Doctoral Thesis

ETH Zurich 2023

High-Level Quantum Programming

Benjamin Bichsel

Diss. ETH

No. 29548

The icon on the cover page was designed by Simeon Zhelev.

DISS. ETH NO. 29548

HIGH-LEVEL QUANTUM PROGRAMMING

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

BENJAMIN BICHSEL
MSc ETH CS, ETH Zurich

born on 14 December 1992

accepted on the recommendation of

Prof. Dr. Martin Vechev (ETH Zurich)
Prof. Dr. Michael W. Hicks (University of Maryland)
Prof. Dr. Jens Palsberg (University of California, Los Angeles)

2023

ABSTRACT

Quantum computation has made remarkable progress in recent years, not only by constructing quantum computers that surpass classical counterparts in specific tasks, but also by developing increasingly complex quantum algorithms. Unfortunately, the development of essential tools for quantum programming, such as high-level programming languages and debugging tools, has not kept pace with these advancements. Therefore, the goal of this thesis is to advance quantum computation by introducing novel tools that both enable experts to fully leverage the potential of quantum computing and lower the entrance barrier for non-expert quantum programmers.

To this end, the thesis presents three innovative tools: Silq, Unqomp, and Abstraqt. Silq is a high-level quantum programming language whose most significant contribution is its ability to ensure all temporary quantum values can be automatically uncomputed, simplifying the programming process. Unqomp is the first procedure to automatically synthesize uncomputation within quantum circuits containing non-classical gates, such as the Hadamard gate. Abstraqt is a novel approach to efficiently simulate arbitrary quantum circuits at the cost of lost precision, enabling the study of circuit properties that were previously intractable.

These tools were inspired by established techniques from the programming languages community, which can serve as a rich reservoir of concepts and approaches beneficial for quantum computing. Silq utilizes a novel type system to enable uncomputation, Unqomp synthesizes uncomputation through a graph representation of quantum circuits, and Abstraqt leverages abstract interpretation to abstract the imprecision it introduces.

Overall, these tools improve the productivity of quantum programmers and reduce the likelihood of errors in quantum algorithm implementations.

ZUSAMMENFASSUNG

Die Quantenberechnung hat in den letzten Jahren signifikante Fortschritte erreicht, nicht nur durch den Bau von Quantencomputern, die klassischen Gegenstücken bei bestimmten Aufgaben überlegen sind, sondern auch durch die Entwicklung immer komplexerer Quantenalgorithmen. Leider aber konnte die Entwicklung essenzieller Werkzeuge für die Quantenprogrammierung wie höheren Programmiersprachen oder Debugging-Tools mit diesen Fortschritten nicht Schritt halten. Folglich ist das es Ziel dieser Arbeit, die Quantenberechnung voranzutreiben indem sie neue Werkzeuge einführt, die es sowohl Experten ermöglichen, das Potenzial der Quantenberechnung voll auszuschöpfen, als auch den Einstieg für unerfahrene Quantenprogrammierer erleichtern.

Diese Arbeit stellt drei innovative Werkzeuge vor: Silq, Unqomp und Abstraqt. Silq ist eine höhere Quantenprogrammiersprache, deren wichtigster Beitrag ihre Fähigkeit ist, sicherzustellen, dass alle temporären Quantenwerte automatisch "uncomputed" werden können, was die Programmierung vereinfacht. Unqomp ist das erste Verfahren zur automatischen Synthese von "Uncomputation" in Quantenschaltungen, die nichtklassische Gatter wie das Hadamard-Gatter enthalten. Abstraqt ist ein neuartiger Ansatz zur effizienten Simulation beliebiger Quantenschaltungen zum Preis eines Verlustes an Genauigkeit, der die Untersuchung von Schaltungen ermöglicht, die zuvor berechnungstechnisch unerreichbar waren.

Diese Werkzeuge wurden von etablierten Techniken aus der Programmiersprachenforschung inspiriert und zeigen die Vorteile der Anwendung solcher Methodologien auf die Quantenberechnung. Silq nutzt ein neuartiges Typsystem, um "uncomputation" zu ermöglichen, Unqomp synthetisiert "uncomputation" durch eine Graphenrepräsentation der Schaltung, und Abstraqt nutzt abstrakte Interpretation, um die Ungenauigkeit, die es einführt, abzuschätzen.

Insgesamt verbessern diese Werkzeuge die Produktivität von Quantenprogrammierern und reduzieren die Wahrscheinlichkeit von Fehlern bei der Implementierung von Quantenalgorithmen.

PUBLICATIONS

This thesis is based on the following publications:

- **Benjamin Bichsel**, Maximilian Baader, Timon Gehr, and Martin Vechev. "Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020. [1]
- Anouk Paradis, **Benjamin Bichsel**, Samuel Steffen, and Martin Vechev. "Unqomp: Synthesizing Uncomputation in Quantum Circuits". In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021. [2]
- **Benjamin Bichsel**, Maximilian Baader, Anouk Paradis, and Martin Vechev. "Abstraqt: Analysis of Quantum Circuits via Abstract Stabilizer Simulation". *(under submission)*. [3]

The following publications were part of my doctoral research but present results outside the scope of this thesis:

- **Benjamin Bichsel**, Timon Gehr, and Martin Vechev. "Fine-grained Semantics for Probabilistic Programs". In: *European Symposium on Programming (ESOP)*. Springer, 2018. [4]
- Marco Cusumano-Towner, **Benjamin Bichsel**, Timon Gehr, Martin Vechev, and Vikash K. Mansinghka. "Incremental Inference for Probabilistic Programs". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018. [5]
- **Benjamin Bichsel**, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. "DP-Finder: Finding Differential Privacy Violations by Sampling and Optimization". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018. [6]

- Victor Chibotaru, **Benjamin Bichsel**, Veselin Raychev, and Martin Vechev. "Scalable Taint Specification Inference with Big Code". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019. [7]

- Samuel Steffen, **Benjamin Bichsel**, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. "zkay: Specifying and Enforcing Data Privacy in Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. [8]

- Pesho Ivanov, **Benjamin Bichsel**, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin Vechev. "AStarix: Fast and Optimal Sequence-to-Graph Alignment". In: *International Conference on Research in Computational Molecular Biology (RECOMB)*. Springer, 2020. [9]

- **Benjamin Bichsel**, Samuel Steffen, Ilija Bogunovic, and Martin Vechev. "DP-Sniper: Black-Box Discovery of Differential Privacy Violations using Classifiers". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. [10]

- Pesho Ivanov, **Benjamin Bichsel**, and Martin Vechev. "Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds". In: *International Conference on Research in Computational Molecular Biology (RECOMB)*. Springer, 2022. [11]

- Samuel Steffen, **Benjamin Bichsel**, Roger Baumgartner, and Martin Vechev. "ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs". In: *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022. [12]

- Samuel Steffen, **Benjamin Bichsel**, and Martin Vechev. "Zapper: Smart Contracts with Data and Identity Privacy". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022. [13]

  ⋆ **Distinguished Paper Award**

- Anouk Paradis, **Benjamin Bichsel**, and Martin Vechev. "Reqomp: Space-constrained Uncomputation for Quantum Circuits". *(under submission)*. [14]

- Jasper Dekoninck*, Anouk Paradis*, **Benjamin Bichsel**, and Martin Vechev. “Synthetiq: Fast and Versatile Quantum Circuit Synthesis”. *(under submission)*.

  * equal contribution

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to all the people who have supported me throughout my doctoral studies.

First and foremost, I would like to thank my advisor, Prof. Dr. Martin Vechev, for his exceptional guidance and support. Your insights, expertise, and efforts in fostering a collaborative research environment within our group have been invaluable in shaping my career. Further, I extend my gratitude to my thesis committee members Prof. Dr. Michael W. Hicks and Prof. Dr. Jens Palsberg for serving on my committee.

I am immensely grateful to my many co-authors, Maximilian Baader, Roger Baumgartner, Ilija Bogunovic, Victor Chibotaru, Marco Cusumano-Towner, Jasper Dekoninck, Dana Drachsler-Cohen, Timon Gehr, Mario Gersbach, Pesho Ivanov, André Kahles, Vikash K. Mansinghka, Noa Melchior, Harun Mustafa, Anouk Paradis, Gunnar Rätsch, Veselin Raychev, Samuel Steffen, and Petar Tsankov. I have enjoyed collaborating with all of you—your contributions and inputs have shaped my research in significant ways. Most influential were Maximilian Baader who brought a much needed physicist's perspective into our quantum research and welcomed me as his roommate, Timon Gehr who created the best implementation of Silq anyone could ask for, Anouk Paradis who contributed excellent work on Chapters 4–5, and Samuel Steffen whose insightful recommendations on both research and personal matters I could always count on.

To my family Sonja Bichsel, Martin Bichsel, and Nina Bichsel, my girlfriend Noémi von Oppersdorff, and all my friends, I am incredibly grateful for your unwavering support, encouragement, and motivation throughout my studies. Thank you for always being there for me and believing in me.

I would also like to acknowledge the contribution of the artificial intelligence language models ChatGPT and GPT-4, which have helped me improve the phrasing of some parts of this thesis.

CONTENTS

1 INTRODUCTION

Quantum algorithms can outperform classical algorithms by exploiting the principles of quantum mechanics. To leverage this potential, researchers keep proposing increasingly complex quantum algorithms [15, 16, 17, 18, 19, 20], with the goal of running them on ever-improving quantum computers. At the same time, significant improvements in building quantum computers have already enabled them to outperform classical computers on certain tasks, albeit ones that are not yet practically useful [21]. Based on these encouraging developments, it seems increasingly likely that future quantum computers will be able to reliably run a wide range of quantum algorithms and achieve significant speed-ups over classical algorithms.

QUANTUM PROGRAMMING Unfortunately, these advances in building powerful quantum computers and inventing novel quantum algorithms are not matched by the development of tools crucial for quantum programming, such as high-level programming languages. As a consequence, programming quantum computers remains a fundamental challenge, even for specialists well-versed in the intricacies of quantum computation. This problem is exacerbated by the fact that debugging quantum programs is notoriously difficult, as quantum state cannot be read without side-effects and the behavior of quantum programs is inherently probabilistic and often unintuitive.

GOAL The overarching goal of this thesis is therefore to advance the field of quantum computation by introducing novel tools that both lower the entrance barrier of non-expert quantum programmers but also enable experts to harness the full potential of quantum computing.

Specifically, this thesis introduces three novel tools that boost the ability of developers to quickly write efficient and correct quantum algorithms:

- Silq (Chapter 3) is a high-level quantum language designed to abstract away low-level implementation details of quantum algorithms. Silq's most significant contribution is its ability to ensure all its temporary quantum values can be automatically *uncomputed*, a tedious task necessary when discarding values from a quantum program's state.

- Unqomp (Chapter 4) is the first procedure that automatically synthesizes uncomputation within quantum circuits containing non-classical gates like the *Hadamard* gate, a crucial step required for the open problem of compiling Silq.

- Abstraqt (Chapter 5) leverages a novel approach to efficiently simulate arbitrary quantum circuits at the cost of lost precision. This allows it to establish interesting properties of quantum circuits which would otherwise be intractable.

DISCUSSION Overall, this thesis helps to mitigate multiple fundamental challenges in the field of quantum computation, including lifting the level of abstraction when describing quantum algorithms (Chapter 3), compiling high-level specifications of quantum algorithms down to low-level circuits (Chapter 4), and efficiently analyzing quantum circuits on a classical computer (Chapter 5).

TECHNIQUES Our tools are inspired by techniques from the programming language community, which has extensive experience in automating complex processes and making them more accessible to a wider audience. Specifically, Silq establishes that uncomputation is possible using a novel *type system*, Unqomp *synthesizes* uncomputation by relying on a graph representation of the circuit, and Abstraqt leverages the mathematical framework of *abstract interpretation* to *(over-)abstract* the imprecision it introduces.

In the following, we provide a brief overview of each tool.

1.1 SILQ

Chapter 3 introduces Silq, a high-level quantum language designed to abstract away low-level implementation details of quantum algorithms.

As its key novelty, Silq helps to bridge the conceptual gap between classical and quantum languages by allowing quantum programs to safely drop temporary values from the program state without unintended side-effects. Generally, temporary values produced during quantum computations must be reset to zero before they can be safely discarded, in a process called *uncomputation*. Before Silq, discarding temporary values often required programmers to manually uncompute these values or to rely on unsafe constructs that could induce implicit measurements, which typically lead to an unintended collapse of the program state (for details, see §3.2).

By safely automating uncomputation, Silq naturally enables an intuitive semantics that implicitly drops temporary values, as in classical computation. To ensure this semantics is indeed physically realizable, Silq features a quantum type system which leverages novel annotations to reject unphysical programs.

Our experimental evaluation demonstrates that Silq programs are not only shorter, but also easier to read and write than analogous programs in existing quantum languages Q# [22] or Quipper [23], as Silq programs require fewer primitives and concepts.

IMPACT Silq has not only paved the way for more recent advancements in automatic uncomputation [2, 14, 24, 25], but is also a useful reference to demonstrate the suitability of a linear type system for quantum languages [26, 27, 28], and to show-case how quantum languages can elegantly mix classical and quantum computation [27, 29, 30]. We provide an overview of the significant impact Silq has had on quantum programming research in terms of language features of other topics like internal representations and verification in §3.9.

Silq is also being used to teach quantum programming, for example in the CS238 Quantum Programming course at the University of California [31].

Finally, Silq has received widespread attention even beyond academia (e.g., >500 stars on GitHub), prompted a book dedicated to learning Silq [32], and was covered by major news publishers[1].

