preserves their values; that the restricted inverse uncomputes the values of the remaining bits is less obvious, but it can be observed that if the computation doesn't *depend* on the value of a bit in A, the computation will be inverted. We formalize and prove this statement in Section 8.4.

8.2.2 Revs compilation

In studying the Revs compiler, we observed that most of what the compiler was doing was evaluating the non-Boolean parts of the program – effectively bookkeeping for registers – only generating circuits for a small kernel of cases. As a result, transformations to different Boolean representations (e.g., circuits, dependence graphs [PRS15]) and the interpreter itself reused significant portions of this bookkeeping code. To make use of this redundancy to simplify both writing and verifying the compiler, we designed ReVerC as a *partial evaluator* parameterized by an abstract machine for evaluating Boolean expressions. As a side effect, we arrive at a unique model for circuit compilation similar to staged computation (see, e.g., [JGS93]).

ReVerC works by evaluating the program with an abstract machine providing mechanisms for initializing and assigning locations on the store to Boolean expressions. We call an instantiation of this abstract machine an *interpretation* $\mathscr{I}$, which consists of a domain D equipped with two operators:

$$\begin{aligned}&\mathsf{assign} : D \times \mathbb{N} \times \mathbf{BExp} \to D\\ &\mathsf{eval} : D \times \mathbb{N} \times \mathbf{State} \rightharpoonup \{0,1\}.\end{aligned}$$

We typically denote an element of an interpretation domain D by σ. A sequence of assignments in an interpretation builds a Boolean computation or circuit within a specific model (i.e., classical, reversible, different gate sets) which may be simulated on an initial state with the eval function – effectively an operational semantics of the model. Practically speaking, an element of D abstracts the store in Figure 8.4 and allows delayed computation or additional processing of the Boolean expression stored in a cell, which may be mapped into reversible circuits immediately or after the entire program has been evaluated. We give some examples of interpretations below.

Example 8.2.1. The standard interpretation $\mathscr{I}_{standard}$ has domain $\mathbf{Store} = \mathbb{N} \rightharpoonup \{0,1\}$, together with the operations

$$\begin{aligned}&\mathsf{assign}_{standard}(\sigma, l, B) = \sigma[l \mapsto [\![B]\!]\sigma]\\ &\mathsf{eval}_{standard}(\sigma, l, s) = \sigma(l).\end{aligned}$$

Partial evaluation over the standard interpretation coincides exactly with the operational semantics of Revs.

Example 8.2.2. The *reversible circuit* interpretation $\mathscr{I}_{circuit}$ has domain $D_{circuit} = (\mathbb{N} \rightharpoonup \mathbb{N}) \times \mathbf{Circ} \times \mathbf{AncHeap}$. In particular, given $(\rho, C, \xi) \in D_{circuit}$, ρ maps heap locations

to bits in C, and ξ is an ancilla heap. Assignment and evaluation are further defined as follows:

$$\begin{aligned}
&\mathsf{assign}_{circuit}((\rho, C, \xi), l, B) = (\rho[l \mapsto i], C \fatsemi C', \xi) \\
&\qquad \text{where } i = \ \text{pop-min}(\xi), \\
&\qquad (C', \xi') = \ \textsc{compile-BExp}\,(B[l' \in \mathsf{vars}(B) \mapsto \rho(l')], i, \xi) \\
&\mathsf{eval}_{circuit}((\rho, C, \xi), l, s)) = (\llbracket C \rrbracket s)\,(\rho(l))
\end{aligned}$$

Interpreting a program with $\mathscr{I}_{circuit}$ builds a reversible circuit executing the program, together with a mapping from heap locations to bits. Since the circuit is required to be reversible, when a location is overwritten, a new ancilla i is allocated and the expression $B \oplus i$ is compiled into a circuit. Evaluation amounts to running the circuit on an initial state, then retrieving the value at the bit associated with a heap location.

Given an interpretation $\mathscr{I}$ with domain D, we define the set of $\mathscr{I}$-configurations as $\mathbf{Config}_{\mathscr{I}} = \mathbf{Term} \times D$ – that is, $\mathscr{I}$-configurations are pairs of programs and elements of D which function as an abstraction of the heap. The relation

$$\Rightarrow_{\mathscr{I}} \subseteq \mathbf{Config}_{\mathscr{I}} \times \mathbf{Config}_{\mathscr{I}}$$

gives the operational semantics of Revs over the interpretation $\mathscr{I}$, and corresponds to the definition of $\Rightarrow$ (Figure 8.4) with all heap updates replaced with assign. Note that $\mathsf{dom}(\sigma)$ refers to the set of locations on which eval is defined. To compile a program term t, ReVerC evaluates t over a particular interpretation $\mathscr{I}$ (for instance, the reversible circuit interpretation) and an initial heap $\sigma \in D$ according to the semantic relation $\Rightarrow_{\mathscr{I}}$. In this way, evaluating a program and compiling a program to a circuit look almost identical. This greatly simplifies the problem of verification, as we will see in the next section.

ReVerC supports three modes of compilation, defined by giving interpretations: a default mode, an eagerly cleaned mode, and a "crush" mode. The default mode evaluates the program using the circuit interpretation, and simply returns the circuit and output bit(s), while the eager cleanup mode operates analogously, using instead the garbage-collected interpretation defined below in Section 8.2.3. The crush mode interprets a program as a list of Boolean expressions over free variables, which while unscalable allows highly optimized versions of small circuits to be compiled, a common practice in circuit synthesis. We omit the details of the Boolean expression interpretation.

8.2.3 Eager cleanup

It was previously noted that the circuit interpretation allocates a new ancilla on every assignment to a location, due to the requirement of reversibility. Apart from ReVerC's additional optimization passes, this is effectively the Bennett method, and hence uses a large amount of extra space. One way to keep the space usage from continually expanding as assignments are made is to clean the old bit as soon as possible and then reuse it, rather than wait until the end of the computation. Here we develop an interpretation that performs this automatic, eager cleanup by augmenting the circuit interpretation with a *cleanup expression* for each bit. Our method is based on the eager cleanup of [PRS15], and was intended as a more easily verifiable alternative to mutable dependency diagrams.

The *eager cleanup* interpretation $\mathscr{I}_{GC}$ has domain

$$D = (\mathbb{N} \rightharpoonup \mathbb{N}) \times \mathbf{Circ} \times \mathbf{AncHeap} \times (\mathbb{N} \rightharpoonup \mathbf{BExp}),$$

where given $(\rho, C, \xi, \kappa) \in D$, ρ, C and ξ are as in the circuit interpretation. The partial function κ maps individual bits to a Boolean expression over the bits of C which can be used to return the bit to its initial state, called the cleanup expression. Specifically, we have the following property:

$$\forall i \in \mathsf{cod}(\rho), s'(i) \oplus [\![\kappa(i)]\!]s' = s(i) \qquad \text{where } s' = [\![C]\!]s.$$

Intuitively, any bit i can then be cleaned by simply computing $i \mapsto i \oplus \kappa(i)$, which in turn can be done by calling compile-BExp($\kappa(i)$, i).

Two problems remain, however. In general it may be the case that a bit *can not* be cleaned without affecting the value of other bits, as it might result in a loss of information – in the context of cleanup expressions, this occurs exactly when a bit's cleanup expression contains an irreducible self-reference. In particular, if $i \in \mathsf{vars}(B)$, then compile-BExp(B, i) does not compile a circuit computing $i \oplus B$ and hence won't clean the target bit correctly. In the case when a garbage bit contains a self-reference in its cleanup expression that can not be eliminated by Boolean simplification, ReVerC simply ignores the bit and performs a final round of cleanup at the end.

The second problem arises when a bit's cleanup expression references another bit that has itself since been cleaned or otherwise modified. In this case, the modification of the latter bit has invalidated the correctness property for the former bit. To ensure that the above relation always holds, whenever a bit is modified – corresponding to an XOR of the bit, i, with a Boolean expression B – all instances of bit i in every cleanup expression is replaced with $i \oplus B$. Specifically we observe that, if $s'(i) = s(i) \oplus [\![B]\!]s$, then

$$s'(i) \oplus [\![B]\!]s = s(i) \oplus [\![B]\!]s \oplus [\![B]\!]s = s(i).$$

The function CLEAN, defined below, performs the cleanup of a bit i if possible, and validates all cleanup expressions in a given element of D:

---

function CLEAN((ρ, C, ξ, κ), i)
  if $i \in \mathsf{vars}(\kappa(i))$ **then return** (ρ, C, ξ, κ)
  else
    $C' \leftarrow$ COMPILE-BEXP$(\kappa(i), i, \xi)$
    if i is an ancilla **then** insert(i, ξ)
    $\kappa' \leftarrow \kappa[i' \in \mathsf{dom}(\kappa) \mapsto \kappa(i')[i \mapsto i \oplus \kappa(i)]]$
    return $(\rho, C \fatsemi C', \xi, \kappa')$
  end if
end function

Assignment and evaluation are defined in the eager cleanup interpretation as follows. Both are effectively the same as in the circuit interpretation, except the assignment operator calls CLEAN on the previous bit mapped to l.

$$\begin{aligned}
&\mathsf{assign}_{GC}((\rho, C, \xi, \kappa), l, B) = \textsc{clean}((\rho[l \mapsto i], C \fatsemi C', \xi, \kappa[i \mapsto B']), i) \\
&\quad \text{where } i = \text{pop-min}(\xi), \\
&\qquad B' = B[l' \in \mathsf{vars}(B) \mapsto \rho(l')] \\
&\qquad C' = \textsc{compile-BExp}(B', i, \xi) \\
&\mathsf{eval}_{GC}((\rho, C, \xi, \kappa), l, s)) = (\llbracket C \rrbracket s)\,(\rho(l))
\end{aligned}$$

The eager cleanup interpretation coincides with a reversible analogue of *garbage collection* for a very specific case when the number of references to a heap location (or in our case, a bit) is trivially zero.

Other optimizations During the course of compilation it is frequently the case that more ancillas are allocated than are actually needed, due to the program structure. For instance, when compiling the expression $i \leftarrow B$, if B can be factored as $i \oplus B'$ the assignment may be performed reversibly rather than allocating a new bit to store the value of B. Likewise if i is provably in the 0 or 1 state, the assignment may be performed reversibly without allocating a new bit. Our implementation identifies some of these common patterns, as well as general Boolean expression simplifications, to further minimize the space usage of compile circuits. All such optimizations in ReVerC have been formally verified.

8.3 Parameter inference

While the definition of ReVerC as a partial evaluator streamlines both development and verification, there is an inherent disconnect between the treatment of a (top-level) function expression by the interpreter and by the compiler, in that we want the compiler to evaluate the function body. Instead of formally defining a two-stage semantics for Revs (e.g., [JGS93]) we took the approach of applying a program transformation, whereby the function being compiled is evaluated on special heap locations representing the parameters. This creates a further problem in that the compiler needs to first determine the size of each parameter; to solve this problem, ReVerC performs a static analysis which we call *parameter interference*, as we frame it as a dependent-type inference.

We note that many other solutions are possible: the original Revs compiler for instance had programmers allocate inputs manually rather than write functions. This led to unnatural-looking programs and semantics which were hard to reason about formally. On the other hand, the problem could be avoided by delaying bit allocation until after the circuit is compiled. We opted for a type inference approach as it simplified verification.