1.2 UNQOMP

While Silq provides a much needed high-level language for quantum programming, it does not address the open problem of compiling Silq down to low-level circuits applying individual gates. The most challenging aspect of such a compilation is synthesizing efficient uncomputation, which we address by Unqomp, introduced in Chapter 4.

Perhaps surprisingly, Unqomp is not integrated with Silq but with Qiskit. This choice is intentional, as it allows us to bring the benefits of automatic uncomputation to the (to date) larger user base of Qiskit. Conveniently, our working integration with Qiskit also demonstrates that Unqomp can be readily integrated into existing quantum languages.

1 See `https://silq.ethz.ch/news`.

Our evaluation shows that compared to pure Qiskit, programs leveraging Unqomp are not only shorter (-19% on average), but also generate more efficient circuits (-71% gates and -19% qubits on average).

COMPILING SILQ Automatically synthesizing uncomputation, as achieved by Unqomp, is a key step towards compiling Silq programs. However, the task of compiling Silq also induces other challenges besides synthesizing uncomputation, such as handling its flexible interleaving of classical and quantum computation. Consequently, developing a full compiler for Silq is an ongoing research effort [33].

IMPACT Since we published Unqomp, several researchers have worked on the topic of automatic uncomputation or even built directly upon Unqomp [14, 25, 34], with one work directly integrating Unqomp [25]. We provide a detailed overview of the impact of Unqomp in §4.8.

In the future, we are expecting to see more usages of Unqomp, as it is simple to integrate into existing languages and solves a pressing problem when writing quantum programs.

1.3 ABSTRAQT

Concluding this thesis, Chapter 5 presents Abstraqt, a novel approach to efficiently simulate arbitrary quantum circuits at the cost of lost precision.

Debugging already written quantum programs is a fundamental challenge because quantum computers give very limited visibility into the program's internals, as quantum states cannot be read without side-effects. Further exacerbating the problem, existing quantum computers are highly error-prone, meaning that advanced debugging techniques to bypass this problem [35] are difficult to exploit in practice. Thus, there is significant interest in simulating quantum computations on classical computers, which allows for full visibility into the quantum state but is typically intractable.

A notable exception are *Clifford circuits*, an important class of quantum circuits which only apply a subset of all quantum gates and can therefore be efficiently simulated using *stabilizer simulation* [36]. While generalizations of stabilizer simulation to arbitrary circuits exist, they again suffer from an exponential runtime as they must process an exponential amount of information about the quantum state [36, §VII-C].

Abstraqt addresses this challenge by compressing the quantum state information into an efficient representation, at the cost of potentially lost precision. Specifically, Abstraqt relies on the mathematical framework of abstract interpretation to ensure that our compression of quantum states is sound.

Our evaluation demonstrates that Abstraqt can establish circuit properties intractable for all existing simulation techniques, comparing to state-of-the-art stabilizer simulators [37, 38], a state vector simulator [39], and a recent abstract simulator based on abstracting density matrices [40].

IMPACT As a very recent work still under submission, Abstraqt has not yet impacted quantum computing research.

However, we hope that Abstraqt will trigger more research into sacrificing precision for efficiency when simulating quantum circuits. In turn, we expect such efforts to benefit the understanding of high-level programs such as Silq programs, without the need to run an intractable simulation.

1.4 DISCUSSION

Overall, Chapters 3–5 introduce a diverse set of novel tools that help developers to write efficient and correct quantum algorithms.

The impact of these tools is significant, as they allow quantum algorithm developers to focus on the high-level logic of their algorithms, without being distracted by low-level implementation details. This greatly improves the productivity of quantum programmers and reduces the likelihood of errors in the implementation of quantum algorithms.

Moreover, our tools have the potential to impact the wider quantum computing community. Silq makes quantum programming accessible to a wider audience, including those without deep knowledge of quantum computation. Unqomp is easy to integrate with existing quantum languages, and Abstraqt efficiently simulates key aspects of quantum circuits.

In conclusion, our tools showcase the power of taking inspiration from established techniques in the programming language community, and the potential of applying these techniques to the field of quantum computing—an approach suitable not just for quantum languages (Chapter 3), circuit synthesis (Chapter 4), and circuit simulation (Chapter 5), but also for tasks like circuit optimization (e.g., [41, 42, 43]), verification (e.g., [44, 45, 46]), or testing (e.g., [35]). We are optimistic that these tools will pave the way for more efficient and correct quantum algorithms, and more generally will encourage the quantum computation community to keep taking inspiration from the programming language community.

2 BACKGROUND

Before diving into the contributions of this thesis, we present a unified background on quantum computation useful throughout this thesis. We refer readers unfamiliar with quantum computation to [47] for an excellent in-depth introduction.

BASIC NOTATION Generally, we write $\mathbb{B}$ for $\{0,1\}$, $\mathbb{Z}_4$ for $\{0,1,2,3\}$, 2^S for the power set of set S, and $\Re(c)$ for the real part of a complex number c.

2.1 QUANTUM STATES

We first introduce quantum states and how to represent them.

QUBIT The state of a quantum bit (qubit) is a superposition (linear combination) $\varphi = \gamma_0 |0\rangle + \gamma_1 |1\rangle$, where $\gamma_0, \gamma_1 \in \mathbb{C}$, and $\|\varphi\|^2 = \|\gamma\|^2 = \|\gamma_0\|^2 + \|\gamma_1\|^2$ denotes the probability of being in state φ. In particular, we allow $\|\varphi\| < 1$ to indicate that a measurement yields state φ with probability $\|\varphi\|$—a common convention [48, Convention 3.3].

Where convenient, if φ is the state of a variable x, we write its state as $\varphi = \gamma_0 |0\rangle_x + \gamma_1 |1\rangle_x$. Especially in Chapters 3–4, this helps emphasize which parts of the quantum state correspond to which variable.

TENSOR PRODUCT A system of multiple qubits can be described using the tensor product $\otimes$. For example, for two qubits $\varphi_0 = |0\rangle$ and $\varphi_1 = \frac{1}{\sqrt{2}} |0\rangle - \mathrm{i}\frac{1}{\sqrt{2}} |1\rangle$, the composite state is

$$\varphi_0 \otimes \varphi_1 = \frac{1}{\sqrt{2}} |0\rangle \otimes |0\rangle - \mathrm{i}\frac{1}{\sqrt{2}} |0\rangle \otimes |1\rangle = \frac{1}{\sqrt{2}} |0\rangle |0\rangle - \mathrm{i}\frac{1}{\sqrt{2}} |0\rangle |1\rangle .$$

Here, we first used the linearity of $\otimes$ in its first argument and then omitted $\otimes$ for convenience. Simplifying notation further, we may also write $|0\rangle |0\rangle$ as $|0,0\rangle$ or even $|00\rangle$.

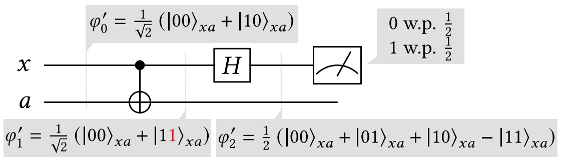

$\varphi_0' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |10\rangle_{xa})$
0 w.p. $\frac{1}{2}$
1 w.p. $\frac{1}{2}$
x
a
H
$\varphi_1' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |11\rangle_{xa})$
$\varphi_2' = \frac{1}{2}(|00\rangle_{xa} + |01\rangle_{xa} + |10\rangle_{xa} - |11\rangle_{xa})$

FIGURE 2.1: Example quantum circuit.

We can emphasize that the first qubit is stored in variable x and the second qubit is stored in variable y by writing

$$\varphi_0 \otimes \varphi_1 = \frac{1}{\sqrt{2}} |0\rangle_x \otimes |0\rangle_y - \mathrm{i}\frac{1}{\sqrt{2}} |0\rangle_x \otimes |1\rangle_y = \frac{1}{\sqrt{2}} |0\rangle_x |0\rangle_y - \mathrm{i}\frac{1}{\sqrt{2}} |0\rangle_x |1\rangle_y .$$

For readability, we often abbreviate $|a\rangle_x \otimes |b\rangle_y$ by $|a\rangle_x |b\rangle_y$ or $|ab\rangle_{xy}$.

ENTANGLEMENT A composite state is called *entangled* if it cannot be written as a tensor product of single qubit states, but needs to be written as a sum of tensor products. For example, the above composite state $\varphi_0 \otimes \varphi_1$ is unentangled, while $\Phi^+ = \frac{1}{\sqrt{2}} |0\rangle |0\rangle + \frac{1}{\sqrt{2}} |1\rangle |1\rangle$ is entangled.

MEASUREMENT To acquire information about a quantum state, we can (partially) measure it. Measurement has a probabilistic nature; if we measure $\varphi = \sum_{v \in \{0,1\}} \gamma_v |v\rangle$, we obtain the value $v' \in \{0,1\}$ with probability $\|\gamma_{v'}\|^2$. As a fundamental law of quantum mechanics, if we measure the value v', the state after the measurement is $\gamma_{v'} |v'\rangle$ (we do not normalize this state to preserve linearity). This is referred to as the *collapse* of φ to $\gamma_{v'} |v'\rangle$, since superposition is lost.

Importantly, measuring part of a state can affect the whole state. To illustrate the effect of measuring the first part $|v\rangle$ of $\sum_{v=0}^{1} \sum_{w=0}^{1} \gamma_{v,w} |v\rangle \otimes |w\rangle$, we first rewrite it to $\sum_{v=0}^{1} \gamma_v |v\rangle \otimes \tilde{\varphi}_v$, separating out the remainder $\tilde{\varphi}_v$ of the state, where $\|\tilde{\varphi}_v\| = 1$. This is a common technique and always possible for appropriate choices of γ_v and $\tilde{\varphi}_v$, even for systems where $\tilde{\varphi}_v$ consists of multiple qubits. Then, measuring the first part to be v' yields state $\gamma_{v'} |v'\rangle \otimes \tilde{\varphi}_{v'}$, also collapsing the remainder of the state.

DENSITY MATRICES We note that in Chapter 5, we will work with an alternative representation of quantum states in terms of *density matrices*, as introduced in §5.2.

2.2 QUANTUM CIRCUITS

A quantum circuit is the most widely used model of quantum computation in which quantum computers apply a sequence of gates to manipulate quantum states. Fig. 2.1 shows an example of a simple quantum circuit.

WIRES Quantum circuits represent each qubit as a wire, depicted as horizontal lines named x and a in Fig. 2.1. For example, initializing x to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and a to $|0\rangle$ yields the initial state φ_0' in Fig. 2.1.

GATES Circuits manipulate qubits using linear unitary operators[1] called *gates*, which may span multiple wires. For example, the first gate in Fig. 2.1 is the controlled NOT gate, called CX. It flips the second qubit ($\oplus$) if the first qubit ($\bullet$) is 1. More formally: $CX_{xy}\,|ab\rangle_{xy} = |a\rangle_x \otimes |a \oplus b\rangle_y$, where $\oplus$ is the XOR operation. Like every gate, CX is linear and this definition hence extends naturally to arbitrary superpositions. For example, in Fig. 2.1, $CX_{xa}\,(\varphi_0') = \varphi_1'$. The second gate applied in Fig. 2.1 is the Hadamard transform H, which maps $|0\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|1\rangle$ to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$. The second qubit a is not modified by H_x. For example, $H_x\left(\varphi_1'\right) = \varphi_2'$. Finally, Fig. 2.1 applies a measurement to the qubit x, shown as .

Mathematically, an n-qubit quantum gate $U \in \mathcal{U}(2^n)$ is a unitary operation which evolves state vector $\varphi \in \mathbb{C}^{2^n}$ to $U\varphi \in \mathbb{C}^{2^n}$. Here, we write $\mathcal{U}(2^n)$ for the set of unitary $2^n \times 2^n$ matrices. Padding a smaller k-qubit quantum gate $U \in \mathcal{U}(2^k)$ to act on an n-qubit state without modifying neither the first i nor last $n-i-k$ qubits yields $U_{(i)} := \mathbb{I}_{2^i} \otimes U \otimes \mathbb{I}_{2^{n-i-k}}$, where $\otimes$ is the tensor product. Here, $\mathbb{I}_l$ is the $l \times l$ identity matrix.

CONTROLS AND TARGETS The qubits involved in a gate can generally be divided into two groups. First, the *controls* (depicted as $\bullet$ in circuits) are preserved by the gate (see also [47, §4.3]). The other qubits, which may be modified by the gate depending on the controls, are called *targets*.

Formally, a gate U controlled by one[2] qubit x maps state

$$\sum_{a\in\{0,1\}} \gamma_a\,|a\rangle_x \otimes \varphi_a \quad \text{to} \quad \sum_{a\in\{0,1\}} \gamma_a\,|a\rangle_x \otimes \varphi_a'\,,$$

1 As we express linear operators as matrices U, U is unitary if its conjugate transpose is also its inverse: $U^\dagger U = UU\dagger = I$.

2 The definition generalizes naturally to multiple controls.

preserving the coefficients γ_a of control x, and only modifying the remainder of the state φ_a to φ'_a, where the mapping $\varphi_a \mapsto \varphi'_a$ may depend on a.

NO-CLONING The no-cloning theorem is a fundamental result in quantum mechanics that states that cloning an arbitrary quantum state is unphysical: we cannot achieve the operation $\varphi \mapsto \varphi \otimes \varphi$. We note that Silq's type system prevents cloning.