Type system

The type system for parameter inference includes three base types – the unit type, Booleans, and fixed-length registers – as well as the function type. As expected, indexing registers beyond their length causes a type error. To regain some polymorphism and flexibility the type system allows (structural) subtyping, in particular so that registers with greater size may be used anywhere a register with lesser size is required. Type inference in such systems is generally considered a difficult problem – see, for e.g., [PZ04] which proves **NP**-completeness of the problem for a similar type system involving record concatenation. While many attempts have been made [Mit91, EST95, TS96, Pot96, OSW99], no such algorithms are currently in common use. Given the simplicity of our type system (in particular, the lack of non-subtype polymorphism), we have found a relatively simple inference algorithm based on constraint generation and *bound maps* is effective in practice.

Figure 8.5 summarizes algorithmic rules of our type system, which specify a set of constraints that any valid typing most satisfy. Constraints are given over a language of type and integer expressions. Type expressions include the basic types of ReVerC, as well variables representing types and registers over integer expressions. Integer expressions are linear expressions over integer-valued variables and constants. Equality and order relations are defined on type expressions as expected, while the integer expression ordering

corresponds to the *reverse* order on integers ($\geq$) – this definition is used so that the maximal solution to an integer constraint gives the smallest register size. Constraints may be equalities or order relations between expressions of the same kind.

A *type substitution* Θ is a mapping from type variables to closed type expressions and integer variables to integers. By an abuse of notation we denote by $\Theta(\mathcal{T})$ the type $\mathcal{T}$ with all variables replaced by their values in Θ. Additionally, we say that Θ satisfies the set of constraints $\mathcal{C}$ if every constraint is true after applying the substitutions. The relation $\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}$ means t can be given type $\Theta(\mathcal{T})$ for any type substitution satisfying $\mathcal{C}$.

Constraint solving

Algorithm 8.1 Computation of bound sets.

$\mathcal{C}$ is a set of constraints
θ a set of (possible open at one end) ranges for each variable
while $c \in \mathcal{C}$ **do**
  if c is $\mathcal{T}_1 = \mathcal{T}_2$ or $\mathcal{I}_1 = \mathcal{I}_2$ **then** use unification
  else if c is $\mathcal{T} \sqsubseteq \mathcal{T}$ or $\mathcal{I} \sqsubseteq \mathcal{I}$ **then** computeBounds($S \setminus \{c\}$, θ)
  else if c is $X \sqsubseteq$ Unit or Unit $\sqsubseteq X$ **then**
    computeBounds($S \setminus \{c\}$, $\theta \cup \{X \mapsto [\text{Unit}, \text{Unit}]\}$)
  else if c is $X \sqsubseteq$ Bool or Bool $\sqsubseteq X$ **then**
    computeBounds($S \setminus \{c\}$, $\theta \cup \{X \mapsto [\text{Bool}, \text{Bool}]\}$)
  else if c is $X \sqsubseteq$ Register $\mathcal{I}$ **then** computeBounds($S \setminus \{c\} \cup \{\mathcal{I} \sqsubseteq x\}$, $\theta \cup \theta'$)
    where $\theta' = \{X \mapsto [\text{Register } x, \text{Register } x], x \mapsto [\mathcal{I}, \infty]\}$
  else if c is Register $\mathcal{I} \sqsubseteq X$ **then** computeBounds($S \setminus \{c\} \cup \{x \sqsubseteq \mathcal{I}\}$, $\theta \cup \theta'$)
    where $\theta' = \{X \mapsto [\text{Register } x, \text{Register } x], x \mapsto [0, \mathcal{I}]\}$
  else if c is $X \sqsubseteq \mathcal{T}_1 \to \mathcal{T}_2$ **then** computeBounds($S \setminus \{c\} \cup \{\mathcal{T}_1 \sqsubseteq \mathcal{T}_1', \mathcal{T}_2' \sqsubseteq \mathcal{T}_2$, $\theta \cup \theta'$)
    where $\theta' = \{X \mapsto [\mathcal{T}_1' \to \mathcal{T}_2', \mathcal{T}_1' \to \mathcal{T}_2']\}$
  else if c is $\mathcal{T}_1 \to \mathcal{T}_2 \sqsubseteq X$ **then** computeBounds($S \setminus \{c\} \cup \{\mathcal{T}_1' \sqsubseteq \mathcal{T}_1, \mathcal{T}_2 \sqsubseteq \mathcal{T}_2'$, $\theta \cup \theta'$)
    where $\theta' = \{X \mapsto [\mathcal{T}_1' \to \mathcal{T}_2', \mathcal{T}_1' \to \mathcal{T}_2']\}$
  else if c is $X_1 \sqsubseteq X_2$ **then** computeBounds($S \setminus \{c\}$, $\theta \cup \{X_1 \mapsto [-, X_2]\}$)
  else if c is $x \sqsubseteq \mathcal{I}$ **then** computeBounds($S \setminus \{c\}$, $\theta \cup \{x \mapsto [-, \mathcal{I}]\}$
  else if c is $\mathcal{I} \sqsubseteq x$ **then** computeBounds($S \setminus \{c\}$, $\theta \cup \{x \mapsto [\mathcal{I}, -]\}$
  else Fail
  end if
end while

The constraint solving algorithm finds a type substitution satisfying a set of constraints

IExp $\mathcal{I} ::= i \in \mathbb{N} \mid x \mid \mathcal{I}_1 \pm \mathcal{I}_2$

TExp $\mathcal{T} ::= X \mid \text{Unit} \mid \text{Bool} \mid \text{Register } \mathcal{I} \mid T_1 \to T_2$

Const $c ::= \mathcal{I}_1 = \mathcal{I}_2 \mid \mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \mid \mathcal{T}_1 = \mathcal{T}_2 \mid \mathcal{T}_1 \sqsubseteq \mathcal{T}_2$

$$[\text{C-LET}]\frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma, x : \mathcal{T}_1 \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{C-VAR}]\frac{(x : \mathcal{T}) \in \Gamma}{\Gamma \vdash x : \mathcal{T} \downarrow \emptyset} \qquad [\text{C-LAMBDA}]\frac{\Gamma, x : X \vdash t : \mathcal{T} \downarrow \mathcal{C} \qquad X \text{ fresh}}{\Gamma \vdash \lambda x.t : X \to T \downarrow \mathcal{C}}$$

$$[\text{C-APP}]\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2 \\ \mathcal{C}_3 = \{\mathcal{T}_1 = X_1 \to X_2, X_2 \sqsubseteq \mathcal{T}_2\} \qquad X_1, X_2 \text{ fresh}\end{array}}{\Gamma \vdash (t_1\ t_2) : X_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3} \qquad [\text{C-SEQ}]\frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1; t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Unit}\}}$$

$$[\text{C-BEXP}]\frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1 \star t_2 : \text{Bool} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Bool}, \mathcal{T}_2 = \text{Bool}\}} \qquad [\text{C-TRUE}]\frac{}{\Gamma \vdash 1 : \text{Bool} \downarrow \emptyset}$$

$$[\text{C-FALSE}]\frac{}{\Gamma \vdash 0 : \text{Bool} \downarrow \emptyset}$$

$$[\text{C-ASSN}]\frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1 \leftarrow t_2 : \text{Unit} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Bool}, \mathcal{T}_2 = \text{Bool}\}}$$

$$[\text{C-APPEND}]\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \qquad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2 \\ \mathcal{C}_3 = \{\mathcal{T}_1 = \text{Register } x_1, \mathcal{T}_2 = \text{Register } x_2, x_3 = x_1 + x_2\} \quad x_1, x_2, x_3 \text{ fresh}\end{array}}{\Gamma \vdash \mathsf{append}\ t_1\ t_2 : \text{Register } x_3 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3\}}$$

$$[\text{C-INDEX}]\frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \qquad \mathcal{C}_2 = \{\mathcal{T} = \text{Register } x, i \sqsubseteq x\} \qquad x \text{ fresh}}{\Gamma \vdash t.[i] : \text{Bool} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{C-SLICE}]\frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \qquad \mathcal{C}_2\{\mathcal{T} = \text{Register } x, i \sqsubseteq j, j \sqsubseteq x\} \qquad x \text{ fresh}}{\Gamma \vdash t.[i..j] : \text{Register } j - i + 1 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{C-REG}]\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \\ \vdots \\ \Gamma \vdash t_n : \mathcal{T}_n \downarrow \mathcal{C}_n \\ \mathcal{C}_{n+1} = \{\mathcal{T}_i = \text{Bool} \mid \forall 1 \le i \le n\}\end{array}}{\Gamma \vdash \mathsf{reg}\ t_1 \ldots t_n : \text{Register } n \downarrow \bigcup_{i=1}^{n+1} \mathcal{C}_i}$$

$$[\text{C-ROTATE}]\frac{\begin{array}{c}\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \\ \mathcal{C}_2\{\mathcal{T} = \text{Register } x, i \sqsubseteq x\} \qquad x \text{ fresh}\end{array}}{\Gamma \vdash \mathsf{rotate}\ t\ i : \text{Register } x \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \qquad [\text{C-CLEAN}]\frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}}{\Gamma \vdash \mathsf{clean}\ t : \text{Unit} \downarrow \mathcal{C} \cup \{\mathcal{T} = \text{Bool}\}}$$

$$[\text{C-ASSERT}]\frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}}{\Gamma \vdash \mathsf{assert}\ t : \text{Unit} \downarrow \mathcal{C} \cup \{\mathcal{T} = \text{Bool}\}}$$

Figure 8.5: Constraint typing rules

$\mathcal{C}$. We use a combination of unification (see, for e.g., [DM82]) and sets of upper and lower bounds for variables, similar to the method used in [TS96]. The function computeBounds, shown in pseudo-code in Algorithm 8.1, iterates through a set of constraints performing unification and computing the closure of the constraints wherever possible. Bounds – both subtype and equality – on variables are translated to a range of type or integer expressions, possibly open at one end, and the algorithm maintains a set of such bounds for each variable. To reduce complex linear arithmetic constraints to a variable bound we use a normalization procedure to write the constraint with a single positive polarity variable on one side.

After all constraints have translated to bounds we iterate through the variables, simplifying and checking the set of upper and lower bounds. Any variable with no unassigned variable in its upper and lower bound sets is assigned the maximum value in the intersection of all its bounds; this process repeats until no variables are left unassigned. It can be observed that the algorithm terminates, as the syntax of Revs does not allow circular systems of constraints to be generated.

8.4 Verification

In this section we describe the formal verification of ReVerC and give the major theorems proven. All theorems given in this section have been formally specified and proven using the F* compiler [SHK+16]. We first give theorems about our Boolean expression compiler, then use these to prove properties about whole program compilation. The total verification of the ReVerC core's approximately 2000 lines of code comprises around 2200 lines of F* code, and took just over 1 person-month. This relatively low-cost verification is a testament to the increasing ease with which formal verification can be carried out using modern proof assistants. Additionally, the verification relies on only 11 unproven axioms regarding properties of lookup tables and sets, such as the fact that a successful lookup is in the codomain of a lookup table.

We give the mathematical intuition behind the formal F* proofs, as it is much more enlightening than the primarily engineering-driven proof code.

8.4.1 Boolean expression compilation

Correctness Below is the main theorem establishing the correctness of the function compile-BExp with respect to the semantics of reversible circuits and Boolean expressions.

It states that if the variables of B, the bits on the ancilla heap and the target are non-overlapping, and if the ancilla bits are 0-valued, then the circuit computes the expression $i \oplus B$.