2.3 QUANTUM RANDOM ACCESS MACHINES

While Chapters 4–5 are phrased in terms of quantum circuits, Chapter 3 assumes a more powerful computational model for quantum computers, namely a quantum random access machine (QRAM) [49]. In contrast to the quantum circuit model, the QRAM model more explicitly incorporates classical computation. Specifically, a QRAM consists of a classical computer enhanced with quantum storage that supports state preparation, certain unitary gates, and measurement operations.

HILBERT SPACE, GROUND SET, BASIS STATE In Chapter 3, we further generalize quantum states beyond qubits to allow for superpositions over arbitrary sets. Consider a classical variable that can take on values from a finite *ground set* S. Then, the quantum states induced by S form the Hilbert space $\mathcal{H}(S)$ consisting of the formal complex linear combinations [50, p. 379] over S:

$$\mathcal{H}(S) := \left\{ \sum_{v \in S} \gamma_v |v\rangle \;\middle|\; \gamma_v \in \mathbb{C} \right\}.$$

Here, each element $v \in S$ corresponds to a *(computational) basis state* $|v\rangle$. For $S = \{0,1\}$, we obtain the Hilbert space of a single qubit $\mathcal{H}(\{0,1\}) = \{\gamma_0 |0\rangle + \gamma_1 |1\rangle \mid \gamma_0, \gamma_1 \in \mathbb{C}\}$, with computation basis states $|0\rangle$ and $|1\rangle$.

We note that we use the (standard) inner product $\langle \cdot | | \cdot \rangle$ throughout this work, defined by

$$\langle \sum_{v \in S} \gamma_v |v\rangle \,|\,|\, \sum_{v \in S} \gamma'_v |v\rangle \rangle = \sum_{v \in S} \gamma^*_v \gamma'_v.$$

MEASUREMENT Analogously to §2.1, measuring the first part $|v\rangle$ of quantum state $\sum_{v \in S} \sum_{w \in S'} \gamma_v |v\rangle \otimes \tilde{\varphi}_v$ yields state $\gamma_{v'} |v'\rangle \otimes \tilde{\varphi}_{v'}$ with probability $\|\gamma_{v'}\|$.

LINEAR ISOMETRIES Besides measurements, Chapter 3 further assumes that we can manipulate quantum states using *linear isometries*, i.e., linear functions $f \colon \mathcal{H}(S) \to \mathcal{H}(S')$ preserving inner products: for all $\varphi, \varphi' \in \mathcal{H}(S)$, $\langle f(\varphi)| \, |f(\varphi')\rangle = \langle \varphi| \, |\varphi'\rangle$. Linear isometries generalize the commonly used notion of unitary operations, which additionally require that vector spaces $\mathcal{H}(S)$ and $\mathcal{H}(S')$ have the same dimension. As this prevents dynamically allocating and deallocating qubits [3], we use the more general notion of linear isometries in Chapter 3.

We note that QRAMs can be extended to support linear isometries, by (i) padding input and output space to have the same dimension (using state preparation) and (ii) approximating the resulting unitary operation arbitrarily well using a standard set of universal quantum gates [47, §4.5.3].

3 Since given input space $\mathcal{H}(S)$, allocating qubits leads to a larger output space $\mathcal{H}(S')$, requiring $\mathcal{H}(S)$ and $\mathcal{H}(S')$ to have the same dimension (as enforced by unitary operations) prevents dynamically allocating qubits.

3 SILQ: A HIGH-LEVEL QUANTUM LANGUAGE WITH SAFE UNCOMPUTATION AND INTUITIVE SEMANTICS

In this chapter, we present Silq, a high-level quantum language designed to abstract away low-level implementation details, most importantly by supporting safe, automatic uncomputation.

Before Silq, quantum languages forced the programmer to work at a low level of abstraction, leading to unintuitive and cluttered code. A fundamental reason is that dropping temporary values from the program state requires explicitly applying quantum operations that safely *uncompute* these values. Silq addresses this challenge by supporting safe, automatic uncomputation. This enables an intuitive semantics that implicitly drops temporary values, as in classical computation. To ensure physicality of Silq's semantics, its type system leverages novel annotations to reject unphysical programs.

Our experimental evaluation demonstrates that Silq programs are not only easier to read and write, but also significantly shorter than equivalent programs in other quantum languages (on average -46% for Q#, -38% for Quipper), while using only half the number of quantum primitives.

3.1 INTRODUCTION

As discussed in Chapter 1, researchers have been proposing increasingly complex quantum algorithms [15, 16, 17, 18, 19, 20], driving the need for expressive, high-level quantum languages.

THE NEED FOR UNCOMPUTATION Analogously to the classical setting, quantum computations often produce temporary values. However, as a key challenge specific to quantum computation, removing such values from consideration induces an *implicit measurement* collapsing the state [47, §4.4], see also Fig. 4.2. In turn, collapsing can result in unintended side-effects on the state due to the phenomenon of *entanglement*. Surprisingly, due to the quantum principle of *deferred measurement* [47, §4.4], preserving values until computation ends is equivalent to measuring them immediately after their last use, and hence cannot prevent this problem.

To remove temporary values from consideration without inducing an implicit measurement, algorithms in existing languages must explicitly *uncompute* all temporary values, i.e., modify their state to enable ignoring them without side-effects. This results in a significant gap from quantum to classical languages, where discarding temporary values typically requires no action (except for heap values not garbage-collected). This gap is a major roadblock preventing the adoption of quantum languages, especially since the implicit side-effects resulting from uncomputation mistakes such as silently dropping temporary values are highly unintuitive.

SILQ This chapter presents Silq, a high-level quantum language bridging this gap by ensuring temporary values can be uncomputed automatically. To this end, Silq's type system exploits a fundamental pattern in quantum algorithms, stating that uncomputation can be done safely if (i) the original evaluation of the uncomputed value can be described classically, and (ii) the variables used to evaluate it are preserved and can thus be leveraged for uncomputation.

As uncomputation happens behind the scenes and is always safe, Silq is the first quantum language to provide *intuitive semantics*: if a program type-checks, its semantics follows an intuitive recipe that simply drops temporary values. Importantly, Silq's semantics is *physical*, i.e., can be realized on a quantum random access machine (QRAM, see §2.3).

Overall, Silq allows expressing quantum algorithms more safely and concisely than existing quantum programming languages, while typically using only half the number of quantum primitives. In our evaluation (§3.7), we show that across 28 challenges from recent coding contests [51, 52], Silq programs require on average 46% fewer lines of code than Q# [22]. Similarly, expressing the triangle finding algorithm [53] in Silq requires 38% less code than Quipper [23].

```
d := a || b || c;
```
Silq

```
with_computed (OR a b) $
  \t -> OR t c
```
Quipper

```
1 using(t=Qubit()){
2   OR(a,b,t);
3   OR(t,c,d);
4   Adjoint OR(a,b,t);
5 }
```
Q#

FIGURE 3.1: Benefit of Silq's automatic uncomputation.

MAIN CONTRIBUTIONS The main contributions of this chapter are:

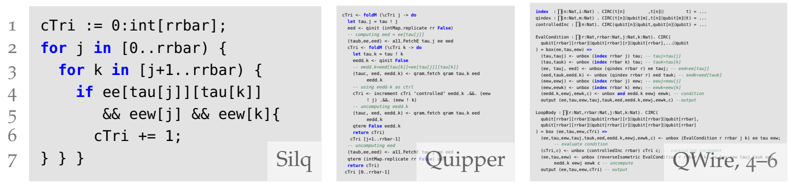
- Silq[1], a high-level quantum language enabling safe, automatic uncomputation (§3.3).
- A full formalization of Silq's key language fragment Silq-Core (§3.4), including its type system (§3.5) and semantics (§3.6), whose physicality relies on its type system.
- An implemented type-checker[2], proof-of-concept simulator[2], and development environment[3] for Silq.
- An evaluation, showing that Silq code is more concise and readable than code in existing languages (§3.7).

3.2 BENEFIT OF AUTOMATIC UNCOMPUTATION

Next, we show the benefit of automatic uncomputation compared to explicit uncomputation in languages that preceded Silq, including Q# [22], Quipper [23], and QWire [54].

EXPLICIT UNCOMPUTATION Fig. 3.1 shows code snippets which compute the `OR` of three qubits. This is easily expressed in Silq (top left), which leverages automatic uncomputation for `a||b`. In contrast, Q# (right) requires (i) allocating a new qubit `t` initialized to `0` in Line 1, (ii) using `OR`[4] to store the result of `a||b` in `t` in Line 2, (iii) using `OR` to store the result of `t||c` in the pre-allocated qubit `d` in Line 3, and (iv) uncomputing `t` by reversing the operation from Line 2 in Line 4. Here, `Adjoint OR` is the inverse of `OR` and thus resets `t` to its original value of `0`. Hence, the *implicit measurement*

1 http://silq.ethz.ch/
2 https://github.com/eth-sri/silq/tree/pldi2020
3 https://marketplace.visualstudio.com/items?itemName=eth-sri.vscode-silq
4 Since Q# does not support `OR` natively, we would need to implement it too.



```
1  cTri := 0:int[rrbar];
2  for j in [0..rrbar) {
3    for k in [j+1..rrbar) {
4      if ee[tau[j]][tau[k]]
5         && eew[j] && eew[k]{
6        cTri += 1;
7  } } }
```

```
cTri <- foldM (\cTri j -> do
  let tau_j = tau ! j
  eed <- qinit (intMap_replicate rr False)
  -- computing eed = ee[tau[j]]
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  cTri <- foldM (\cTri k -> do
    let tau_k = tau ! k
    eedd_k <- qinit False
    -- eedd_k=eed[tau[k]]=ee[tau[j]][tau[k]]
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed
         eedd_k
    -- using eedd_k as ctrl
    cTri <- increment cTri ‘controlled‘ eedd_k .&&. (eew
         ! j) .&&. (eew ! k)
    -- uncomputing eedd_k
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed
         eedd_k
    qterm False eedd_k
    return cTri)
   cTri [j+1..rrbar-1]
  -- uncomputing eed
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  qterm (intMap_replicate rr False) eed
  return cTri)
 cTri [0..rrbar-1]
```

```
index  : ∏(n:Nat,i:Nat) . CIRC(t[n]         ,t[n]⊗         t) = ...
qindex : ∏(n:Nat,m:Nat) . CIRC(t[n]⊗qubit[m],t[n]⊗qubit[m]⊗t) = ...
controlledInc : ∏(n:Nat). CIRC(qubit[n]⊗qubit,qubit[n]⊗qubit) = ...

EvalCondition : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar],...⊗qubit
) = box(ee,tau,eew) =>
  (tau,tauj) <- unbox (index rrbar j) tau; -- tauj=tau[j]
  (tau,tauk) <- unbox (index rrbar k) tau; -- tauk=tau[k]
  (ee, tauj, eed) <- unbox (qindex rrbar r) ee tauj; -- eed=ee[tauj]
  (eed,tauk,eedd_k) <- unbox (qindex rrbar r) eed tauk; -- eedk=eed[tauk]
  (eew,eewj) <- unbox (index rrbar j) eew; -- eewj=eew[j]
  (eew,eewk) <- unbox (index rrbar k) eew; -- eewk=eew[k]
  (eedd_k,eewj,eewk,c) <- unbox and eedd_k eewj eewk; -- condition
  output (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) --output

LoopBody : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar],
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar]
) = box (ee,tau,eew,cTri) =>
  (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) <- unbox (EvalCondition r rrbar j k) ee tau eew;
      -- evaluate condition
  (cTri,c) <- unbox (controlledInc rrbar) cTri c; -- controlled increment
  (ee,tau,eew) <- unbox (reverseIsometric EvalCondition r rrbar j k) ee tau eew tauj tauk eed
      eedd_k eewj eewk c -- uncompute
  output (ee,tau,eew,cTri) -- output
```

FIGURE 3.2: Comparing Silq to Quipper and QWire code, more readable version in App. A.1.

induced by removing t from consideration in Line 5 always measures the value 0, which has no side-effects (see Chapter 2). We note that we cannot allocate t within OR, as Q# enforces that qubits must be deallocated in the function that allocates them. [5] Overall, handling uncomputation in Q# is more tedious but also more error-prone, as erroneous uncomputation can trigger implicit measurements.

Explicit uncomputation is even more tedious in Fig. 3.2, which shows part of a triangle finding algorithm originally encoded[6] by the authors of Quipper [23] (middle). The condition is easily expressed in Silq (left, Lines 4–5) using nested expressions. In contrast, the equivalent Quipper code is obfuscated by uncomputation of sub-expressions.

CONVENIENCE FUNCTIONS As uncomputation is a common task, various quantum languages try to reduce its boiler-plate code by introducing convenience functions such as ApplyWith in Q#. Fig. 3.1 shows a Quipper implementation using a similar function with_computed, which (i) evaluates a||b in Line 1, (ii) uses the result t to compute t||c in Line 2, and (iii) implicitly uncomputes t. However, this still requires explicitly triggering uncomputation using with_computed and introducing a name t for the result of a||b. In particular, this does not enable a natural nesting of expressions, as the sub-expression OR a b needs to be managed by with_computed. Moreover, with_computed cannot ensure safety: we can make the uncomputation unsafe by flipping the bit stored in b between Line 1 and Line 2, triggering an implicit measurement.

5 The Quantum memory management of Q# is explained on https://learn.microsoft.com/en-us/azure/quantum/user-guide/language/statements/quantummemorymanagement#quantum-memory-management, accessed on May 2, 2023.

6 Taken from: https://www.mathstat.dal.ca/~selinger/quipper/doc/src/Quipper/Algorithms/TF/QWTFP.html#line-494

NON-LINEAR TYPE SYSTEMS Most quantum languages cannot ensure that all temporary values are safely uncomputed for a fundamental reason: they support reference sharing in a non-linear type system and hence cannot statically detect when values are removed from consideration (which happens when the last reference to the value goes out of scope). Besides Quipper, which we discuss in more detail, there are many other works of this flavor, including LIQuiD [55], ProjectQ [56], Cirq [57], and QisKit [39].

LINEAR TYPE SYSTEMS Other languages, like QPL [48] and QWire, introduce a linear type system to prevent accidentally removing values from consideration, which corresponds to not using a value. However, linear type systems still require explicit uncomputation that ends in *assertive termination* [23]: the programmer must (manually) assert that uncomputation correctly resets temporary values to `0`. ReQWire [58] introduced syntactic conditions sufficient to verify assertive termination. However, ReQWire can only verify explicitly provided uncomputation (except for purely classical oracle functions, see below), and cannot statically reason across function boundaries as unlike Silq, its type system does not address uncomputation.

Further, linear type systems introduce significant syntactic overhead for constant (i.e., read-only) variables where enforcing linearity is not necessary. Fig. 3.2 demonstrates this in QWire code (right), where encoding only Lines 4–6 from Silq (left) requires 19 lines of code, even when we generously assume built-in primitives and omit parts of the required type annotations. We note that while QWire [54] does not explicitly claim to be high-level, we are not aware of more high-level quantum languages that achieve a level of safety similar to QWire — even though it cannot prevent implicit measurement caused by incorrect manual uncomputation.

In contrast, Silq uses a linear type system to detect values removed from consideration (which are automatically uncomputed), but reduces programming overhead by treating constant variables non-linearly. This more liberal treatment of constant variables is possible because they can be safely duplicated and uncomputed whenever convenient.

BENNETT'S CONSTRUCTION Various quantum languages, including Quipper, ReVerC [59], and ReQWire, support Bennett's construction [60], which can lift purely classical (oracle) functions to quantum inputs, automatically uncomputing all temporary values computed in the function. Concretely, this standard approach (i) lifts all primitive classical operations in the oracle function to quantum operations, (ii) evaluates the function while preserving all temporary values, (iii) uses the function's result, and (iv) uncomputes temporary values by reversing step (ii). Bennett's construction is also supported by Qumquat[7], which skips step (i) above by annotating quantum functions as `@qq.garbage` and calling them with notation analogous to Quipper's `with_computed`.

However, Bennett's construction is unsafe when the oracle function contains quantum operations: as we demonstrate in App. A.3, it can fail to drop temporary values without side-effects. In contrast, Silq safely uncomputes temporary values in functions containing quantum operations.

Importantly, Silq's workflow when defining oracle functions is different from existing languages: while the latter typically require programmers to define a purely classical oracle function and then apply Bennett's construction, Silq programmers can define oracle functions directly using primitive quantum operations, implicitly relying on Silq's automatic uncomputation.

SUMMARY In contrast to previous languages, Silq (i) enables intuitive yet physical semantics and (ii) statically prevents errors that are not detected in existing languages, while (iii) avoiding the notational overhead associated with languages that achieve (less) static safety (e.g., QWire).

3.3 OVERVIEW OF SILQ

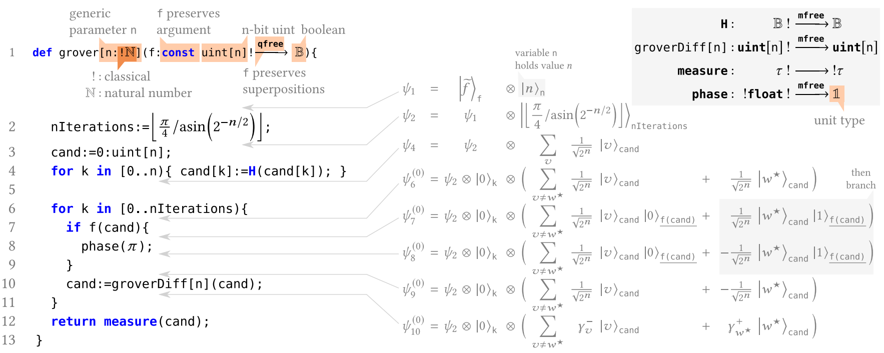
We now illustrate Silq on Grover's algorithm, a widely known quantum search algorithm [61], [47, §6.1]. It can be applied to any NP problem, where finding the solution may be hard, but verification of a solution is easy.

Fig. 3.3 shows a Silq implementation of `grover`. Its input is an oracle function `f` from (quantum) unsigned integers represented with `n` qubits to (quantum) booleans, mapping all but one input $w^\star$ to 0. Here, Silq uses the generic parameter `n` to parametrize the input type `uint[n]` of `f`. Then, `grover` outputs an `n`-bit unsigned integer w which is equal to $w^\star$ with high probability.

3.3.1 *Silq Annotations*

CLASSICAL TYPES The first argument of `grover` is a *generic parameter* `n`, used to parametrize `f`. It has type $!\mathbb{N}$, which indicates classical natural numbers of arbitrary size. Here, annotation $!$ indicates `n` is classically known, i.e., it is in a basis state (not in superposition), and we can manipulate it

7 Available at `https://github.com/patrickrall/Qumquat`, commit 27d6794



```
def grover[n:!ℕ](f:const uint[n]!$\xrightarrow{\texttt{qfree}}$ 𝔹){
  nIterations:=$\left\lfloor \frac{\pi}{4}/\mathrm{asin}\left(2^{-n/2}\right)\right\rfloor$;
  cand:=0:uint[n];
  for k in [0..n){ cand[k]:=H(cand[k]); }

  for k in [0..nIterations){
    if f(cand){
      phase($\pi$);
    }
    cand:=groverDiff[n](cand);
  }
  return measure(cand);
}
```

$$\begin{aligned}
\mathbf{H}&: \mathbb{B}\,! \xrightarrow{\texttt{mfree}} \mathbb{B}\\
\texttt{groverDiff[n]}&: \mathbf{uint}[n]\,! \xrightarrow{\texttt{mfree}} \mathbf{uint}[n]\\
\mathbf{measure}&: \tau\,! \longrightarrow !\tau\\
\mathbf{phase}&: !\mathbf{float}\,! \xrightarrow{\texttt{mfree}} \mathbb{1}
\end{aligned}$$

$$\begin{aligned}
\psi_1 &= \left|\tilde{f}\right\rangle_{\mathtt{f}} \otimes |n\rangle_{\mathtt{n}}\\
\psi_2 &= \psi_1 \otimes \left|\left\lfloor \frac{\pi}{4}/\mathrm{asin}\left(2^{-n/2}\right)\right\rfloor\right\rangle_{\mathtt{nIterations}}\\
\psi_4 &= \psi_2 \otimes \sum_{v} \frac{1}{\sqrt{2^n}} |v\rangle_{\mathtt{cand}}\\
\psi_6^{(0)} &= \psi_2 \otimes |0\rangle_{\mathtt{k}} \otimes \left(\sum_{v\neq w^\star} \frac{1}{\sqrt{2^n}} |v\rangle_{\mathtt{cand}} + \frac{1}{\sqrt{2^n}} |w^\star\rangle_{\mathtt{cand}} \right)\\
\psi_7^{(0)} &= \psi_2 \otimes |0\rangle_{\mathtt{k}} \otimes \left(\sum_{v\neq w^\star} \frac{1}{\sqrt{2^n}} |v\rangle_{\mathtt{cand}} |0\rangle_{\underline{\mathtt{f(cand)}}} + \frac{1}{\sqrt{2^n}} |w^\star\rangle_{\mathtt{cand}} |1\rangle_{\underline{\mathtt{f(cand)}}} \right)\\
\psi_8^{(0)} &= \psi_2 \otimes |0\rangle_{\mathtt{k}} \otimes \left(\sum_{v\neq w^\star} \frac{1}{\sqrt{2^n}} |v\rangle_{\mathtt{cand}} |0\rangle_{\underline{\mathtt{f(cand)}}} + -\frac{1}{\sqrt{2^n}} |w^\star\rangle_{\mathtt{cand}} |1\rangle_{\underline{\mathtt{f(cand)}}} \right)\\
\psi_9^{(0)} &= \psi_2 \otimes |0\rangle_{\mathtt{k}} \otimes \left(\sum_{v\neq w^\star} \frac{1}{\sqrt{2^n}} |v\rangle_{\mathtt{cand}} + -\frac{1}{\sqrt{2^n}} |w^\star\rangle_{\mathtt{cand}} \right)\\
\psi_{10}^{(0)} &= \psi_2 \otimes |0\rangle_{\mathtt{k}} \otimes \left(\sum_{v\neq w^\star} \gamma_v^- |v\rangle_{\mathtt{cand}} + \gamma_{w^\star}^+ |w^\star\rangle_{\mathtt{cand}} \right)
\end{aligned}$$

FIGURE 3.3: Grover's algorithm in Silq. We provide an unannotated version, including `groverDiff`, in App. A.2. The top-right box shows the type of all used functions. The shown sums range over all n-bit unsigned integers $\{0, \ldots, 2^n - 1\}$.

classically. For example, `0` has type !𝔹. In contrast, **H**(0) applies Hadamard **H** (defined shortly) to `0` and yields $\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle \right)$. Thus, **H**$(0)$ is of type 𝔹 and not of (classical) type !𝔹.

In general, we can liberally use classical variables like normal variables on a classical computer: we can use them multiple times, or drop them. We also annotate parameter `f` as classical, writing the annotation as $\tau ! \!\rightarrow \tau'$ instead of $!\tau \rightarrow \tau'$ to avoid the ambiguity between $!(\tau \rightarrow \tau')$ and $(!\tau) \rightarrow \tau'$. [8]

QFREE FUNCTIONS The type of `f` is annotated as **qfree**, which indicates the semantics of `f` can be described classically: we can capture the semantics of a **qfree** function `g` as a function $\overline{g}\colon S \rightarrow S'$ for ground sets S and S'. Note that since S' is a ground set, $\overline{g}$ can never output superpositions. Then, `g` acting on $\sum_{v\in S} \gamma_v |v\rangle$ yields $\sum_{v\in S} \gamma_v |\overline{g}(v)\rangle$, where for simplicity $\sum_{v\in S} \gamma_v |v\rangle$ does not consider other qubits untouched by `g`.

For example, the **qfree** function **X** flips the bit of its input, mapping $\sum_{v=0}^{1} \gamma_v |v\rangle$ to $\sum_{v=0}^{1} \gamma_v |\overline{\mathbf{X}}(v)\rangle$, for $\overline{\mathbf{X}}(v) = 1 - v$. In contrast, the Hadamard transform **H** maps $\sum_{v=0}^{1} \gamma_v |v\rangle$ to $\sum_{v=0}^{1} \gamma_v \frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^v |1\rangle \right)$. As this semantics cannot be described by a function on ground sets, **H** is not **qfree**.

8 Annotating functions as classical indicates that their function bodies are classically known (at runtime). We note that classical functions can still perform quantum operations: for example, **H**: $\mathbb{B}! \xrightarrow{\texttt{mfree}} \mathbb{B}$ is classical, meaning that the quantum operations performed by **H** are classically known.

CONSTANT PARAMETERS Note that **X** (introduced above) transforms its input — it does not preserve it. In contrast, the parameter of `f` is annotated as **`const`**, indicating `f` preserves its input, i.e., treats it as a read-only control. Thus, running `f` on $\sum_{v \in S} \gamma_v \, |v\rangle$ yields $\sum_{v \in S} \gamma_v \, |v\rangle \otimes \varphi_v$, where φ_v follows the semantics of `f`. Because `f` is also **`qfree`**, $|v\rangle \otimes \varphi_v = |\overline{f}(v)\rangle$ for some $\overline{f}\colon S \to S \times S'$. Combining both, we conclude that $\overline{f}(v) = (v, \tilde{f}(v))$ for some function $\tilde{f}\colon S \to S'$.

An example instantiation of f is NOT, which maps $\gamma_0|0\rangle + \gamma_1|1\rangle$ to $\gamma_0|0,1\rangle + \gamma_1|1,0\rangle$. Here, $\widetilde{\text{NOT}}\colon \{0,1\} \to \{0,1\}$ maps $v \mapsto 1 - v$ while $\overline{\text{NOT}}\colon \{0,1\} \to \{0,1\} \times \{0,1\}$ maps $v \mapsto (v, 1 - v) = (v, \widetilde{\text{NOT}}(v))$.

Function parameters not annotated as **`const`** are not accessible after calling the function — the function *consumes* them. For example, `groverDiff` consumes its argument (see top-right box in Fig. 3.3). Hence, the call in Line 10 consumes `cand`, transforms it, and writes the result into a new variable with the same name `cand`. Similarly, `measure` in Line 12 consumes `cand` by measuring it.

LIFTED FUNCTIONS We introduce the term **`lifted`** to describe **`qfree`** functions with exclusively **`const`** parameters, as such functions are crucial for uncomputation. In particular, we could write the type of `f` as $\texttt{uint}[n] \xrightarrow{\texttt{lifted}} \mathbb{B}$.

3.3.2 *Silq Semantics*

Next, we discuss the semantics of Silq on `grover`.

INPUT STATE In Fig. 3.3, the state of the system after Line 1 is ψ_1, where the state of $\texttt{f:uint[n]!} \xrightarrow{\texttt{qfree}} \mathbb{B}$ is described in terms of a function $\tilde{f}\colon \{0, \ldots, 2^n - 1\} \to \{0,1\}$. We note that later, our formal semantics represents the state of functions as Silq-Core expressions (§3.6). However, as the semantics of `f` can be captured by $\tilde{f}$, this distinction is irrelevant here. Next, Line 2 initializes the classical variable `nIterations`, yielding ψ_2.