Theorem 8.4.1. *Let B be a Boolean expression, ξ be an ancilla heap, $i \in \mathbb{N}$, $C \in$ **Circ** and s be a map from bits to Boolean values. Suppose $\mathsf{vars}(B)$, ξ and $\{i\}$ are all disjoint and $s(j) = 0$ for all $j \in \xi$. Then*

$$(\llbracket \textsc{compile-BExp}(B, i, \xi) \rrbracket s)\,(i) = s(i) \oplus \llbracket B \rrbracket s.$$

Cleanup As remarked earlier, a crucial part of reversible computing is cleaning ancillas both to reduce space usage, and in quantum computing to prevent entangled qubits from influencing the computation. Moreover, the correctness of our cleanup is actually necessary to prove correctness of the compiler, as the compiler re-uses cleaned ancillas on the heap, potentially interfering with the precondition of Theorem 8.4.1. We use the following lemma to establish the correctness of our cleanup method, stating that the uncompute transformation reverses all changes on bits not in the target set under the condition that no bits in the target set are used as controls.

Lemma 8.4.2. *Let C be a well-formed reversible circuit and $A \subset \mathbb{N}$ be some set of bits. If $A \cap \mathsf{control}(C) = \emptyset$ then for all states $s, s' = \llbracket C \fatsemi \mathsf{uncompute}(C, A) \rrbracket s$ and any $i \notin A$,*

$$s(i) = s'(i)$$

Lemma 8.4.2 largely relies on the following important lemma stating in effect that the action of a circuit is determined by the values of the bits used as controls:

Lemma 8.4.3. *Let $A \subset \mathbb{N}$ and s, s' be states such that for all $i \in A$, $s(i) = s'(i)$. If C is a reversible circuit where $\mathsf{control}(C) \subseteq A$, then*

$$(\llbracket C \rrbracket s)(i) = (\llbracket C \rrbracket s')(i)$$

for all $i \in A$.

Lemma 8.4.2, together with the fact that COMPILE-BEXP produces a well-formed circuit under disjointness constraints, gives us our cleanup theorem below that Boolean expression compilation with cleanup correctly reverses the changes to every bit except the target.

Theorem 8.4.4. *Let B be a Boolean expression, ξ be an ancilla heap and $i \in \mathbb{N}$ such that $\mathsf{vars}(B)$, ξ and $\{i\}$ are all disjoint. Suppose $\textsc{compile-BExp}(B, i, \xi) = C$. Then for all $j \neq i$ and states s we have*

$$(\llbracket C \fatsemi \mathsf{uncompute}(C, \{i\}) \rrbracket s)\,(j) = s(j).$$

8.4.2 REVS compilation

It was noted in Section 8.2 that the design of REVERC as a partial evaluator simplifies proving correctness. We expand on that point now, and in particular show that if a relation between elements of two interpretations is preserved by assignment, then the evaluator also preserves the relation. We state this formally in the theorem below.

Theorem 8.4.5. *Let $\mathscr{I}_1, \mathscr{I}_2$ be interpretations and suppose whenever $(\sigma_1, \sigma_2) \in R$ for some relation $R \subseteq \mathscr{I}_1 \times \mathscr{I}_2$,*

$$(\mathsf{assign}_1(\sigma_1, l, B), \mathsf{assign}_2(\sigma_2, l, B)) \in R$$

for any l, B. Then for any term t, if $\langle t, \sigma_1 \rangle \Rightarrow_{\mathscr{I}_1} \langle v_1, \sigma_1' \rangle$ and $\langle t, \sigma_2 \rangle \Rightarrow_{\mathscr{I}_2} \langle v_2, \sigma_2' \rangle$, then $v_1 = v_2$ and $(\sigma_1', \sigma_2') \in R$.

Theorem 8.4.5 lifts properties about interpretations to properties of evaluation over those abstract machines – in particular, we only need to establish that *assignment* is correct for an interpretation to establish correctness of the corresponding evaluator/compiler. In practice we found this significantly reduces boilerplate proof code that is otherwise currently necessary in F* due to a lack of automated induction.

Given two interpretations $\mathscr{I}, \mathscr{I}'$, we say elements σ and σ' of $\mathscr{I}$ and $\mathscr{I}'$ are *observationally equivalent* with respect to a supplied set of initial values $s \in \mathbf{State}$ if for all $i \in \mathbb{N}$, $\mathsf{eval}_{\mathscr{I}}(\sigma, i, s) = \mathsf{eval}_{\mathscr{I}'}(\sigma', i, s)$. We say $\sigma \sim_s \sigma'$ if σ and σ' are observationally equivalent with respect to s. As observational equivalence of two domain elements σ, σ' implies that any location in scope has the same valuation in either interpretation, it suffices to show that any compiled circuit is observationally equivalent to the standard interpretation. The following lemmas are used along with Theorem 8.4.5 to establish this fact for the default and eager-cleanup interpretations – a similar lemma is proven in the implementation of REVERC for the crush mode.

Lemma 8.4.6. *Let σ, σ' be elements of $\mathscr{I}_{standard}$ and $\mathscr{I}_{circuit}$, respectively. For all $l \in \mathbb{N}, B \in \mathbf{BExp}, s \in \mathbf{State}$, if $\sigma \sim_s \sigma'$ and $s(i) = 0$ whenever $i \in \xi$, then*

$$\mathsf{assign}_{standard}(\sigma, l, B) \sim_s \mathsf{assign}_{circuit}(\sigma', l, B).$$

Moreover, the ancilla heap remains 0-filled.

We say that $(\rho, C, \xi) \in D_{circuit}$ is *valid* with respect to $s \in \mathbf{State}$ if and only if $s(i) = 0$ for all $i \in \xi$. For elements of D_{GC} the validity conditions are more involved, so we introduce a relation, $\mathcal{V} \subseteq D_{GC} \times \mathbf{State}$, defining the set of valid domain elements:

$$((\rho, C, \xi, \kappa), s) \in \mathcal{V} \iff \forall i \in \xi, s(i) = 0 \wedge \forall l, l' \in \mathsf{dom}(\rho), \rho(l) \neq \rho(l') \\ \wedge \forall i \in \mathsf{cod}(\rho), [\![i \oplus \kappa(i)]\!]([\![C]\!]s) = s(i)$$

Informally, $\mathcal{V}$ specifies that all bits on the heap have initial value 0, that ρ is a one-to-one mapping, and that for every active bit i, XORing i with $\kappa(i)$ returns the initial value of i – that is, $i \oplus \kappa(i)$ *cleans* i.

Lemma 8.4.7. *Let σ, σ' be elements of $\mathscr{I}_{standard}$ and $\mathscr{I}_{GC}$, respectively. For all $l \in \mathbb{N}, B \in$ **BExp**, $s \in$ **State**, if $\sigma \sim_s \sigma'$ and $(\sigma', s) \in \mathcal{V}$, then*

$$\mathsf{assign}_{standard}(\sigma, l, B) \sim_s \mathsf{assign}_{GC}(\sigma', l, B).$$

Moreover, $(\mathsf{assign}_{GC}(\sigma', l, B), s) \in \mathcal{V}$.

By setting the relation R_{GC} as

$$(\sigma_1, \sigma_2) \in R_{GC} \iff \sigma_2 \in \mathcal{V} \wedge \sigma_1 \sim_{s_0} \sigma_2$$

for $\sigma_1 \in D_{standard}$, by Theorem 8.4.5 and Lemma 8.4.7 it follows that partial evaluation/-compilation preserves observational equivalence between $\mathscr{I}_{standard}$ and $\mathscr{I}_{GC}$. A similar result follows for $\mathscr{I}_{circuit}$.

8.5 Experiments

We ran experiments to compare the bit, gate and Toffoli counts of circuits compiled by ReVerC to the original Revs compiler. The number of Toffoli gates is distinguished as such gates are generally much more costly than NOT and controlled-NOT gates. We compiled circuits for various arithmetic and cryptographic functions written in Revs using both compilers and reported the results in Table 8.1. The experimental set up is the same as previous chapters.

The results show that both compilers are more-of-less evenly matched in terms of bit counts across both modes, despite ReVerC being certifiably correct. ReVerC's eager cleanup mode never used more bits than the default mode, as expected, and in half of the benchmarks reduced the number of bits. Moreover, in the cases of the carryRippleAdder and MD5 benchmarks, ReVerC's eager cleanup mode produced circuits with significantly fewer bits than either of Revs' modes. On the other hand, Revs saw dramatic decreases in bit numbers for carryLookahead and SHA-2 compared to ReVerC.

While the results show there is clearly room for optimization of gate counts, they appear consistent with other verified compilers (e.g., [Ler06]) which take some performance hit

Table 8.1: Bit and gate counts for Revs and ReVerC in default and eager cleanup modes. Entries with the fewest bits used or Toffolis are bolded.

Benchmark	Revs			Revs (eager)			ReVerC			ReVerC (eager)		
	bits	gates	Toffolis	bits	gates	Toffolis	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdd$_{32}$	129	281	**62**	129	467	124	128	281	**62**	**113**	361	90
carryRippleAdd$_{64}$	257	569	**126**	257	947	252	256	569	**126**	**225**	745	186
mult$_{32}$	128	6,016	4,032	128	6,016	4,032	128	6,016	4,032	128	6,016	4,032
mult$_{64}$	256	24,320	16,256	256	24,320	16,256	256	24,320	16,256	256	24,320	16,256
carryLookahead$_{32}$	160	345	**103**	**109**	1,036	344	165	499	120	146	576	146
carryLookahead$_{64}$	424	1,026	**307**	**271**	3,274	1,130	432	1,375	336	376	1,649	428
modAdd$_{32}$	65	188	62	65	188	62	65	188	62	65	188	62
modAdd$_{64}$	129	380	126	129	380	126	129	380	126	129	380	126
cucarroAdder$_{32}$	65	98	32	65	98	32	65	98	32	65	98	32
cucarroAdder$_{64}$	129	194	64	129	194	64	129	194	64	129	194	64
ma4	17	24	8	17	24	8	17	24	8	17	24	8
SHA-2	449	1,796	**594**	**353**	2,276	754	452	1,796	**594**	449	1,796	**594**
MD5	7,841	81,664	**27,520**	7,905	82,624	27,968	4,833	70,912	**27,520**	**4,769**	70,912	**27,520**

when compared to unverified compilers. In particular, unverified compilers may use more aggressive optimizations due to the increased ease of implementation and the lack of a requirement to prove their correctness compared to certified compilers. In some cases, the optimizations are even known to not be correct in all possible cases, as in the case of fast arithmetic and some loop optimization passes in the GNU C Compiler [GCC16].

8.6 Related work

Reversible circuit compilation Due to the reversibility requirement of quantum computing, quantum programming languages and compilers typically have methods for generating reversible circuits. Quantum programming languages typically allow compilation of classical, irreversible code in order to minimize the effort of porting existing code into the quantum domain. In QCL [Ö00], "pseudo-classical" operators – classical functions meant to be run on a quantum computer – are written in an imperative style and compiled with automatic ancilla management. As in Revs, such code manipulates registers of bits, splitting off sub-registers and concatenating them together. The more recent Quipper [GLR+13] automatically generates reversible circuits from classical code by a process called *lifting*: using Haskell metaprogramming, Quipper lifts the classical code to the reversible domain with automated ancilla management. However, little space optimization is performed [SVM+17].

Other approaches to high-level synthesis of reversible circuits are based on writing code in *reversible programming languages* – that is, the programs themselves are written in

a reversible way. Perhaps most notable in this line of research is Janus [YG07] and its various extensions [WOD10, Tho12, Per14]. These languages typically feature a *reversible update* and some bi-directional control operators, such as if statements with exit assertions. Due to the presence of dynamic control in the source language, such languages typical target reversible instruction set architectures like Pendulum [Vie95], as opposed to the combinational circuits that ReVerC and in general, quantum compilers target.