SUPERPOSITIONS Lines 3–4 result in state ψ_4, where `cand` holds the equal superposition of all n-bit unsigned integers. To this end, Line 4 updates the k^{th} bit of `cand` by applying the Hadamard transform **H** to it.

LOOPS The loop in Line 6 runs `nIterations` times. Each loop iteration increases the coefficient of $|w^\star\rangle$, thus increasing the probability of measuring $w^\star$ in Line 12. We now discuss the first loop iteration ($k = 0$). It starts from state $\psi_6^{(0)}$ which introduces variable `k`. For convenience of presentation, $\psi_6^{(0)}$ splits the superposition into $w^\star$ and all other values.

CONDITIONALS Intuitively, Lines 7–9 flip the sign of those coefficients for which `f(cand)` returns true. To this end, we first evaluate `f(cand)` and place the result in a temporary variable $\underline{\texttt{f(cand)}}$, yielding state $\psi_7^{(0)}$. Here and in the following, we write $\underline{e}$ for a temporary variable that contains the result of evaluating e. Then, we determine those summands of $\psi_7^{(0)}$ where $\underline{\texttt{f(cand)}}$ is true (marked as "then branch" in Fig. 3.3), and run **phase**(π) on them. This yields $\psi_8^{(0)}$, as **phase**(π) flips the sign of coefficients. Lastly, we drop $\underline{\texttt{f(cand)}}$ from the state, yielding $\psi_9^{(0)}$.

GROVER'S DIFFUSION OPERATOR Completing the explanations of our example, Line 10 applies Grover's diffusion operator to `cand`. Its implementation consists of 6 lines of code (see App. A.2). It increases the weight of solution $w^\star$, obtaining $\|\gamma_{w^\star}^+\| > \|\frac{1}{\sqrt{2^n}}\|$, and decreases the weight of non-solutions $v \neq w^\star$, obtaining $\|\gamma_v^-\| < \|\frac{1}{\sqrt{2^n}}\|$. After one loop iteration, this results in state $\psi_{10}^{(0)}$. Repeated iterations of the loop in Lines 6–11 further increase the coefficient of $w^\star$, until it is approximately 1. Thus, measuring `cand` in Line 12 returns $w^\star$ with high probability.

3.3.3 *Uncomputation*

While dropping the temporary value $\underline{\texttt{f(cand)}}$ from $\psi_8^{(0)}$ is intuitive, achieving this physically requires uncomputation.

Without uncomputation, simply removing $\underline{\texttt{f(cand)}}$ from consideration in Line 9 would induce an implicit measurement. [9] Concretely, measuring and dropping $\underline{\texttt{f(cand)}}$ would collapse $\psi_8^{(0)}$ to one of the following two states (ignoring `f`, `n`, and `k`):

$$\psi_8^{(0,0)} = \sum_{v \neq w^\star} \frac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}} \quad \text{or} \quad \psi_8^{(0,1)} = -\frac{1}{\sqrt{2^n}} |w^\star\rangle_{\texttt{cand}} .$$

9 Formally, this corresponds to taking the partial trace over $\underline{\texttt{f(cand)}}$.

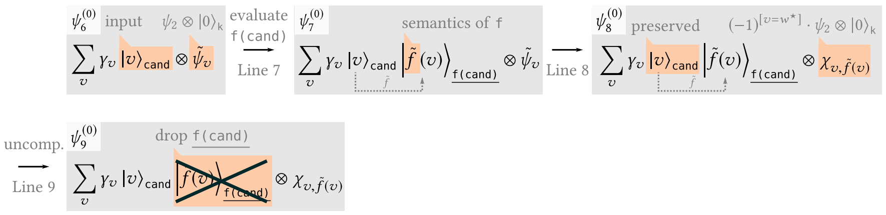

$\psi_6^{(0)}$ input $\psi_2 \otimes |0\rangle_k$
$\sum_v \gamma_v |v\rangle_{cand} \otimes \tilde{\psi}_v$
evaluate f(cand)
Line 7
$\psi_7^{(0)}$ semantics of f
$\sum_v \gamma_v |v\rangle_{cand} |\tilde{f}(v)\rangle_{f(cand)} \otimes \tilde{\psi}_v$
$\tilde{f}$
Line 8
$\psi_8^{(0)}$ preserved $(-1)^{[v=w^\star]} \cdot \psi_2 \otimes |0\rangle_k$
$\sum_v \gamma_v |v\rangle_{cand} |\tilde{f}(v)\rangle_{f(cand)} \otimes \chi_{v,\tilde{f}(v)}$
$\tilde{f}$
uncomp.
Line 9
$\psi_9^{(0)}$ drop f(cand)
$\sum_v \gamma_v |v\rangle_{cand} \cancel{|\tilde{f}(v)\rangle_{f(cand)}} \otimes \chi_{v,\tilde{f}(v)}$

FIGURE 3.4: Uncomputation of `f(cand)` is safe. Orange boxes show the correspondence to Fig. 3.3 where $(-1)^{[v=w^\star]}$ equals -1 if $v = w^\star$ and 1 otherwise.

In this case, as the probability of obtaining $\psi_8^{(0,1)}$ is only $\frac{1}{2^n}$, `grover` returns the correct result $w^\star$ with probability $\frac{1}{2^n}$, i.e., it degrades to random guessing.

Without correct intervention from the programmer, all quantum languages before Silq would induce an implicit measurement in Line 9, or reject `grover`. This is unfortunate as `grover` cleanly and concisely captures the programmer's intent. In contrast, Silq achieves the intuitive semantics of dropping $\underline{\texttt{f(cand)}}$ from $\psi_8^{(0)}$, using uncomputation. In general, uncomputing x is possible whenever in every summand of the state, the value of x can be reconstructed (i.e., determined) from all other values in this summand. Then, reversing the operations of this reconstruction removes x from the state.

AUTOMATIC UNCOMPUTATION To ensure that uncomputing $\underline{\texttt{f(cand)}}$ is possible, the type system of Silq ensures that `f(cand)` is **`lifted`**, i.e., (i) `f` is **`qfree`** and (ii) `cand` is **`const`**: it is preserved until uncomputation in Line 9.

Fig. 3.4 illustrates why this is sufficient. Evaluating `f` in Line 7 adds a temporary variable $\underline{\texttt{f(cand)}}$ to the state, whose value can be computed from `cand` using $\tilde{f}$ (as `f` is **`qfree`** and `cand` is **`const`**). Then, Line 8 transforms the remainder $\tilde{\psi}_v$ of the state to $\chi_{v,\tilde{f}(v)}$. The exact effect of Line 8 on the state is irrelevant for uncomputation, as long as it preserves `cand`, ensuring we can still reconstruct $\underline{\texttt{f(cand)}}$ from `cand` in $\psi_8^{(0)}$. Thus, reversing the operations of this reconstruction (i.e., reversing `f`) uncomputes $\underline{\texttt{f(cand)}}$ and yields $\psi_9^{(0)}$.

```
def useConsumed(x:𝔹){
  y := H(x); // consumes x
  return (x,y);
} // undefined identifier x

def useConsumedFixed(const x:𝔹){
  // $\psi_1 = \sum_{v=0}^{1} \gamma_v \left|v\right\rangle_{\mathtt{x}}$
  // $\psi_2 = \sum_{v=0}^{1} \gamma_v \left|v\right\rangle_{\mathtt{x}} \otimes \left|v\right\rangle_{\underline{\mathtt{x}}}$
  y := H(x);
  // $\psi_3 = \sum_{v=0}^{1} \gamma_v \left|v\right\rangle_{\mathtt{x}} \otimes \frac{1}{\sqrt{2}} \left(\left|0\right\rangle_{\mathtt{y}} + (-1)^v \left|1\right\rangle_{\mathtt{y}}\right)$
  return (x,y);
}

def discard[n:!ℕ](x:uint[n]){
  y := x % 2; // '%' supports quantum inputs
  return y;
} // parameter 'x' is not consumed (but caller expects it to be consumed)

def nonQfree(const x:𝔹,y:𝔹){
  if H(x) { y := X(y); }
  return y;
} // non-lifted quantum expression must be consumed

def nonConst(c:𝔹){
  if X(c) { phase(π); } // X consumes c
} // non-lifted quantum expression must be consumed

def nonConstFixed(const c:𝔹){
  // $\psi_1 = \sum_{v=0}^{1} \gamma_v \left|v\right\rangle_{\mathtt{c}}$
  if X(c) { phase(π); }
  // $\psi_2 = \sum_{v=0}^{1} (-1)^{1-v} \gamma_v \left|v\right\rangle_{\mathtt{c}}$
}

def condMeas(const c:𝔹,x:𝔹){
  if c { x := measure(x); }
} // cannot call function
// 'measure[𝔹]' in 'mfree' context

def revMeas(){
  return reverse(measure);
} // reversed function must be mfree
```

FIGURE 3.5: Examples of invalid Silq programs, their error messages, and possible fixes (where applicable).

3.3.4 *Preventing Errors: Rejecting Invalid Programs*

Fig. 3.5 demonstrates how the type system of Silq rejects invalid programs. We note that the presented examples are not exhaustive — we discuss additional challenges in §3.5.

ERROR: USING CONSUMED VARIABLES In `useConsumed`, **`H`** consumes `x` and stores its result in `y`. Then, it accesses `x`, which leads to a type error as `x` is no longer available.

Assuming we want to preserve `x`, we can fix this code by marking `x` as **`const`** (see `useConsumedFixed`). Then, instead of consuming `x` in the call to **`H`** (which is disallowed as `x` must be preserved), Silq implicitly duplicates `x`, resulting in ψ_2, and then only consumes the duplicate $\underline{\texttt{x}}$.

IMPLICIT DUPLICATION It is always safe to implicitly duplicate constant variables, as such duplicates can be uncomputed (in `useConsumedFixed`, uncomputation is not necessary as the duplicate is consumed). In contrast, it is typically impossible to uncompute duplicates of consumed quantum variables, which may not be available for uncomputation later. Hence, Silq treats constant variables non-linearly (they can be duplicated or ignored), but treats non-constant variables linearly (they must be used exactly once).

We note that duplication $\sum_v \gamma_v |v\rangle \mapsto \sum_v \gamma_v |v\rangle |v\rangle$ is physical and can be implemented using `CNOT`, unlike the unphysical cloning $\sum_v \gamma_v |v\rangle \mapsto (\sum_v \gamma_v |v\rangle) \otimes (\sum_v \gamma_v |v\rangle) = \sum_{v,w} \gamma_v\gamma_w |v\rangle |w\rangle$ discussed in §2.2.

ERROR: DISCARDING VARIABLES Function `discard` does not annotate `x` as **`const`**, meaning that its callers expect it to consume `x`. However, the body of `discard` does not consume `x`, hence any caller of `discard` would silently discard `x`, falsely assuming that `discard` would consume `x`. As the callee does not know if `x` can be uncomputed, Silq rejects this code. A possible fix is annotating `x` as **`const`**, which would be in line with preserving `x` in the function body.

ERROR: UNCOMPUTATION WITHOUT QFREE Silq rejects the function `nonQfree`, as **`H`**`(x)` is not **`lifted`** (since **`H`** is not **`qfree`**), and hence its result cannot be automatically uncomputed. Indeed, automatic uncomputation of **`H`**`(x)` is not possible in this case, intuitively because **`H`** introduces additional entanglement preventing uncomputation in the end. We provide a more

```
def f(x:𝔹){ // x is not const
    if g(x){  // x is temporarily marked as const
        x:=H(x); // cannot consume const variable x
    }
    // x is no longer const
    return x;
}

def g(const x:𝔹)qfree{
    return x;
}
```

Figure 3.6: Temporarily marking variables as constant.

detailed mathematical derivation of this subtle fact in App. A.3. To prevent this case, Silq only supports uncomputing **qfree** expressions.

We note that because `x` is **const** in `nonQfree`, **H** does not consume it, but a duplicate of `x`.

error: uncomputation without const Silq rejects the function `nonConst`, as `X(c)` is not **lifted** (since it consumes `c`). Indeed, automatic uncomputation is not possible in this case, as the original value of `c` is not available for uncomputation of `X(c)`. To get this code to type-check, we can mark `c` as **const** (see `nonConstFixed`) to clarify that `c` should remain in the context. Then, Silq automatically duplicates `c` before calling **X**, which thus consumes a duplicate of `c`, leaving the original `c` available for later uncomputation. Note that `X(c)` is automatically uncomputed in `nonConstFixed`.

temporary constants In contrast to `nonConst`, which consumes `c`, `grover` does not consume `cand` in Line 7 (Fig. 3.3), even though `cand` is not annotated as **const** either. This is because Silq temporarily annotates `cand` as **const** in `grover`. In general, Silq allows temporarily annotating some variables as **const** for the duration of a statement or a consumed subexpression. Our implementation determines which variables to annotate as **const** as follows: If a variable is encountered in a position where it is not expected to be **const** (as in `X(c)`), it is consumed, and therefore any further occurrence of that variable will result in an error (whether **const** or not). If a variable is encountered in a position where it is expected to be **const** (as in `f(cand)`), we temporarily mark it as **const** until the innermost enclosing statement or consumed subexpression finishes type checking. Fig. 3.6 shows such an example, where `x` is temporarily marked as **`const`**.

MFREE Silq's main advantage over existing quantum languages is its safe, automatic uncomputation, enabled by its novel annotations **`const`** and **`qfree`**. To ensure all Silq programs are physical (i.e., can be physically realized on a QRAM), we leverage one additional annotation **`mfree`**, indicating a function does not perform measurements. This allows us to detect (and thus prevent) attempts to reverse measurements and to apply measurements conditioned on quantum values.