Verification As noted in the previous chapter, verification of reversible circuits has been studied from the viewpoint of checking equivalence against a benchmark circuit or specification [WGMD09, YM10]. This can double as both *program verification* and *translation validation*, but every compiled circuit needs to be verified separately. Moreover, a program that is easy to formally verify may be translated into a circuit with hundreds of bits, and is thus very difficult to verify. Recent work has shown progress towards verification of more general properties of reversible and quantum circuits –for example, via model checking [APTZ16] – but to the authors' knowledge, no verification of a reversible circuit compiler has yet been carried out. By contrast, many compilers for general purpose programming languages have been formally verified in recent years – most famously, the CompCert optimizing C compiler [Ler06], written and verified in Coq. Since then, many other compilers have been developed and verified in a range of languages and logics including Coq, HOL, F*, etc., with features such as shared memory [BSDA14], functional programming [Chl10, FSC+13] and modularity [PA14, NHK+15].

Part IV

Conclusion

Chapter 9

Conclusion

Throughout this thesis we have studied aspects of quantum circuit design for quantum compilation. We looked at the questions of both optimization of quantum circuits, and their functional verification. On the optimization side, we developed a general framework for circuit optimization which first applies phase-folding to merge phase gates and group them into individual CNOT-dihedral subcircuits – these subcircuits can then be optimally or sub-optimally synthesized for a variety of cost functions. We studied this optimal synthesis problem from the perspective of R_Z- and CNOT-count minimization, which were shown to reduce to identifiable combinatorial problems in certain cases. Using these characterizations, we developed concrete optimization algorithms which scale to large circuits and reduce resource costs of real-world quantum circuits by 42% (T-count) and 22% (CNOT-count).

On the verification side, we developed an algorithm for the verification of the unitary operator or partial isometry implemented by a circuit over Clifford hierarchy gates and ancillary qubits. The algorithm was shown to scale to some large circuits of up to 200 qubits, exceeding previous equivalence checking results as well as the performance of recent simulators on a Hidden Shift algorithm. Going beyond functional verification, a compiler for strictly reversible circuits was formally verified. Together, the methods of this thesis – implemented in the reversible circuit compiler ReVerC[1] and quantum compiler backend Feynman[2] – form a stack of tools for compiling optimized, verified quantum circuits. Between these pieces of software, a classical program can be compiled to a certified correct reversible circuit, then decomposed into a Clifford+T circuit, optimized, and verified against the original.

[1] https://github.com/msr-quarc/ReVerC
[2] https://github.com/meamy/feynman

9.1 Future work

This thesis represents one microscopic step towards the herculean task of making quantum computing a reality. Many more steps will need to be taken before a scalable, reliable set of circuit design tools are developed, let alone a programmable, accurate universal quantum computer. And so, we close by discussing some directions those steps might take.

Path integrals & Formal models A major theme of this thesis was the use of path integrals and phase polynomials as a formal model for the analysis of *pure* quantum circuits. A natural direction of future research is to develop a framework for similar analyses applied to more general quantum circuits – for instance, those involving measurements and classical control. Indeed, recent research [BK18] has combined the intuition of phase polynomials with graphical calculi (e.g, [CD11]) to develop new formal models applicable to the full range of quantum computation, while retaining some of the insight developed here.

Optimization One of the natural next steps for the optimization algorithms present in this thesis – in particular, the phase-folding framework – is to extend it in a natural way to quantum *programs* as opposed to quantum circuits. One way to go about this may be to develop analogous models for such programs as noted above. On the other hand, it may be possible to forgo a direct path integral model of more general quantum programs by using classical techniques of abstraction and collecting semantics to soundly lift these purely unitary optimizations to such programs.

Many questions still remain regarding both R_Z-count and CNOT-count optimizations. For R_Z-counts, the question of a exact upper bound – that is, the exact covering radius of higher order Reed-Muller codes – for CNOT-dihedral circuits remains. More significantly, our methods give a very loose upper bound of $O(k \cdot n^2)$ T-gates in a Clifford+T circuit containing k Hadamard gates, which can no doubt be brought down by examining the (more difficult) problem of T or R_Z gate optimization in Clifford+R_Z circuits. Indeed, recent work by Heyfron and Campbell [HC18] has brought this down to $O(n^2 + k^2)$ by pushing Hadamard gates onto ancillas at the beginning and end of the circuit, and further given a new Reed-Muller decoder which empirically achieve this scaling. It remains a question for future research to determine the relationship between ancillas, phase and Hadamard gates, and moreover to find methods – if they exist – with $O(n^2 + k^2)$ T gates that do not require ancillas.

Regarding CNOT minimization, even in the restricted case of CNOT-dihedral circuits, the exact complexity of CNOT-minimization remains unknown. The first order of business

is to establish the difficulty in general for the parity network synthesis problem, possibly by showing that certain cases reduce to the fixed-target version of the problem. On the other hand, it may yet turn out to be possible to find an efficient algorithm for synthesizing minimal parity networks. While we have presented an effective heuristic algorithm for synthesizing parity networks, better heuristics may also exist. As an additional point of consideration, physical chip designs typically have limited connectivity and can only apply CNOT gates between connected qubits. While arbitrary CNOT gates can be implemented with a sequence of CNOT gates having length at most proportional to the diameter of the connectivity graph, it nonetheless remains possible that a better circuit may be found by synthesizing one directly for a given topology. A natural and important direction for future research is then to find CNOT optimization algorithms which take into account such connectivity constraints, a problem which likewise appears to be computationally intractable [HND17].

Verification The functional verification work presented in this thesis represents a preliminary step towards a fully-automated system of formal specification and verification for quantum circuits, and as such there are many issues for future work to address. First and foremost is to develop a concrete syntax – or *specification language* – so that libraries of *portable, verified* circuits may be developed, greatly reducing the difficulty of implementing quantum algorithms. Specification languages form the backbone of most modern hardware design workflows, and so it is highly desirable – and in particular, not yet well studied – to develop such languages for quantum circuit design. A related direction, motivated by our experience writing path integral rewriting proofs "by hand," is to implement path integral-based verification in an interactive proof assistant. This could further extend the applicability of this framework to inductive proofs, allowing the verification of entire families of quantum circuits. As in classic interactive provers, the rewrite rules and verification algorithm we develop here could ostensibly be the starting point for proof automation, just as the F* language uses SMT solvers to automatically prove verification sub-goals.

On the algorithmic side many improvements can yet be made to the verification algorithm. It was noted that the algorithm has high memory usage, due to the cubic (or higher) degree of the phase polynomial. Representing phase polynomials by their Fourier expansion, breaking normalization but saving space, may be useful time-space trade-off for difficult to verify circuits. On the other hand, specifications of algorithms as well as the intermediate states of the path integrals can grow exponentially large – in these cases, it remains to be seen whether the use of relational representations, which have proven useful for saving space in classical verification methods such as model checking, may be an effective solution here. Other open questions regard finding new reduction rules, particularly finding a complete set

of rules for Clifford+T circuits, as well as to improve methods for completing verification once no more reductions can be made. An interesting direction for this latter problem may be to used *algebraic decision diagrams* [NWM+16], which have proven remarkably useful for uniquely and compactly representing finite functions in the classical world. Indeed, recent work has hinted at the use of algebraic decision diagrams in quantum circuit simulation and verification [ZW18].

The development of a verified compiler for *full* quantum circuits – rather than strictly reversible circuits – is also left for future work. However, in contrast to reversible circuits where the programming language is naturally any classical programming language, the exact nature of this problem depends on what we mean by a *quantum programming language*. If we take a quantum programming language to simply mean a circuit description language, the job of the compiler is massively simplified; in most cases, the compiler effectively unrolls a program to a straight line circuit, possibly decomposing some gates according to their implementation in a particular gate basis. The natural verification questions for such a compiler, beyond equivalence checking of gate decompositions, are not yet clear. As quantum programs are naturally probabilistic, a more interesting question may be *does the compiler preserve probabilities up to some error rate?* A natural direction to look in this case is the classical work done on *reliability* (e.g., [CMR13]). More concretely, improvements can be made to the compilation methods of ReVerC– particularly its space optimizations – as well as the verification of the interpreter with respect to a more formal relational implementation of the semantics of Revs.

References

[AADS16] Nabila Abdessaied, Matthew Amy, Rolf Drechsler, and Mathias Soeken. Complexity of Reversible Circuits and their Quantum Implementations. *Theoretical Computer Science*, 618:85–106, 2016. doi:10.1016/j.tcs.2016.01.011.

[AAM18] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the CNOT-complexity of CNOT-PHASE Circuits. *Quantum Science and Technology*, 4(1):015002, 2018. doi:10.1088/2058-9565/aad8ca.

[AASD16] Nabila Abdessaied, Matthew Amy, Mathias Soeken, and Rolf Drechsler. Technology Mapping of Reversible Circuits to Clifford+T Quantum Circuits. In *Proceedings of the 46th International Symposium on Multiple-Valued Logic*, ISMVL '16, pages 150–155, 2016. doi:10.1109/ISMVL.2016.33.

[ABL+18] Divesh Aggarwal, Gavin Brennen, Troy Lee, Miklos Santha, and Marco Tomamichel. Quantum Attacks on Bitcoin, and How to Protect Against Them. *Ledger*, 3(0), 2018. doi:10.5195/ledger.2018.127.

[AC04] Samson Abramsky and Bob Coecke. A Categorical Semantics of Quantum Protocols. In *Proceedings of the 19th annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 415–425, 2004. doi:10.1109/LICS.2004.1.

[ACJR17] Matthew Amy, Jianxin Chen, and Neil J. Ross. A Finite Presentation of CNOT-Dihedral Operators. In *Proceedings of the 14th International Conference on Quantum Physics and Logic*, QPL '17, pages 84–97, 2017. doi:10.4204/EPTCS.266.5.

[ADH97] Leonard Adleman, Jonathan DeMarrais, and Ming-Deh A. Huang. Quantum Computability. *SIAM Journal on Computing*, 26(5):1524–1540, 1997. doi:10.1137/S0097539795293639.

[AJO16] Jonas T. Anderson and Tomas Jochym-O'Connor. Classification of Transversal Gates in Qubit Stabilizer Codes. *Quantum Information & Computation*, 16(9-10):771–802, 2016. doi:10.26421/QIC16.9-10.

[AM16] Matthew Amy and Michele Mosca. T-count Optimization and Reed-Muller Codes. *arXiv preprint*, 2016, arXiv:1601.07363.

[AMG+16] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. Estimating the Cost of Generic Quantum Preimage Attacks on SHA-2 and SHA-3. In *Proceedings of the 24th Conference on Selected Areas in Cryptography*, SAC '16, pages 317–337, 2016. doi:10.1007/978-3-319-69453-5_18.

[AMM14] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-depth optimization of Clifford+T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014. doi:10.1109/TCAD.2014.2341953.

[AMMR13] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013. doi:10.1109/TCAD.2013.2244643.

[Amy18] Matthew Amy. Towards Large-Scale Functional Verification of Universal Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic*, QPL '18, 2018. doi:10.4204/EPTCS.287.1.

[APTZ16] Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. Towards Quantum Programs Verification: From Quipper Circuits to QPMC. In *Proceedings of the 8th International Conference on Reversible Computation*, RC '16, pages 213–219, 2016. doi:10.1007/978-3-319-40578-0_16.

[ARS17] Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified Compilation of Space-Efficient Reversible Circuits. In *Proceedings of the 29th International Conference on Computer Aided Verification*, CAV '17, pages 3–21, 2017. doi:10.1007/978-3-319-63390-9_1.

[ASD14] Nabila Abdessaied, Mathias Soeken, and Rolf Drechsler. Quantum Circuit Optimization by Hadamard Gate Reduction. In *Proceedings of the 6th International Conference on Reversible Computation*, RC '14, pages 149–162, 2014. doi:10.1007/978-3-319-08494-7_12.

[BCM08] Pedro Baltazar, Rohit Chadha, and Paulo Mateus. Quantum Computation Tree Logic – Model Checking and Complete Calculus. *International Journal of Quantum Information*, 06(02):219–236, 2008. doi:10.1142/S0219749908003530.

[Ben73] Charles H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17:525–532, 1973. doi:10.1147/rd.176.0525.

[Ben89] Charles H. Bennett. Time/Space Trade-Offs for Reversible Computation. *SIAM Journal on Computing*, 18(4):766–776, 1989. doi:10.1137/0218053.

[BG16] Sergey Bravyi and David Gosset. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Physical Review Letters*, 116:250501, 2016. doi:10.1103/PhysRevLett.116.250501.

[BGK+16] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Rewriting Modulo Symmetric Monoidal Structure. In *Proceedings of the 31st ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 710–719, 2016. doi:10.1145/2933575.2935316.

[BH12] Sergey Bravyi and Jeongwan Haah. Magic-State Distillation with Low Overhead. *Physical Review A*, 86:052329, 2012. doi:10.1103/PhysRevA.86.052329.

[BK05] Sergey Bravyi and Alexei Kitaev. Universal Quantum Computation with Ideal Clifford Gates and Noisy Ancillas. *Physical Review A*, 71:022316, 2005. doi:10.1103/PhysRevA.71.022316.

[BK18] Miriam Backens and Aleks Kissinger. ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity. In *Proceedings of the 15th International Conference on Quantum Physics and Logic*, QPL '18, pages 23–42, 2018. doi:10.4204/EPTCS.287.2.

[BKV18] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an Online Proof Assistant for Higher-dimensional Rewriting. *Logical Methods in Computer Science*, Volume 14, Issue 1, 2018. doi:10.23638/LMCS-14(1:8)2018.

[BMvT06] Elwyn Berlekamp, Robert McEliece, and Hank van Tilborg. On the Inherent Intractability of Certain Coding Problems. *IEEE Transactions on Information Theory*, 24(3):384–386, September 2006. doi:10.1109/TIT.1978.1055873.

[BRS15] Alex Bocharov, Martin Roetteler, and Krysta M. Svore. Efficient Synthesis of Universal Repeat-Until-Success Quantum Circuits. *Physical Review Letters*, 114:080502, 2015. doi:10.1103/PhysRevLett.114.080502.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew. Appel. Verified Compilation for Shared-Memory C. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, ESOP '14, pages 107–127, 2014. doi:10.1007/978-3-642-54833-8_7.

[BU11] David Buchfuhrer and Christopher Umans. The Complexity of Boolean Formula Minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011. doi:10.1016/j.jcss.2010.06.011.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum Complexity Theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. doi:10.1137/S0097539796300921.

[CAB12] Earl T. Campbell, Hussain Anwar, and Dan E. Browne. Magic-State Distillation in All Prime Dimensions Using Quantum Reed-Muller Codes. *Physical Review X*, 2:041021, 2012. doi:10.1103/PhysRevX.2.041021.

[CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv preprint*, 2017, arXiv:1707.03429.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977. doi:10.1145/512950.512973.

[CD11] Bob Coecke and Ross Duncan. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics*, 13(4):043016, 2011. doi:10.1088/1367-2630/13/4/043016.

[CDKW16] Bob Coecke, Ross Duncan, Aleks Kissinger, and Quanlong Wang. Generalised Compositional Theories and Diagrammatic Reasoning. In G. Chiribella and R. W. Spekkens, editors, *Quantum Theory: Informational Foundations and Foils*, pages 309–366. Springer Netherlands, Dordrecht, 2016. doi:10.1007/978-94-017-7303-4_10.

[CFH97] David G. Cory, Amr F. Fahmy, and Timothy F. Havel. Ensemble quantum computing by NMR spectroscopy. *Proceedings of the National Academy of Sciences*, 94(5):1634–1639, 1997. doi:10.1073/pnas.94.5.1634.

[CGK17] Shawn X. Cui, Daniel Gottesman, and Anirudh Krishna. Diagonal Gates in the Clifford Hierarchy. *Physical Review A*, 95:012329, 2017. doi:10.1103/PhysRevA.95.012329.

[CH17a] Earl T. Campbell and Mark Howard. Unified Framework for Magic State Distillation and Multiqubit Gate Synthesis with Reduced Resource Cost. *Physical Review A*, 95:022316, 2017. doi:10.1103/PhysRevA.95.022316.

[CH17b] Earl T. Campbell and Mark Howard. Unifying gate synthesis and magic state distillation. *Physical Review Letters*, 118:060501, 2017. doi:10.1103/PhysRevLett.118.060501.

[Chl10] Adam Chlipala. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 93–106, 2010. doi:10.1145/1706299.1706312.

[CL92] Gérard D. Cohen and Simon N. Litsyn. On the Covering Radius of Reed-Muller Codes. *Discrete Mathematics*, 106:147 – 155, 1992. doi:http://dx.doi.org/10.1016/0012-365X(92)90542-N.

[Cla01] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology and Göteborg University, 2001.

[CM09] Sephen L. Campbell and Carl D. Meyer. *Generalized Inverses of Linear Transformations*. Classics in Applied Mathematics. SIAM, 2009. doi:10.1137/1.9780898719048.

[CMR13] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, 2013. doi:10.1145/2509136.2509546.

[CO16] Earl T. Campbell and Joe O'Gorman. An Efficient Magic State Approach to Small Angle Rotations. *Quantum Science and Technology*, 1(1):015007, 2016. doi:10.1088/2058-9565/1/1/015007.

[Coq17] The Coq Proof Assistant, version 8.7.0, 2017. doi:10.5281/zenodo.1028037.

[CZ95] J. Ignacio Cirac and Peter Zoller. Quantum Computations with Cold Trapped Ions. *Physical Review Letters*, 74:4091–4094, 1995. doi:10.1103/PhysRevLett.74.4091.

[DCP15] Guillaume Duclos-Cianci and David Poulin. Reducing the Quantum-computing Overhead with Complex Gate Distillation. *Physical Review A*, 91:042315, 2015. doi:10.1103/PhysRevA.91.042315.

[Deu85] David Deutsch. Quantum theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97–117, 1985. doi:10.1098/rspa.1985.0070.

[DHM+05] Christopher M. Dawson, Andrew P. Hines, Duncan Mortimer, Henry L. Haselgrove, Michael A. Nielsen, and Tobias J. Osborne. Quantum Computing and Polynomial Equations over the Finite Field $\mathbb{Z}_2$. *Quantum Information & Computation*, 5(2):102–112, 2005. doi:10.26421/QIC5.2.

[DL13] Ross Duncan and Maxime Lucas. Verifying the Steane Code with Quantomatic. In *Proceedings of the 10th International Conference on Quantum Physics and Logic*, QPL '13, pages 33–49, 2013. doi:10.4204/EPTCS.171.4.

[DLF+16] Shantanu Debnath, Norbert M. Linke, Caroline Figgatt, Kevin Landsman, Kevin Wright, and Chris Monroe. Demonstration of a Small Programmable Quantum Computer with Atomic Qubits. *Nature*, 536(7614):63–66, 2016. doi:10.1038/nature18648.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, 1982. doi:10.1145/582153.582176.

[DN06] Christopher M. Dawson and Michael A. Nielsen. The Solovay-Kitaev Algorithm. *Quantum Information & Computation*, 6(1):81–95, 2006. doi:10.26421/QIC6.1.

[Dum04] Ilya Dumer. Recursive Decoding and its Performance for Low-rate Reed-Muller Codes. *IEEE Transactions on Information Theory*, 50(5):811–823, 2004. doi:10.1109/TIT.2004.826632.

[EK09] Bryan Eastin and Emanuel Knill. Restrictions on Transversal Encoded Quantum Gate Sets. *Physical Review Letters*, 102:110502, 2009. doi:10.1103/PhysRevLett.102.110502.

[EKP85] Jarmo Ernvall, Jyrki Katajainen, and Martti Penttonen. NP-Completeness of the Hamming Salesman Problem. *BIT Numerical Mathematics*, 25(1):289–292, 1985. doi:10.1007/BF01935007.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the Tenth annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 169–184, 1995. doi:10.1145/217838.217858.

[Fey82] Richard Feynman. Simulating Physics with Computers. *International Journal of Theoretical Physics*, 21:467–488, 1982. doi:10.1007/BF02650179.

[FH65] Richard P. Feynman and Albert R. Hibbs. *Quantum Mechanics and Path Integrals*. International series in pure and applied physics. McGraw-Hill, 1965.

[FHTZ15] Yuan Feng, Ernst Moritzb Hahn, Andrea Turrini, and Lijun Zhang. QPMC: A Model Checker for Quantum Programs and Protocols. In *Proceedings of the 19th International Symposium on Formal Methods*, FM '15, pages 265–272, 2015. doi:10.1007/978-3-319-19249-9_17.

[FMMC12] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface Codes: Towards Practical Large-scale Quantum Computation. *Physical Review A*, 86:032324, 2012. doi:10.1103/PhysRevA.86.032324.

[FR99] Lance Fortnow and John Rogers. Complexity Limitations on Quantum Computation. *Journal of Computer and System Sciences*, 59(2):240–252, 1999. doi:10.1006/jcss.1999.1651.

[FSC+13] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully Abstract Compilation to JavaScript. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 371–384, 2013. doi:10.1145/2429069.2429114.

[FYY13] Yuan Feng, Nengkun Yu, and Mingsheng Ying. Model Checking Quantum Markov Chains. *Journal of Computer and System Sciences*, 79(7):1181–1198, 2013. doi:10.1016/j.jcss.2013.04.002.

[GC97] Neil A. Gershenfeld and Isaac L. Chuang. Bulk Spin-Resonance Quantum Computation. *Science*, 275(5298):350–356, 1997. doi:10.1126/science.275.5298.350.

[GCC16] Using the GNU Compiler Collection. Free Software Foundation, Inc., 2016. URL https://gcc.gnu.org/onlinedocs/gcc/.

[GD17] Liam Garvie and Ross Duncan. Verifying the Smallest Interesting Colour Code with Quantomatic. In *Proceedings of the 14th International Conference on Quantum Physics and Logic*, QPL '17, pages 147–163, 2017. doi:10.4204/EPTCS.266.10.

[GLR+13] Alexander S. 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 '13, pages 333–342, 2013. doi:10.1145/2491956.2462177.

[GLRS16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's Algorithm to AES: Quantum Resource Estimates. In *Proceedings of the 7th International Workshop on Post-Quantum Cryptography*, PQCrypto '16, pages 29–43, 2016. doi:10.1007/978-3-319-29360-8_3.

[GNP08] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. QMC: A Model Checker for Quantum Systems. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 543–547, 2008. doi:10.1007/978-3-540-70545-1_51.

[Got98] Daniel Gottesman. The Heisenberg Representation of Quantum Computers. In *International Conference on Group Theoretic Methods in Physics*, page 9807006, 1998, arXiv:quant-ph/9807006.

[Gra53] Frank Gray. Pulse Code Communication, March 17 1953. US Patent 2,632,058.