ERROR: CONDITIONAL MEASUREMENT Silq rejects `condMeas`, as it applies a measurement conditioned on quantum variable `c`. This is not realizable on a QRAM, as the `then`-branch requires a physical action and we cannot determine whether or not we need to carry out the physical action without measuring the condition. However, changing the type of `c` to $!\mathbb{B}$ would fix this error, as conditional measurement *is* possible if `c` is classical. We note that Silq could also detect this error if measurement was hidden in a function passed to `condMeas`, as this function would not be **`mfree`**. Here, it is crucial that Silq disallows implicit measurement — otherwise, it would be hard to determine which functions are **`mfree`**.

REVERSE Silq additionally also supports reversing functions, where expression **`reverse`**(f) returns the inverse of function f. In general, all quantum operations except measurement describe linear isometries (see Chapter 2) and are thus injective. Hence, if f is also surjective (and thus bijective), we can reverse it, meaning **`reverse`**(f) is well-defined on all its inputs.

REVERSE RETURNS UNSAFE FUNCTIONS In case f is not surjective, **`reverse`**(f) is only well-defined on the range of f. Hence, it is the programmer's responsibility to ensure reversed functions never operate on invalid inputs.

For example, `y:=`**`dup`**`(x)` duplicates `x`, mapping $\sum_v \gamma_v \left|v\right\rangle_{\mathtt{x}}$ to $\sum_v \gamma_v \left|v\right\rangle_{\mathtt{x}} \left|v\right\rangle_{\mathtt{y}}$. Thus, **`reverse`**`(`**`dup`**`)(x,y)` operates on states $\sum_v \gamma_v \left|v\right\rangle_{\mathtt{x}} \left|v\right\rangle_{\mathtt{y}} \otimes \tilde{\psi}_v$, for which it yields $\sum_v \gamma_v \left|v\right\rangle_{\mathtt{x}} \otimes \tilde{\psi}_v$, uncomputing `y`. On other states, **`reverse`**`(`**`dup`**`)` is undefined. As **`reverse`**`(`**`dup`**`)` is generally useful for (unsafe) uncomputation, we introduce its (unsafe) shorthand **`forget`**.

When realizing a reversed function on a QRAM, the resulting program is defined on all inputs but only behaves correctly on valid inputs. For example, we can implement `reverse(dup)(x,y)` by `if x{ y:=X(y); }` and then discarding `y`, which has unintended side-effects (due to implicit measurement) unless originally `x==y`.

ERROR: REVERSING MEASUREMENT Silq rejects `revMeas` as it tries to reverse a measurement, which is physically impossible according to the laws of quantum mechanics. Thus, **`reverse`** only operates on **`mfree`** functions.

DISCUSSION: ANNOTATIONS AS NEGATED EFFECTS We can view annotations **`mfree`** and **`qfree`** as indicating the absence of effects: **`mfree`** indicates a function does not perform a measurement, while **`qfree`** indicates the function does not introduce quantum superposition. As we will see later, all **`qfree`** functions in Silq are also **`mfree`**.

3.4 THE SILQ-CORE LANGUAGE FRAGMENT

In this section, we present the language fragment Silq-Core of Silq, including syntax (§3.4.1) and types (§3.4.2).

Silq-Core is selected to contain Silq's key features, in particular all its annotations. Compared to Silq, Silq-Core omits features (such as the imperative fragment and dependent types) that distract from its key insights. We note that in our implementation, we type-check and simulate full Silq.

$$e ::= c \mid x \mid \texttt{measure} \mid \texttt{reverse} \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid$$
$$e'\underbrace{(e_1, \dots, e_n)}_{(\vec{e})} \mid \lambda\underbrace{(\beta_1 x_1 \colon \tau_1, \dots, \beta_n x_n \colon \tau_n)}_{(\vec{\beta}\vec{x}\colon\,\vec{\tau})}.e$$

expressions

types

$$\tau ::= \mathbb{1} \mid \mathbb{B} \mid \bigtimes_{k=1}^{n} \tau_k \mid \bigtimes_{k=1}^{n} \beta_k \tau_k \ ! \xrightarrow{\alpha} \tau' \mid\ !\tau$$

annotations

$$\alpha \subseteq \{\texttt{mfree}, \texttt{qfree}\}$$
$$\beta \subseteq \{\texttt{const}\}$$

FIGURE 3.7: Syntax, types, and annotations.

3.4.1 *Syntax of Silq-Core*

Fig. 3.7 summarizes the syntax of Silq-Core.

EXPRESSIONS Silq-Core expressions include constants and built-in functions (c), variables (x), measurement (**measure**), and reversing quantum operations (**reverse**). Further, its if-then-else construct **if** e **then** e_1 **else** e_2 is syntactically standard, but supports both classical ($!\mathbb{B}$) and quantum ($\mathbb{B}$) condition e. Function application $e'(\vec{e})$ explicitly takes multiple arguments. Likewise, lambda abstraction $\lambda(\vec{\beta}\vec{x}\colon \vec{\tau}).e$ describes a function with multiple parameters $\{x_i\}_{i=1}^{n}$ of types $\{\tau_i\}_{i=1}^{n}$, annotated by $\{\beta\}_{i=1}^{n}$, as discussed in §3.4.2 (next).

We note that Silq-Core can support tupling as a built-in function c.

UNIVERSALITY Assuming built-in functions c include **X** (enabling `CNOT` by **if** x {y:=**X**(y)}) and arbitrary operations on single qubits (e.g., enabled by **rotX**, **rotY**, and **rotZ**), Silq-Core is *universal for quantum computation*, i.e., it can approximate any quantum operation to arbitrary accuracy [47].

3.4.2 *Types and Annotations of Silq-Core*

Further, Fig. 3.7 introduces the types τ of Silq-Core.

PRIMITIVE TYPES Silq-Core types include standard primitive types, including $\mathbb{1}$, the singleton type that only contains the element "()", and $\mathbb{B}$, the Boolean type describing a single qubit. We note that it is straightforward to add other primitive types like integers or floats to Silq-Core.

PRODUCTS AND FUNCTIONS Silq-Core also supports products, where we often write $\tau_1 \times \cdots \times \tau_n$ for $\bigtimes_{k=1}^{n} \tau_k$, and functions, where ! emphasizes

$\Gamma ::= \beta_1 x_1 \colon \tau_1, \ldots, \beta_n x_n \colon \tau_n$ $\quad$ $\Gamma \overset{\alpha}{\vdash} e \colon \tau$

FIGURE 3.8: Typing judgments.

that functions are classically known (i.e., we do not discuss superpositions of functions). Function parameters and functions themselves may be annotated by β_i and α, respectively, as discussed shortly. As usual, $\times$ binds stronger than $\rightarrow$.

Finally, Silq-Core supports annotating types as classical.

ANNOTATIONS Fig. 3.7 also lists all Silq-Core annotations.

Our annotations express restrictions on the computations of Silq-Core expressions and functions, ensuring the physicality of its programs. For example, for quantum variable $x \colon \mathbb{B}$, the expression **if** x **then** $f(0)$ **else** $f(1)$ is only physical if f is **mfree** (note that x does not appear in the two branches).

3.5 TYPING RULES

In this section, we introduce the typing rules of Silq. Most importantly, they ensure that every sub-expression that is not consumed can be uncomputed, by ensuring these sub-expressions are **lifted**.

FORMAT OF TYPING RULES In Fig. 3.8, $\Gamma \overset{\alpha}{\vdash} e \colon \tau$ indicates an expression e has type τ under context Γ, and the evaluation of e is $\alpha \subseteq \{\texttt{qfree}, \texttt{mfree}\}$. For example, $x \colon \mathbb{B} \overset{\alpha}{\vdash} \texttt{H}(x) \colon \mathbb{B}$ for $\alpha = \{\texttt{mfree}\}$, where $\texttt{mfree} \in \alpha$ since evaluating $\texttt{H}(x)$ does not induce a measurement, and $\texttt{qfree} \notin \alpha$ since the effect of evaluating $\texttt{H}(x)$ cannot be described classically. We note that in general, $x \colon \tau \overset{\alpha}{\vdash} f(x) \colon \tau'$ if f has type $\tau! \xrightarrow{\alpha} \tau'$, i.e., the annotation of f determines the annotation of the turnstile $\vdash$.

A context Γ is a multiset $\{\beta_i x_i \colon \tau_i\}_{i \in \mathcal{I}}$ that assigns a type τ_i to each variable x_i, where $\mathcal{I}$ is a finite index set, and x_i may be annotated by $\texttt{const} \in \beta_i$, indicating that it will not be consumed during evaluation of e. As a shorthand, we often write $\Gamma = \vec{\beta}\vec{x} \colon \vec{\tau}$.

We write $\Gamma, \beta x \colon \tau$ for $\Gamma \uplus \{\beta x \colon \tau\}$, where $\uplus$ denotes the union of multisets. Analogously Γ, Γ' denotes $\Gamma \uplus \Gamma'$. In general, we require that types and

$$\frac{}{\beta x\colon \tau \overset{\texttt{mfree,qfree}}{\vdash} x\colon \tau}\,\text{var} \qquad \frac{\Gamma \overset{\alpha}{\vdash} e\colon \tau'}{\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'}\,!\text{W} \qquad \frac{\Gamma \overset{\alpha}{\vdash} e\colon \tau'}{\Gamma, \textbf{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'}\,\text{W}$$

$$\frac{\Gamma, x\colon !\tau, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'}{\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'}\,!\text{C} \qquad \frac{\Gamma, \textbf{const}\ x\colon \tau, \textbf{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'}{\Gamma, \textbf{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'}\,\text{C}$$

Figure 3.9: Typing variables, including weakening and contraction.

annotations of contexts can never be conflicting, i.e., $\beta x\colon \tau \in \Gamma$ and $\beta' x\colon \tau' \in \Gamma$ implies $\beta = \beta'$ and $\tau = \tau'$.

3.5.1 *Typing Constants and Variables*

If c is a constant of type τ, its typing judgement is given by $\varnothing \overset{\texttt{mfree,qfree}}{\vdash} c\colon \tau$. For example, $\varnothing \overset{\texttt{mfree,qfree}}{\vdash} \texttt{H}\colon \mathbb{B}! \xrightarrow{\texttt{mfree}} \mathbb{B}$. Here, we annotate the turnstile $\vdash$ as **qfree**, because evaluating expression **H** maps the empty state $|\rangle$ to $|\rangle \otimes |\texttt{H}\rangle_{\underline{\texttt{H}}}$, which can be described classically by $\overline{f}(|\rangle) = |\texttt{H}\rangle_{\underline{\texttt{H}}}$. In contrast, the function type of **H** is not **qfree**, as evaluating **H** can introduce quantum superposition. We provide the types of other selected built-in functions in App. A.4.2.

Likewise, the typing judgement of variables carries annotations **qfree** and **mfree** (rule var in Fig. 3.9), as all constants c and variables x in Silq-Core can be evaluated without measurement, and their semantics can be described classically. Further, both rules assume an empty context (for constants c) or a context consisting only of the evaluated variable (for variables), preventing ignoring variables from the context. To drop constant and classical variables from the context, we introduce an explicit weakening rule, discussed next.

weakening and contraction Fig. 3.9 further shows weakening and contraction typing rules for classical and constant variables. These rules allow us to drop classical and constant variables from the context (weakening rules !W and W) and duplicate them (contraction rules !C and C). For weakening, the interpretation of "dropping variable x" arises from reading the rule bottom-up, which is also the way our semantics operates (analogously for contraction).

We note that variables with classical type can be used more liberally than **const** variables (e.g., as `if`-conditions). Hence, annotating a classical

$$\frac{\Gamma_i \overset{\alpha_i}{\vdash} e_i\colon \tau_i \qquad \Gamma' \overset{\alpha'}{\vdash} e'\colon \bigtimes_{i=1}^{n} \beta_i\tau_i \;! \xrightarrow{\alpha'_{\text{func}}} \tau'}{\Gamma_1,\dots,\Gamma_n,\Gamma' \overset{\alpha''}{\vdash} e'(e_1,\dots,e_n)\colon \tau'}\ \text{func-eval}$$

$$\texttt{const} \in \beta_i \implies \texttt{qfree} \in \alpha_i \wedge \Gamma_i = \texttt{const}\ \vec{x}\colon \vec{\tau}'' \tag{3.1}$$

$$\texttt{qfree} \in \alpha'' \iff \texttt{qfree} \in \bigcap_i \alpha_i \cap \alpha' \cap \alpha'_{\text{func}} \tag{3.2}$$

$$\texttt{mfree} \in \alpha'' \iff \texttt{mfree} \in \bigcap_i \alpha_i \cap \alpha' \cap \alpha'_{\text{func}} \tag{3.3}$$

FIGURE 3.10: Typing rule and constraints for function calls.

variable as **`const`** has no effect. We annotate variables (not types) as **`const`** as our syntax does not allow partially consuming variables.

3.5.2 *Measurement*

We type **`measure`** as $\varnothing \overset{\texttt{mfree,qfree}}{\vdash}$ **`measure`**: $\tau\ ! \xrightarrow{}\ !\tau$, where the lack of **`const`** annotation for τ indicates **`measure`** consumes its argument, and $!\tau$ indicates the result is classical. We annotate the judgement itself as **`mfree`**, as evaluating the expression **`measure`** simply yields the function **`measure`**, without inducing a measurement. In contrast, the function type of **`measure`** itself is not **`mfree`** (indicated by $! \xrightarrow{}$), as evaluating the function **`measure`** on an argument induces a measurement. Thus, **`measure`**(0) is not **`mfree`**, as evaluating it induces a measurement: $\varnothing \vdash$ **`measure`**$(0)\colon !\mathbb{B}$.