[HB09] Peter Mark Hines and Sam Braunstein. The Structure of Partial Isometries. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 361–388. Cambridge University Press, 11 2009. doi:10.1017/CBO9781139193313.

[HC18] Luke E. Heyfron and Earl T. Campbell. An Efficient Quantum Compiler that Reduces T Count. *Quantum Science and Technology*, 4(1):015004, 2018. doi:10.1088/2058-9565/aad604.

[HFDM12] Clare Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. Surface Code Quantum Computing by Lattice Surgery. *New Journal of Physics*, 14(12):123011, 2012. doi:10.1088/1367-2630/14/12/123011.

[HND17] Daniel Herr, Franco Nori, and Simon J. Devitt. Optimization of Lattice Surgery is NP-hard. *npj Quantum Information*, 3(1):35, 2017. doi:10.1038/s41534-017-0035-1.

[HR68] Peter L. Hammer and Sergiu Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Ökonometrie und Unternehmensforschung. Springer-Verlag, 1968.

[IAR13] IARPA. Quantum Computer Science, 2013. URL `https://www.iarpa.gov/index.php/research-programs/qcs`.

[IKY02] Kazuo Iwama, Yahiko Kambayashi, and Shigeru Yamashita. Transformation Rules for Designing CNOT-based Quantum Circuits. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 419–424, 2002. doi:10.1145/513918.514026.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[JPK+15] Ali JavadiAbhari, 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(C):2–17, 2015. doi:10.1016/j.parco.2014.12.001.

[KIQ+18] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry Fields: A Software Platform for Photonic Quantum Computing. *arXiv preprint*, 2018, arXiv:1804.03159.

[KLM01] Emanuel Knill, Raymond Laflamme, and Gerard J. Milburn. A Scheme for Efficient Quantum Computation with Linear Optics. *Nature*, 409:46–52, 02 2001. doi:10.1038/35051009.

[KLM07] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[KLZ96] Emanuel Knill, Raymond Laflamme, and Wojciech Zurek. Threshold Accuracy for Quantum Computation. *arXiv preprint*, 1996, arXiv:quant-ph/9610011.

[KMM13a] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Asymptotically Optimal Approximation of Single Qubit Unitaries by Clifford and T Circuits Using a Constant Number of Ancillary Qubits. *Physical Review Letters*, 110:190502, 2013. doi:10.1103/PhysRevLett.110.190502.

[KMM13b] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Fast and Efficient Exact Synthesis of Single-qubit Unitaries Generated by Clifford and T Gates. *Quantum Information & Computation*, 13(7-8):607–630, 2013. doi:10.26421/QIC13.7-8.

[KZ15] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A Proof Assistant for Diagrammatic Reasoning. In *Proceedings of the International Conference on Automated Deduction*, CADE-25, pages 326–336, 2015. doi:10.1007/978-3-319-21401-6_22.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004. doi:10.1109/CGO.2004.1281665.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th annual ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, 2014. doi:10.1145/2594291.2594334.

[LC13] Andrew J. Landahl and Chris Cesare. Complex Instruction Set Computing Architecture for Performing Accurate Quantum Z Rotations with Less Magic. *arXiv preprint*, 2013, arXiv:1302.3240.

[LD98] Daniel Loss and David P. DiVincenzo. Quantum Computation with Quantum Dots. *Physical Review A*, 57:120–126, 1998. doi:10.1103/PhysRevA.57.120.

[Ler06] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT International Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, 2006. doi:10.1145/1111037.1111042.

[Llo96] Seth Lloyd. Universal Quantum Simulators. *Science*, 273(5278):1073–1078, 1996. doi:10.1126/science.273.5278.1073.

[LWZ+17] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. $Q|SI\rangle$: A Quantum Programming Environment. *arXiv preprint*, 2017, arXiv:1710.09500.

[Mas16] Dmitri Maslov. Advantages of using Relative-Phase Toffoli Gates with an Application to Multiple Control Toffoli Optimization. *Physical Review A*, 93:022311, 2016. doi:10.1103/PhysRevA.93.022311.

[Mit91] John C. Mitchell. Type Inference with Simple Subtypes. *Journal of Functional Programming*, 1:245–285, 1991. doi:10.1017/S0956796800000113.

[Mon17] Ashley Montanaro. Quantum Circuits and Low-degree Polynomials over $\mathbb{F}_2$. *Journal of Physics A: Mathematical and Theoretical*, 50(8):084002, 2017. doi:10.1088/1751-8121/aa565f.

[MP67] John Mccarthy and James Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 33–41. American Mathematical Society, 1967. URL http://jmc.stanford.edu/articles/mcpain.html.

[MS78] Florence J. MacWilliams and Neil J.A. Sloane. *The Theory of Error Correcting Codes*. North-Holland Mathematical Library, Volume 16. North-Holland Publishing Company, 1978.

[Mul54] David E. Muller. Application of Boolean Algebra to Switching Circuit Design and to Error Detection. *Transactions of the I.R.E. Professional Group on Electronic Computers*, EC-3(3):6–12, Sept 1954. doi:10.1109/IREPGELC.1954.6499441.

[MW72] Robin Milner and Richard W. Weyhrauch. Proving Compiler Correctness in a Mechanised Logic. *Machine Intelligence*, 7:51–73, 1972. URL http://www.cs.umd.edu/~hjs/pubs/compilers/archive/mi72-mil-wey.pdf.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge Series on Information and the Natural Sciences. Cambridge University Press, 2000.

[NHK+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pages 166–178, 2015. doi:10.1145/2784731.2784764.

[NRS+18] 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. doi:10.1038/s41534-018-0072-4.

[NWM+16] Philipp Niemann, Robert Wille, David M. Miller, Mitchell A. Thornton, and Rolf Drechsler. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):86–99, Jan 2016. doi:10.1109/TCAD.2015.2459034.

[Ö00] Bernhard Ömer. *Quantum Programming in QCL*. Master's thesis, Technical University of Vienna, 2000. URL http://tph.tuwien.ac.at/~oemer/qcl.html.

[Ö03] Bernhard Ömer. *Structured Quantum Programming*. PhD thesis, Technical University of Vienna, 2003. URL http://tph.tuwien.ac.at/~oemer/qcl.html.

[O'D14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. doi:10.1017/CBO9781139814782.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

[PA14] James T. Perconti and Amal Ahmed. Verifying an Open Compiler Using Multi-language Semantics. *ACM Transactions on Programming Languages and Systems*, 8410:128–148, 2014. doi:10.1007/978-3-642-54833-8_8.

[Pay18] Jennifer Paykin. Private Communication, 2018.

[Per08] Simon Perdrix. Quantum Entanglement Analysis Based on Abstract Interpretation. In *Proceedings of the 15th International Symposium on Static Analysis*, SAS '18, pages 270–282, 2008. doi:10.1007/978-3-540-69166-2_18.

[Per14] Kalyan S. Perumalla. *Introduction to Reversible Computing*. CRC Press, 2014.

[PMH08] Ketan N. Patel, Igor L. Markov, and John P. Hayes. Optimal Synthesis of Linear Reversible Circuits. *Quantum Information & Computation*, 8(3):282–294, 2008. doi:10.26421/QIC8.3-4.

[Pot96] François Pottier. Simplifying Subtyping Constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 122–133, 1996. doi:10.1145/232627.232642.

[Pre18] John Preskill. Quantum Computing in the NISQ Era and Beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

[PRS15] Alex Parent, Martin Roetteler, and Krysta M Svore. Reversible Circuit Compilation with Space Constraints. *arXiv preprint*, 2015, arXiv:1510.00377.

[PS14] Adam Paetznick and Krysta M. Svore. Repeat-until-success: Non-deterministic Decomposition of Single-qubit Unitaries. *Quantum Information & Computation*, 14(15-16):1277–1301, 2014. doi:10.26421/QIC14.15-16.

[PZ04] Jens Palsberg and Tian Zhao. Type Inference for Record Concatenation and Subtyping. *Information and Computation*, 189(1):54 – 86, 2004. doi:10.1016/j.ic.2003.10.001.

[RÏ10] Martin Rötteler. Quantum Algorithms for Highly Non-linear Boolean Functions. In *Proceedings of the 21st annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 448–457, 2010. doi:10.1137/1.9781611973075.37.

[Ree54] Irving Reed. A Class of Multiple-error-correcting Codes and the Decoding Scheme. *Transactions of the I.R.E. Professional Group on Information Theory*, 4(4):38–49, 1954. doi:10.1109/TIT.1954.1057465.

[RPZ17] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings of the 14th International Conference on Quantum Physics and Logic*, QPL '17, pages 119–132, 2017. doi:10.4204/EPTCS.266.8.

[RS16] Neil J. Ross and Peter Selinger. Optimal Ancilla-free Clifford+T Approximation of Z-rotations. *Quantum Information & Computation*, 16(11-12):901–953, 2016. doi:10.26421/QIC16.11-12.

[SB16] Peter Selinger and Xiaoning Bian. Relations for 2-qubit Clifford+T Operator Group, 2016. URL https://www.mathstat.dal.ca/~xbian/talks/slide_cliffordt2.pdf.

[SCZ16] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. *arXiv preprint*, 2016, arXiv:1608.03355.

[Sel15] Peter Selinger. Efficient Clifford+T Approximation of Single-qubit Operators. *Quantum Information & Computation*, 15(1-2):159–180, 2015. doi:10.26421/QIC15.1-2.

[SGT+18] 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 3rd ACM International Workshop on Real World Domain Specific Languages*, RWDSL '18, pages 7:1–7:10, 2018. doi:10.1145/3183895.3183901.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, 2016. doi:10.1145/2837614.2837655.

[Sho94] Peter W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 34th annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, 1994. doi:10.1109/SFCS.1994.365700.

[Sho95] Peter W. Shor. Scheme for Reducing Decoherence in Quantum Computer Memory. *Physical Review A*, 52:R2493–R2496, 1995. doi:10.1103/PhysRevA.52.R2493.

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

[SL83] Gadiel Seroussi and Abraham Lempel. Maximum Likelihood Decoding of Certain Reed-Muller Codes. *IEEE Transactions on Information Theory*, 29(3):448–450, 1983. doi:10.1109/TIT.1983.1056662.

[SM09] Vivek V. Shende and Igor L. Markov. On the CNOT-cost of TOFFOLI Gates. *Quantum Information & Computation*, 9(5):461–486, 2009. doi:10.26421/QIC9.5-6.

[SPMH02] Vivek V. Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes. Reversible Logic Circuit Synthesis. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '02, pages 353–360, 2002. doi:10.1145/774572.774625.

[SVM+17] Artur Scherer, Benoît Valiron, Siun-Chuon Mau, Scott Alexander, Eric van den Berg, and Thomas E. Chapuran. Concrete Resource Analysis of the Quantum Linear-system Algorithm used to Compute the Electromagnetic Scattering Cross Section of a 2D Target. *Quantum Information Processing*, 16(3):60, 2017. doi:10.1007/s11128-016-1495-5.

[Tho12] Michael K. Thomsen. A Functional Language for Describing Reversible Logic. In *Proceedings of the 2012 Forum on Specification and Design Languages*, FDL '12, pages 135–142, 2012. URL https://ieeexplore.ieee.org/document/6336999.

[Tof80] Tommaso Toffoli. Reversible Computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, ICALP '80, pages 632–644, 1980. doi:10.1007/3-540-10003-2_104.

[TS96] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Symposium on Static Analysis*, SAS '96, pages 349–365, 1996. doi:10.1007/3-540-61739-6_52.