3.5.3 *Function Calls*

Fig. 3.10 shows the typing rule for function calls $e'(e_1, \dots, e_n)$. Ignoring annotations, the rule follows the standard pattern, which we provide for convenience in App. A.4.1 (Fig. A.5).

We now discuss the annotation Constraints (3.1)–(3.3). Constraint (3.1) ensures that if a function leaves its argument constant (**`const`** $\in \beta_i$), e_i is **`lifted`**, i.e., **`qfree`** and depending only on **`const`** variables. In turn, this ensures that we can automatically uncompute e_i when it is no longer needed. We note that non-constant arguments (**`const`** $\notin \beta_i$) do not need to be uncomputed, as they are no longer available after being consumed within the call. To illustrate that Constraint (3.1) is critical, consider a function

$$\frac{\vec{\beta}\vec{x}\colon \vec{\tau}, \vec{\beta}'\vec{y}\colon !\vec{\tau}' \overset{\alpha}{\vdash} e\colon \tau''}{\vec{\beta}'\vec{y}\colon !\vec{\tau}' \overset{\substack{\texttt{mfree,}\\ \texttt{qfree}}}{\vdash} \lambda(\vec{\beta}\vec{x}\colon \vec{\tau}).e\colon \bigtimes_{i=1}^{n} \beta_i\tau_i \; ! \xrightarrow{\alpha} \tau''} \lambda\text{-abs}$$

FIGURE 3.11: Typing lambda abstraction.

$f\colon$ **`const`** $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, and a call $f(x+1,$ **`H`**$(x))$ with non-constant variable x in context Γ_1. This call must be rejected by Silq-Core as there is no way to uncompute $x+1$ after the call to f, since **`H`** consumes x. Indeed, since x is not **`const`** in Γ_1 even though **`const`** $\in \beta_1$, (3.1) does not hold.

Constraint (3.2) ensures an expression is only **`qfree`** if all its components are **`qfree`**, and if the evaluated function only uses **`qfree`** operations. Constraint (3.3) is analogous for **`mfree`**.

We note that Fig. 3.10 does not allow temporarily marking variables as **`const`**, as discussed in §3.3.4. To allow this, we can replace the top-left Γ_i in Fig. 3.10 by $\Gamma_i, \star_i$, where $\star_i =$ **`const`** $\Gamma_{i+1}, \ldots,$ **`const`** Γ_n if **`const`** $\notin \beta_i$, and $\star_i = \emptyset$ otherwise. This would allow us to temporarily treat variables as **`const`**, if they appear in a consumed expression e_i and they are consumed in a later expression e_j for $j > i$. Fig. 3.10 omits this for conciseness.

3.5.4 *Lambda Abstraction*

Fig. 3.11 shows the rule for lambda abstraction. Its basic pattern without annotations is again standard (App. A.4.1) . In terms of annotations, the rule enforces multiple constraints. First, it ensures that the annotation of the abstracted function follows the annotation α of the original typing judgment. Second, we tag the resulting type judgment as **`mfree`** and **`qfree`**, since function abstraction requires neither measurement nor quantum operations. Third, the rule allows capturing classical variables (y_i has type $!\tau_i$), but not quantum variables. This ensures that all functions in Silq-Core are classically known, i.e., can be described by a classical state.

3.5.5 *Reverse*

Fig. 3.12 shows the type of **`reverse`**. We only allow reversing functions without classical components in input or output types (indicated by $\not{!}$), as reconstructing classical components of inputs is typically impossible.

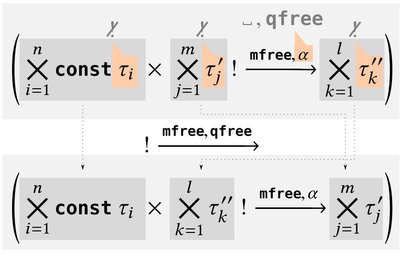


FIGURE 3.12: Type of **reverse**.

Concretely, types without classical components are (i) $\mathbb{1}$, (ii) $\mathbb{B}$, and (iii) products of types without classical components. In particular, this rules out all classical types $!\tau$, function types, and products of types with classical components.

The input to **reverse**, i.e., the function f to be reversed must be measure-free, because measurement is irreversible. Further, the function f may or may not be **qfree** (as indicated by a callout). Then, the type rule for **reverse** splits the input types of f into constant and non-constant ones. The depicted rule assumes the first parameters of f are annotated as constant, but we can easily generalize this rule to other orders. Based on this separation, **reverse** returns a function which starts from the constant input types and the output types of f, and returns the non-constant input types. The returned function **reverse**(f) is measure-free, and **qfree** if f is **qfree**.

As discussed in §3.3.4, **reverse** returns unsafe functions, and it is the programmer's responsibility to ensure reversed functions never operate on invalid inputs.

3.5.6 *Control Flow*

Even though **if** e **then** e_1 **else** e_2 is syntactically standard, it supports both classical and quantum conditions e. A classical condition induces classical control flow, while a non-classical (i.e., quantum) condition induces quantum control flow. In Fig. 3.13, we provide the typing rules for both cases, which follow the standard basic patterns when ignoring annotations (App. A.4.1).

QUANTUM CONTROL FLOW Constraint (3.4) ensures that e is **lifted** and can thus be uncomputed after the conditional, analogously to uncomputing constant arguments in Constraint (3.1). Constraint (3.5) requires both

$$\dfrac{\Gamma_c \overset{\alpha_c}{\vdash} e\colon \mathbb{B} \quad \Gamma \overset{\alpha_1}{\vdash} e_1 \colon \tau \quad \Gamma \overset{\alpha_2}{\vdash} e_2 \colon \tau}{\Gamma_c, \Gamma \overset{\alpha}{\vdash} \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \colon \underset{\cancel{!}}{\tau}}\ \text{ite-q}$$

$$\texttt{qfree} \in \alpha_c \wedge \Gamma_c = \texttt{const } \vec{x}\colon \vec{\tau}' \tag{3.4}$$
$$\texttt{mfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \cap \alpha \tag{3.5}$$
$$\texttt{qfree} \in \alpha \iff \texttt{qfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{3.6}$$

$$\dfrac{\Gamma_c \overset{\alpha_c}{\vdash} e\colon !\mathbb{B} \quad \Gamma \overset{\alpha_1}{\vdash} e_1 \colon \tau \quad \Gamma \overset{\alpha_2}{\vdash} e_2 \colon \tau}{\Gamma_c, \Gamma \overset{\alpha}{\vdash} \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \colon \tau}\ \text{ite-c}$$

$$\texttt{mfree} \in \alpha \iff \texttt{mfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{3.7}$$
$$\texttt{qfree} \in \alpha \iff \texttt{qfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{3.8}$$

FIGURE 3.13: Typing quantum (top) and classical (bottom) control flow.

branches to be `mfree`, which is important because we cannot condition a measurement on a quantum value (this would violate physicality). Further, it also requires the condition to be `mfree` (which is already implicitly ensured by Constraint (3.4) as all `qfree` expressions are also `mfree`), meaning the whole expression is `mfree`. Constraint (3.6) ensures that the resulting typing judgment gets tagged as `qfree` if all subexpressions are `qfree`. Finally, the rule does not allow the return type τ to contain classical components (indicated by $\cancel{!}$), as otherwise we could introduce unexpected superpositions of classical values.

CLASSICAL CONTROL FLOW Classical control flow requires the condition to be classical, in addition to our usual restrictions on annotations. Concretely, Constraints (3.7) and (3.8) propagate `mfree` and `qfree` annotations.

3.6 SEMANTICS OF SILQ-CORE

In this section, we discuss the operational semantics of Silq-Core. We use big-step semantics, as this is more convenient to define reverse and control flow.

Classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$

$$\begin{aligned}
\llbracket\mathbb{1}\rrbracket^{\mathsf{c}} &:= \{()\}\\
\llbracket\mathbb{B}\rrbracket^{\mathsf{c}} &:= \{()\}\\
\left\llbracket \bigtimes_{k=1}^{n} \tau_k \right\rrbracket^{\mathsf{c}} &:= \bigtimes_{k=1}^{n} \llbracket\tau_k\rrbracket^{\mathsf{c}}\\
\left\llbracket \bigtimes_{k=1}^{n} \beta_k\tau_k \mathrel{!\xrightarrow{\alpha}} \tau' \right\rrbracket^{\mathsf{c}} &:= \left\{ e,\sigma \;\middle|\; \begin{array}{c} \vec{y}\colon !\vec{\tau}'' \overset{\alpha'}{\vdash} e\colon \times_{k=1}^{n} \beta_k\tau_k \mathrel{!\xrightarrow{\alpha}} \tau', \\ \sigma \in \llbracket\vec{y}\colon !\vec{\tau}''\rrbracket^{\mathsf{c}} \end{array} \right\}\\
\llbracket!\tau\rrbracket^{\mathsf{c}} &:= \llbracket\tau\rrbracket^{\mathsf{c}} \times \llbracket\tau\rrbracket^{\mathsf{q}}
\end{aligned}$$

Quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$

$$\begin{aligned}
\llbracket\mathbb{1}\rrbracket^{\mathsf{q}} &:= \{()\}\\
\llbracket\mathbb{B}\rrbracket^{\mathsf{q}} &:= \{0,1\}\\
\left\llbracket \bigtimes_{k=1}^{n} \tau_k \right\rrbracket^{\mathsf{q}} &:= \bigtimes_{k=1}^{n} \llbracket\tau_k\rrbracket^{\mathsf{q}}\\
\left\llbracket \bigtimes_{k=1}^{n} \beta_k\tau_k \mathrel{!\xrightarrow{\alpha}} \tau' \right\rrbracket^{\mathsf{q}} &:= \{()\}\\
\llbracket!\tau\rrbracket^{\mathsf{q}} &:= \{()\}
\end{aligned}$$

FIGURE 3.14: Classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$ and quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$ to build the semantics $\llbracket\tau\rrbracket = \llbracket\tau\rrbracket^{\mathsf{c}} \times \mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$ of type τ.

3.6.1 *Semantics of Types*

We build the semantics $\llbracket\tau\rrbracket$ of type τ from a *classical set* $\llbracket\tau\rrbracket^{\mathsf{c}}$ and a *quantum ground set* $\llbracket\tau\rrbracket^{\mathsf{q}}$ as $\llbracket\tau\rrbracket = \llbracket\tau\rrbracket^{\mathsf{c}} \times \mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$. Note that $\llbracket\tau\rrbracket$ stores the classical and quantum parts of τ separately, which is in line with how a QRAM can physically store values of type τ. In particular, $\llbracket\tau\rrbracket^{\mathsf{q}}$ contains the ground set from which we build the Hilbert space $\mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$.

CLASSICAL SET AND QUANTUM GROUND SET Fig. 3.14 defines both the classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$ and the quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$ for all possible types τ. For type $\mathbb{1}$, both the classical set and the quantum ground set are the singleton set $\{()\}$. The (quantum) Boolean type $\mathbb{B}$ stores no classical information and hence, its classical set is again the singleton set. In contrast, its quantum ground set is $\{0,1\}$, for which $\mathcal{H}(\{0,1\})$ contains all superpositions of $|0\rangle$ and $|1\rangle$. The sets associated with the product type are standard. Functions store no quantum information, and hence their quantum ground set is $\{()\}$. In contrast, the classical set associated with a function type contains all

Type semantics

$$\begin{aligned}
[\![!\mathbb{B}]\!] &= [\![!\mathbb{B}]\!]^{\mathsf{c}} \times \mathcal{H}\left([\![!\mathbb{B}]\!]^{\mathsf{q}}\right) = \Big([\![\mathbb{B}]\!]^{\mathsf{c}} \times [\![\mathbb{B}]\!]^{\mathsf{q}}\Big) \times \mathcal{H}\left(\{()\}\right) \\
&= \Big(\{()\} \times \{0,1\}\Big) \times \mathcal{H}\left(\{()\}\right) \simeq \{0,1\} \times \mathcal{H}\left(\{()\}\right) \\
[\![!\mathbb{B} \times \mathbb{B}]\!] &= [\![!\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{c}} \times \mathcal{H}\left([\![!\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{q}}\right) = \Big([\![!\mathbb{B}]\!]^{\mathsf{c}} \times [\![\mathbb{B}]\!]^{\mathsf{c}}\Big) \times \mathcal{H}\left([\![!\mathbb{B}]\!]^{\mathsf{q}} \times [\![\mathbb{B}]\!]^{\mathsf{q}}\right) \\
&= \Big(\{()\} \times \{0,1\} \times \{()\}\Big) \times \mathcal{H}\left(\{()\} \times \{0,1\}\right) \simeq \{0,1\} \times \mathcal{H}\left(\{0,1\}\right) \\
[\![!\mathbb{B} \times \mathbb{B}]\!]^{+} &= \mathcal{H}\left([\![!\mathbb{B}]\!]^{\mathsf{c}} \times [\![\mathbb{B}]\!]^{\mathsf{q}}\right) \simeq \mathcal{H}\left(\{0,1\} \times \{0,1\}\right) = \mathcal{H}\left(\{0,1\}^2\right) \\
&= \left\{ \sum_{w \in \{0,1\}^2} \gamma_w \left|w\right\rangle \;\middle|\; \gamma_w \in \mathbb{C} \right\}
\end{aligned}$$