[Vie95] Carlin Vieri. *Pendulum: A Reversible Computer Architecture*. Master's thesis, MIT Artificial Intelligence Laboratory, 1995. URL https://dspace.mit.edu/bitstream/handle/1721.1/36039/33342527-MIT.pdf.

[WGMAG14] Jonathan Welch, Daniel Greenbaum, Sarah Mostame, and Alan Aspuru-Guzik. Efficient Quantum Circuits for Diagonal Unitaries without Ancillas. *New Journal of Physics*, 16(3):033040, 2014. doi:10.1088/1367-2630/16/3/033040.

[WGMD09] Robert Wille, Daniel Grosse, David M. Miller, and Rolf Drechsler. Equivalence Checking of Reversible Circuits. In *Proceedings of the 39th International Symposium on Multiple-Valued Logic*, ISMVL '09, pages 324–330, 2009. doi:10.1109/ISMVL.2009.19.

[WOD10] Robert Wille, Sebastian Offermann, and Rolf Drechsler. SyReC: A Programming Language for Synthesis of Reversible Circuits. In *Proceedings of the 2010 Forum on Specification and Design Languages*, FDL '10, pages 1–6, 2010. doi:10.1049/ic.2010.0150.

[WZ82] William K. Wootters and Wojciech H. Zurek. A Single Quantum Cannot be Cloned. *Nature*, 299:802, 1982. doi:10.1038/299802a0.

[YG07] Tetsuo Yokoyama and Robert Glück. A Reversible Programming Language and its Invertible Self-interpreter. In *Proceedings of the 2007 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, pages 144–153, 2007. doi:10.1145/1244381.1244404.

[YM10] Shigeru Yamashita and Igor L. Markov. Fast Equivalence-checking for Quantum Circuits. *Quantum Information & Computation*, 10(9):721–734, 2010. doi:10.26421/QIC10.9-10.

[ZCC11] Bei Zeng, Andrew Cross, and Isaac L. Chuang. Transversality Versus Universality for Additive Quantum Codes. *IEEE Transactions on Information Theory*, 57(9):6272–6284, 2011. doi:10.1109/TIT.2011.2161917.

[ZPW18] Alwin Zulehner, Alexandru Paler, and Robert Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018. doi:10.1109/TCAD.2018.2846658.

[ZW18] Alwin Zulehner and Robert Wille. Advanced Simulation of Quantum Computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018. doi:10.1109/TCAD.2018.2834427.

Appendices

Appendix A

Correctness of Phase-folding

In this appendix we formally prove the correctness of the phase-folding algorithm. It will be convenient to consider $\mathcal{L}$-*parametrized* circuits $C : \mathbb{R}^{\mathcal{L}} \to \langle \mathcal{G} \rangle$. We call $\theta \in \mathbb{R}^{\mathcal{L}}$ a *parameter set* and denote the $\mathcal{L}$-parametrized circuit C applied to phase angles θ by $C(\theta)$. We extend $[\![\cdot]\!]_a$ to $\mathcal{L}$-parametrized circuits in the obvious way.

The intuition behind the phase-folding algorithm is that for any term $T \in \mathcal{T}$ of the phase polynomial, any two parameter sets θ, θ' which agree on the sum of all angles at locations in T – that is, $\sum_{(\ell,a)\in T} a \cdot \theta_\ell = \sum_{(\ell,a)\in T} a \cdot \theta'_\ell$ – give circuits $C(\theta)$ and $C(\theta')$ which are equivalent up to global phase. We say in this case that θ and θ' are $f^\sharp$-*equivalent*.

Definition A.1.1 (f-equivalence)**.** Two parameter sets $\theta, \theta' \in \mathbb{R}^{\mathcal{L}}$ are $f^\sharp$-equivalent for some $f^\sharp \in \mathcal{F}_n^\sharp$, denoted $\theta \sim_{f^\sharp} \theta'$, if or all $T \in \text{range}(f^\sharp)$,

$$\sum_{(\ell,a)\in T} a \cdot \theta_\ell = \sum_{(\ell,a)\in T} a \cdot \theta'_\ell$$

and for all other ℓ, $\theta_\ell = \theta'_\ell$. Here range$(f^\sharp)$ denotes $\{T \mid (r, T) \in f^\sharp\}$.

With the definition of $f^\sharp$-equivalence, we can now state the main theorem.

Theorem A.1.2. *Given a labelled circuit C and input state $A_{\mathbf{b}}$, let $(f^\sharp, A'_{\mathbf{b}'}) = [\![C]\!]_a(\emptyset, A_{\mathbf{b}})$. For any parameter sets $\theta, \theta' \in \mathbb{R}^{\mathcal{L}}$ such that $\theta \sim_{f^\sharp} \theta'$, and for any $\mathbf{x} \in \mathbb{F}_2^n$*

$$[\![C(\theta)]\!]|A\mathbf{x} + \mathbf{b}\rangle = [\![C(\theta')]\!]|A\mathbf{x} + \mathbf{b}\rangle$$

up to global phase.

Note that the circuit C before and after phase folding corresponds to $C(\theta)$ and $C(\theta')$ where $\theta \sim_{f^\sharp} \theta'$, hence Theorem A.1.2 implies the correctness of Algorithm 4.1.

We prove Theorem A.1.2 by showing that $[\![\cdot]\!]_a$ is a sound abstraction of a $\mathcal{L}$-parametrized *path integral semantics*, which itself is equivalent to the standard semantics. Specifically, we denote by $\mathcal{F}_n = \mathbb{R}^{\mathcal{L}} \to (\mathbb{F}_2^n \to \mathbb{R})$ the $\mathcal{L}$-parametrized phase polynomials, $\Phi_n : \mathbb{F}_2^n \to \mathbb{C}$ amplitude functions, and define

$$\mathcal{D}_n = \bigsqcup_{m \in \mathbb{N}} \mathcal{F}_n \times \Phi_n \times \mathrm{A}(\mathbb{F}_2^{n+m}, \mathbb{F}_2^n)$$

as the domain of $\mathcal{L}$-parametrized path integrals. Intuitively, the state given by an element $(f, \phi, A_{\mathbf{b}}) \in \mathcal{D}_n$ and parameter set θ is defined as

$$|f(\theta), \phi, A_{\mathbf{b}}\rangle = \sum_{\mathbf{x} \in \mathbb{F}_2^{n+m}} e^{if(\theta)(\mathbf{x})} \phi(\mathbf{x}) |A\mathbf{x} + \mathbf{b}\rangle.$$

The amplitude function ϕ is included to simplify the proof by separating the amplitude due to uninterpreted gates. The concrete semantics, $[\![C]\!]_c : \mathcal{D}_n \to \mathcal{D}_n$ is defined recursively as

$$\begin{aligned}
[\![X_i]\!]_c(f, \phi, A_{\mathbf{b}}) &= (f, \phi, A_{\mathbf{b} \oplus e_i}) \\
[\![\mathrm{CNOT}_{i,j}]\!]_c(f, \phi, A_{\mathbf{b}}) &= (f, \phi, E_{i,j} A_{\mathbf{b}}) \\
[\![R_Z(\cdot)_i^\ell]\!]_c(f, \phi, A_{\mathbf{b}}) &= (f'(\theta)(\mathbf{x}) = f(\theta)(\mathbf{x}) + \theta_\ell (A_i \mathbf{x} \oplus b_i), \phi, A_{\mathbf{b}}) \\
[\![U_{n-k+1,\ldots,n}]\!]_c(f, \phi, A_{\mathbf{b}}) &= (f, \phi'(\mathbf{x}) = \phi(\mathbf{x}) + \langle A_k \mathbf{x} + \mathbf{b}_k | U | A\mathbf{x} + \mathbf{b} \rangle, (A_k)_{\mathbf{b}_k}) \\
[\![C \fatsemi C']\!]_c &= [\![C']\!]_c \circ [\![C]\!]_c
\end{aligned}$$

The lemma below establishes the correctness of the path integral semantics with respect to the standard matrix semantics.

Lemma A.1.3. *For all $\mathcal{L}$-labelled circuits C and parameters θ, if $|f(\theta), \phi, A_{\mathbf{b}}\rangle = |\psi\rangle$, then*

$$([\![C]\!]_c(f, \phi, A_{\mathbf{b}}))(\theta) = [\![C(\theta)]\!]|\psi\rangle.$$

Proof. The base cases all follow from direct calculation:

$$\begin{aligned}
[\![X_i]\!]|f(\theta),\phi,A_{\mathbf{b}}\rangle &= \sum_{\mathbf{x}\in\mathbb{F}_2^{n+m}} e^{if(\theta)(\mathbf{x})}\phi(\mathbf{x})|A\mathbf{x}+(\mathbf{b}\oplus e_i)\rangle \\
[\![\mathrm{CNOT}_{i,j}]\!]|f(\theta),\phi,A_{\mathbf{b}}\rangle &= \sum_{\mathbf{x}\in\mathbb{F}_2^{n+m}} e^{if(\theta)(\mathbf{x})}\phi(\mathbf{x})|E_{i,j}A\mathbf{x}+E_{i,j}\mathbf{b}\rangle \\
[\![R_Z(\theta_\ell)_i]\!]|f(\theta),\phi,A_{\mathbf{b}}\rangle &= \sum_{\mathbf{x}\in\mathbb{F}_2^{n+m}} e^{i(f(\theta)(\mathbf{x})+\theta_\ell(A_i\mathbf{x}\oplus b_i))}\phi(\mathbf{x})|A\mathbf{x}+\mathbf{b}\rangle \\
[\![U_{n-k+1,\ldots,n}]\!]|f(\theta),\phi,A_{\mathbf{b}}\rangle &= \sum_{\mathbf{x}\in\mathbb{F}_2^{n+m}} e^{if(\theta)(\mathbf{x})}\phi(\mathbf{x}) \sum_{\mathbf{y}\in\mathbb{F}_2^k} (\langle A_k(\mathbf{x},\mathbf{y})+\mathbf{b}_k|U|A\mathbf{x}+\mathbf{b}\rangle)|A_k(\mathbf{x},\mathbf{y})+\mathbf{b}_k\rangle \\
&= \sum_{\mathbf{x}\in\mathbb{F}_2^{n+m+k}} e^{if(\theta)(\mathbf{x})}\left(\phi(\mathbf{x})+\langle A_k(\mathbf{x},\mathbf{y})+\mathbf{b}_k|U|A\mathbf{x}+\mathbf{b}\rangle\right)|A_k(\mathbf{x},\mathbf{y})+\mathbf{b}_k\rangle
\end{aligned}$$

Hence by induction on C, $([\![C]\!]_c(f,\phi,A_{\mathbf{b}}))(\theta) = [\![C(\theta)]\!]|\psi\rangle$. □

As is customary [CC77], we define a *soundness relation* $\tau \subseteq \mathcal{A}_n \times \mathcal{D}_n$ between states of the phase analysis and $\mathcal{L}$-parametrized path integrals.

Definition A.1.4 (soundness relation)**.** For $P^\sharp = (f^\sharp, A^\sharp_{\mathbf{b}^\sharp}) \in \mathcal{A}_n$ and $P = (f,\phi,A_{\mathbf{b}}) \in \mathcal{D}_n$, $(P^\sharp, P) \in \tau$ if and only if $\mathbf{b}^\sharp = \mathbf{b}$, and for all $\mathbf{x} \in \mathbb{F}_2^{n+m}$, there exists $\mathbf{x}^\sharp \in \mathbb{F}_2^n$ such that

$$A^\sharp\mathbf{x}^\sharp = A\mathbf{x}$$

and

$$f(\theta)(\mathbf{x}) = \sum_{(T,r)\in f^\sharp} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell,a)\in T} a\cdot\theta_\ell$$

up to global phase, where $\chi_{\mathbf{y}}(\mathbf{x}) = \chi_{\mathbf{y}^\sharp}(\mathbf{x}^\sharp)$ whenever $r = \mathbf{y}^\sharp \in \mathbb{F}_2^n$ and $A^\sharp\mathbf{x}^\sharp = A\mathbf{x}$.

The intuition behind the soundness relation above is that $A^\sharp_{\mathbf{b}^\sharp}$ overapproximates the basis states with non-zero amplitude in $|f(\theta),\phi,A_{\mathbf{b}}\rangle$ – i.e. for every $\mathbf{y} = A_{\mathbf{x}} + \mathbf{b}$ there exists $\mathbf{x}^\sharp$ such that $A^\sharp\mathbf{x}^\sharp + \mathbf{b}^\sharp = \mathbf{y}$. Moreover, the *abstract* phase polynomial $f^\sharp$ has the same (non-global) coefficients as f – $\sum_{(\ell,a)\in T} a\cdot\theta_\ell$ where $(T,r) \in f^\sharp$ – and in particular when $r \neq \bot$, the parities match.

From this definition it can be easily verified that if $(P^\sharp, P) \in \tau$, then for all θ,θ' such that $\theta \sim_{f^\sharp} \theta'$, we have $f(\theta) = f(\theta')$ up to constant factors. Moreover, this shows that our soundness relation is in a sense *strong enough* to prove Theorem A.1.2, in particular with the following lemma:

Lemma A.1.5. *If $(P^\sharp, P) \in \tau$, then for any $\theta, \theta' \in \mathbb{R}^{\mathcal{L}}$ such that $\theta \sim_{f^\sharp} \theta'$,*

$$|f(\theta), \phi, A_{\mathbf{b}}\rangle = |f(\theta'), \phi, A_{\mathbf{b}}\rangle$$

up to global phase.

Proof. Trivial since $f(\theta) = f(\theta')$ up to constant factors. □

By Lemma A.1.5 above, the last piece we need is to show that $[\![\cdot]\!]_a$ preserves soundness.

Lemma A.1.6. *Let C be a $\mathcal{L}$-parametrized circuit, and let $(P^\sharp, P) \in \tau$. Then*

$$([\![C]\!]_a P^\sharp, [\![C]\!]_c P) \in \tau.$$

Proof. Our proof follows by induction on the structure of C.

Case: $C = X_i$

We have

$$[\![C]\!]_a(f^\sharp, A^\sharp_{\mathbf{b}^\sharp}) = (f^\sharp, A^\sharp_{\mathbf{b}^\sharp \oplus e_i}), \qquad [\![C]\!]_c(f, \phi, A_{\mathbf{b}}) = (f, \phi, A_{\mathbf{b} \oplus e_i})$$

Since $\mathbf{b}^\sharp = \mathbf{b}$, $\mathbf{b}^\sharp \oplus e_i = \mathbf{b} \oplus e_i$. Moreover, since $A^\sharp$, A, are unchanged for all $\mathbf{x} \in \mathbb{F}_2^{n+m}$ there exists $\mathbf{x}^\sharp \in \mathbb{F}_2^n$ such that $A^\sharp \mathbf{x}^\sharp = A\mathbf{x}$, and likewise

$$f(\theta)(\mathbf{x}) = \sum_{(T,r) \in f^\sharp} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell,a) \in T} a \cdot \theta_\ell$$

where $\chi_{\mathbf{y}}(\mathbf{x}) = \chi_{\mathbf{y}^\sharp}(\mathbf{x}^\sharp)$ whenever $r = \mathbf{y}^\sharp \in \mathbb{F}_2^n$.

Case: $C = \mathrm{CNOT}_{i,j}$

Similar to the previous case.

Case: $C = R_Z(\cdot)_i^\ell$

Let $[\![C]\!]_a(f^\sharp, A^\sharp_{\mathbf{b}^\sharp}) = ((f^\sharp)', A^\sharp_{\mathbf{b}^\sharp})$, $[\![C]\!]_c(f, \phi, A_{\mathbf{b}}) = (f', \phi, A_{\mathbf{b}})$. As the basis states are unchanged, we only need to check that

$$f'(\theta)(\mathbf{x}) = \sum_{(T,r) \in (f^\sharp)'} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell',a) \in T} a \cdot \theta_{\ell'}$$

where $\chi_{\mathbf{y}}(\mathbf{x}) = \chi_{\mathbf{y}^\sharp}(\mathbf{x}^\sharp)$ whenever $r = \mathbf{y}^\sharp \in \mathbb{F}_2^n$.

By the definition of $[\![\cdot]\!]_c$ we have

$$\begin{aligned} f'(\theta)(\mathbf{x}) &= \theta_\ell(A_i\mathbf{x} \oplus \mathbf{b}_i) + f(\theta)(\mathbf{x}) \\ &= \theta_\ell b_i + (-1)_i^b \cdot \theta_\ell \chi_{A_i^T}(\mathbf{x}) + \sum_{(T,r)\in f^\sharp} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell',a)\in T} a \cdot \theta_{\ell'} \\ &= (-1)_i^b \cdot \theta_\ell \chi_{A_i^T}(\mathbf{x}) + \sum_{(T,r)\in f^\sharp} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell',a)\in T} a \cdot \theta_{\ell'} \end{aligned}$$

up to global phase. Since the basis states are unchanged, we only need to verify that whenever $A^\sharp \mathbf{x}^\sharp = A\mathbf{x}$, then $\chi_{A_i^T}(\mathbf{x}) = \chi_{(A^\sharp)_i^T}(\mathbf{x}^\sharp)$. In particular, we know

$$\chi_{(A^\sharp)_i^T}(\mathbf{x}^\sharp) = A_i^\sharp \mathbf{x}^\sharp = A_i\mathbf{x} = \chi_{A_i^T}(\mathbf{x}).$$

Case: $C = U_{n-k+1,\ldots,n}$

First note that $(\mathbf{b}^\sharp)' = \mathbf{b}_k^\sharp = \mathbf{b}_k = \mathbf{b}'$, as required.

To show that for all $\mathbf{x} \in \mathbb{F}_2^{n+m+k}$, there exists $\mathbf{x}^\sharp \in \mathbb{F}_2^n$ such that $A_k^\sharp (A_k^\sharp)^g \mathbf{x}^\sharp = A_k\mathbf{x}$, first note that there exists $\mathbf{y}$ such that

$$A_k^\sharp \mathbf{y} = A_k\mathbf{x}$$

By properties of the generalized inverse, $A_k^\sharp (A_k^\sharp)^g A_k^\sharp = A_k^\sharp$, hence $\mathbf{x}^\sharp = A_k^\sharp \mathbf{y}$ gives

$$A_k^\sharp (A_k^\sharp)^g \mathbf{x}^\sharp = A_k^\sharp (A_k^\sharp)^g A_k^\sharp \mathbf{y} = A_k^\sharp \mathbf{y} = A_k\mathbf{x} = A_k\mathbf{x}.$$

Finally, observe that $f' = f$ and $(f^\sharp)'$ has the same terms as $f^\sharp$, so

$$\begin{aligned} f'(\theta)(\mathbf{x}) = f(\theta)(\mathbf{x}) &= \sum_{(T,r)\in f^\sharp} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell,a)\in T} a \cdot \theta_\ell \\ &= \sum_{(T,r')\in (f^\sharp)'} \chi_{\mathbf{y}}(\mathbf{x}) \sum_{(\ell,a)\in T} a \cdot \theta_\ell \end{aligned}$$

where $\chi_{\mathbf{y}}(\mathbf{x}) = \chi_{\mathbf{y}^\sharp}(\mathbf{x}^\sharp)$ whenever $r = \mathbf{y}^\sharp \in \mathbb{F}_2^n$ and $A^\sharp \mathbf{x}^\sharp = A\mathbf{x}$. We only need to show that whenever $r' = (\mathbf{y}^\sharp)' \in (f^\sharp)'$, $\chi_{\mathbf{y}}(\mathbf{x}) = \chi_{(\mathbf{y}^\sharp)'}(\mathbf{x}^\sharp)$ for all $\mathbf{x}, \mathbf{x}^\sharp$ where $A_k^\sharp (A_k^\sharp)^g \mathbf{x}^\sharp = A_k\mathbf{x}$.

Suppose $(T, r') \in (f^\sharp)'$ and $r = (\mathbf{y}^\sharp)'$. Then by the definition of $[\![\cdot]\!]_a$ we have

$$\begin{aligned}
\chi_{(\mathbf{y}^\sharp)'}(\mathbf{x}^\sharp) &= \chi_{(\mathbf{y}^\sharp,\mathbf{0})}((A_k^\sharp)^g \mathbf{x}^\sharp) \\
&= \chi_{(\mathbf{y}^\sharp,\mathbf{0})}((A_k^\sharp)^g A_k^\sharp (A_k^\sharp)^g \mathbf{x}^\sharp) \\
&= \chi_{(\mathbf{y}^\sharp,\mathbf{0})}((A_k^\sharp)^g A_k \mathbf{x}) \\
&= \chi_{((A_k^\sharp)^g A_k)^T (\mathbf{y}^\sharp,\mathbf{0})}(\mathbf{x}) \\
&= \chi_{((A^\sharp)^g A)^T \mathbf{y}^\sharp}(\mathbf{x}) \\
&= \chi_{\mathbf{y}}(\mathbf{x})
\end{aligned}$$

The second from last equality follows since $(\mathbf{y}^\sharp, \mathbf{0})$ is in the column space of $(A_k^\sharp)^T$ – i.e. is a linear combination of the first $n-k$ rows of $A_k^\sharp$. The last equality follows since $\chi_{((A^\sharp)^g A)^T \mathbf{y}^\sharp}(\mathbf{x}) = \chi_{\mathbf{y}^\sharp}((A^\sharp)^g A\mathbf{x})$ and $A^\sharp (A^\sharp)^g A\mathbf{x} = A\mathbf{x}$ since $A\mathbf{x}$ is in the column space of $A^\sharp$ (see, e.g., [CM09]).

Case: $C = C' \fatsemi C''$

By the inductive hypothesis.

□

Proof (Theorem A.1.2). Given an input state $A_{\mathbf{b}}$, let $(f^\sharp, A^\sharp_{\mathbf{b}^\sharp}) = [\![C]\!]_a(\emptyset, A_{\mathbf{b}})$ and $(f', \phi', A'_{\mathbf{b}'}) = [\![C]\!]_c(0, 0, A_{\mathbf{b}})$. Clearly $((\emptyset, A_{\mathbf{b}}), (0, 0, A_{\mathbf{b}})) \in \tau$, and hence by Lemma A.1.6, $((f^\sharp, A^\sharp_{\mathbf{b}^\sharp}), (f', \phi', A'_{\mathbf{b}'}) \in \tau$.

By Lemmas A.1.3 and A.1.5 we have for any $\mathbf{x} \in \mathbb{F}_2^n$,

$$[\![C(\theta)]\!]|A\mathbf{x} + \mathbf{b}\rangle = |f'(\theta), \phi', A'_{\mathbf{b}'}\rangle = |f'(\theta'), \phi', A'_{\mathbf{b}'}\rangle = [\![C(\theta')]\!]|A\mathbf{x} + \mathbf{b}\rangle$$

up to global phase, as required. □