Context semantics

$$\begin{aligned}
[\![x\colon !\mathbb{B} \times \mathbb{B}]\!] &= [\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{c}} \times \mathcal{H}\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{q}}\right) \\
&\simeq \{(v)_x \mid v \in \{0,1\}\} \times \mathcal{H}\left(\{(v')_x \mid v' \in \{0,1\}\}\right) \\
&= \left\{ \Big((v)_x, \gamma_0 \left|0\right\rangle_x + \gamma_1 \left|1\right\rangle_x\Big) \;\middle|\; \begin{matrix} v \in \{0,1\} \\ \gamma_0, \gamma_1 \in \mathbb{C} \end{matrix} \right\} \\
[\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{+} &= \mathcal{H}\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{c}} \times [\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{\mathsf{q}}\right) \simeq \mathcal{H}\left(\left\{ \underbrace{\Big((v)_x, (v')_x\Big)}_{(v,v')_x} \;\middle|\; \begin{matrix} v \in \{0,1\} \\ v' \in \{0,1\} \end{matrix} \right\}\right) \\
&= \left\{ \sum_{w \in \{0,1\}^2} \gamma_w \left|w\right\rangle_x \;\middle|\; \gamma_w \in \mathbb{C} \right\}
\end{aligned}$$

Embedding

$$\begin{aligned}
\iota\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]\right) &\simeq \left\{ \iota\Big((v)_x, \gamma_0 \left|0\right\rangle_x + \gamma_1 \left|1\right\rangle_x\Big) \;\middle|\; v \in \{0,1\}, \gamma_0, \gamma_1 \in \mathbb{C} \right\} \\
&\simeq \left\{ \gamma_0 \left|v,0\right\rangle_x + \gamma_1 \left|v,1\right\rangle_x \;\middle|\; v \in \{0,1\}, \gamma_0, \gamma_1 \in \mathbb{C} \right\}
\end{aligned}$$

Figure 3.15: Example semantics of type $!\mathbb{B}$, type $!\mathbb{B} \times \mathbb{B}$, and context $x\colon !\mathbb{B} \times \mathbb{B}$.

expressions e of this type, and a state σ storing the variables captured in e. Finally, classical types $!\tau$ store no quantum information and hence their quantum ground set is $\{()\}$. In contrast, their classical set consists of (i) the classical set $[\![\tau]\!]^{\mathsf{c}}$ which remains classical and (ii) the quantum ground set $[\![\tau]\!]^{\mathsf{q}}$. As a straightforward consequence of our definition, duplicate classical annotations do not affect the semantics: $[\![!!\tau]\!] \simeq [\![!\tau]\!]$.

To illustrate the semantics of types, Fig. 3.15 provides semantics for two example types. In particular, $[\![!\mathbb{B}]\!]$ is isomorphic to $\{0,1\} \times \mathcal{H}\left(\{()\}\right)$ — note that this is not formally isomorphic to $\{0,1\}$ because $\mathcal{H}\left(\{()\}\right) = \{\gamma \left|()\right\rangle \mid \gamma \in \mathbb{C}\}$ tracks a (physically irrelevant) global phase $\gamma \in \mathbb{C}$.

extended semantic space Unfortunately, working with elements $(v, \varphi) \in [\![\tau]\!]$ for $v \in [\![\tau]\!]^{\mathsf{c}}$ and $\varphi \in \mathcal{H}\left([\![\tau]\!]^{\mathsf{q}}\right)$ is inconvenient because (i) every

$$\begin{aligned}
\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right] &:= \left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{c}} \times \mathcal{H}\left(\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{q}}\right) \\
&= \left\{(v_1)_{x_1}, \dots, (v_n)_{x_n} \mid v_i \in [\![\tau_i]\!]^{\mathsf{c}}\right\} \times \mathcal{H}\left(\left\{(v'_1)_{x_1}, \dots, (v'_n)_{x_n} \mid v'_i \in [\![\tau_i]\!]^{\mathsf{q}}\right\}\right) \\
\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{+} &:= \mathcal{H}\left(\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{c}} \times \left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{q}}\right) \\
&\simeq \left\{ \overset{\text{standard representation}}{\sum_{w_i \in [\![\tau_i]\!]^{\mathsf{c}} \times [\![\tau_i]\!]^{\mathsf{q}}} \gamma_{w_1,\dots,w_n} \left|w_1\right\rangle_{x_1} \otimes \dots \otimes \left|w_n\right\rangle_{x_n}} \;\middle|\; \gamma_{w_1,\dots,w_n} \in \mathbb{C} \right\}
\end{aligned}$$

FIGURE 3.16: Semantics $\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]$ and extended semantics $\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{+}$ of context $\vec{\beta}\vec{x}\colon \vec{\tau}$.

operation on (v, φ) needs to handle v and φ separately (as they are different mathematical objects) and (ii) some operations, like $(v, \varphi) + (v', \varphi')$ are not defined because $[\![\tau]\!]$ is not a vector space.

Therefore, we define the semantics of expressions (§3.6.2) and annotations (§3.6.4) on a more convenient, larger space that also allows superpositions of classical values:

$$[\![\tau]\!]^{+} := \mathcal{H}\left([\![\tau]\!]^{\mathsf{c}} \times [\![\tau]\!]^{\mathsf{q}}\right) = \mathcal{H}\left([\![\tau]\!]^{\mathsf{c}}\right) \otimes \mathcal{H}\left([\![\tau]\!]^{\mathsf{q}}\right).$$

Here, (i) the classical and quantum part of $\phi \in [\![\tau]\!]^{+}$ can be handled analogously and (ii) operation $+$ is defined on $[\![\tau]\!]^{+}$. We provide an example of this extended semantics in Fig. 3.15.

While elements of $[\![\tau]\!]$ can be naturally lifted to $[\![\tau]\!]^{+}$ via embedding $\iota\colon [\![\tau]\!] \to [\![\tau]\!]^{+}$ defined by $\iota(v, \varphi) := |v\rangle \otimes \varphi$, the larger space $[\![\tau]\!]^{+}$ also contains *invalid elements*, namely those where classical values are in superposition. In contrast, *valid elements* do not put classical values in superposition, i.e., the classical part of every summand in their superposition coincides. For example, state $|0\rangle \otimes \frac{1}{\sqrt{2}}\big(\,|0\rangle + |1\rangle\,\big)$ is valid for type $!\mathbb{B} \times \mathbb{B}$ but invalid for type $\mathbb{B} \times !\mathbb{B}$.

Overall, our semantics exclusively produces valid elements, as we formally prove in Thm. 3.6.1.

SEMANTICS OF CONTEXTS Fig. 3.16 provides the semantics of context $\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]$. Here, $(v_i)_{x_i}$ indicates that variable x_i stores value v_i. Fig. 3.15 provides semantics for an example context, where we write $|0\rangle_x$ as a shorthand for $|(0)_x\rangle$.

Analogously to $[\![\tau]\!]^+$, Fig. 3.16 also introduces the extended semantics $\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^+$ for contexts, and a standard representation that stores the classical and quantum value of variable x together in a single location $|v\rangle_x$. We use this representation throughout this chapter (including Fig. 3.3). Again, we illustrate this extended semantics in Fig. 3.15.

For contexts, the embedding $\iota\colon \left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right] \to \left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^+$ is

$$\iota\Big((\vec{v})_{\vec{x}}, \sum_{\vec{v}'} \gamma_{\vec{v}'} \,|\vec{v}'\rangle_{\vec{x}}\Big) = \sum_{\vec{v}'} \gamma_{\vec{v}'} \,|\vec{v}, \vec{v}'\rangle_{\vec{x}},$$

for $(\vec{v})_{\vec{x}} \in \left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{c}}$ and $\sum_{\vec{v}'} \gamma_{\vec{v}'} \,|\vec{v}'\rangle_{\vec{x}} \in \mathcal{H}\left(\left[\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\right]^{\mathsf{q}}\right)$. We illustrate this in Fig. 3.15 on an example context.

3.6.2 *Semantics of Expressions*

Our operational semantics evaluates an expression e in state ψ by constructing derivation trees whose structure follows the structure of our type derivations. Since e may contain measurements with probabilistic outcome, we provide an evaluation $\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi_i'$ for each possible sequence of measurement results, indicating that evaluating e (typed as $\Gamma \overset{\alpha}{\vdash} e\colon \tau$), on state ψ yields state ψ_i' with probability $\|\psi_i'\|^2$, assuming $\|\psi\|^2 = 1$ (see §2.1). If e is undefined for a given input ψ (possible since `reverse` returns unsafe functions), we do not provide any evaluation.

DOMAIN OF ψ, ψ' When our semantics evaluates e according to

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi',$$

it requires that $\psi \in \iota\left([\![\Gamma, \Delta]\!]\right)$. By construction, for a context of the form $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, output state ψ' lies in $[\![\texttt{const}\ \vec{x}\colon \vec{\tau}, \Delta, \underline{e}\colon \tau'']\!]^+$, i.e., we preserve constant variables $\vec{x}$ and the additional context Δ (discussed next), and store the value of e in a temporary variable $\underline{e}$.

Here, Δ is additional context containing the remainder of the state preserved while evaluating e. We illustrate the need for Δ in Fig. 3.17. Here, we cannot evaluate $(x \,||\, y)$ and z independently, as their values may be entangled in ψ. Hence, we must evaluate $x \,||\, y$ in a state that not only contains x, y (in the context when typing $x \,||\, y$, cp. blue box), but also z

$$\frac{\begin{array}{c}\cdots\\ \left[\texttt{const}\ z\colon \mathbb{B} \overset{\alpha}{\vdash} z\colon \mathbb{B} \,\middle|\, |1\rangle_z\, |0\rangle_x\, |0\rangle_y\, |0\rangle_{\underline{x\,||\,y}}\right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_y\, |1\rangle_z\, |0\rangle_{\underline{x\,||\,y}}\, |1\rangle_{\underline{z}} \\ \left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B} \overset{\alpha}{\vdash} x\,||\,y\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |0\rangle_y\, |1\rangle_z\right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_y\, |1\rangle_z\, |0\rangle_{\underline{x\,||\,y}}\end{array}}{\left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B}, \texttt{const}\ z\colon \mathbb{B} \overset{\alpha}{\vdash} (x\,||\,y)\,||\,z\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |0\rangle_y\, |1\rangle_z\right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_y\, |1\rangle_z\, |1\rangle_{\underline{x\,||\,y\,||\,z}}}$$

FIGURE 3.17: Part of derivation when evaluating expression $x\,||\,y\,||\,z$ on state $\psi = |0\rangle_x\, |0\rangle_y\, |1\rangle_z$. The corresponding typing rule is func-eval. For a complete semantics derivation tree that leverages more rules, see App. A.5.3.

(in Δ, cp. red box). After this, we evaluate z in a state containing z (in the context when typing z), x, y (in Δ), and the value of $x\,||\,y$ (stored as $\underline{x\,||\,y}$ in context Δ).

FORMAL SEMANTICS Here, we discuss the most important aspects of the formal semantics of Silq-Core expressions (see App. A.5.1 for details, and App. A.5.3 for an example). Recall that the structure of semantic derivation trees follows the structure of the type derivation trees, and hence, every type rule corresponds to a semantic derivation rule.

The semantics of evaluating a variable x is to rename x to $\underline{x}$ in the new state if x is consumed, and to duplicate x to $\underline{x}$ if x is constant. Contraction of constant variable x duplicates x, according to $\sum_v \gamma_v\, |v\rangle_x \otimes \tilde{\psi}_v \mapsto \sum_v \gamma_v\, |v\rangle_x\, |v\rangle_{\underline{x}} \otimes \tilde{\psi}_v$. Weakening of constant variables postpones uncomputing them until the end of the function body. When evaluating a function call $e'(e_1, \ldots, e_n)$, we uncompute the constant arguments e_i (i.e., preserved according to the signature of e') at the end of the function call. To reverse functions, we postpone the reversal until the reversed function is called. We handle control flow **if** e **then** e_1 **else** e_2 by separately evaluating e_1 (respectively e_2) in the part of the state where e is true (respectively false).

3.6.3 *Type Preservation*

Thm. 3.6.1 ensures that our semantics never produces invalid states ψ', meaning that the classical values in ψ' can never be in superposition (since ι only returns *valid* elements).

Theorem 3.6.1 (Type Preservation). *If we have*

$$\begin{aligned}
&\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}',\\
&\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{run} \psi', \text{ and}\\
&\psi \in \iota\left(\llbracket \Gamma, \Delta \rrbracket\right),
\end{aligned}$$

then ψ' *lies in* $\iota\left(\llbracket \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta \rrbracket\right)$.

We provide a proof for Thm. 3.6.1 in App. A.6. Here, $\iota\left(\llbracket \Gamma, \Delta \rrbracket\right)$ contains all elements of $\llbracket \Gamma, \Delta \rrbracket^{+}$ where classical values are not in superposition.

3.6.4 *Semantics of Annotations*

In the following, we show theorems formalizing the guarantees of annotations of Silq-Core expressions. We do not formally discuss the guarantees of annotations of Silq-Core functions, which are analogous. We note that the guarantees of ! were already discussed in §3.6.3.

PRESERVING CONSTANTS Thm. 3.6.2 ensures that constant variables are indeed preserved by Silq-Core.

Theorem 3.6.2 (Const Semantics). *If we have*

$$\begin{aligned}
&\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}',\\
&\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{run} \psi', \text{ and}\\
&\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},
\end{aligned}$$

then $\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$ *for some* $\chi_{\vec{v},\vec{w}}$.

We provide a proof for Thm. 3.6.2 in App. A.6.

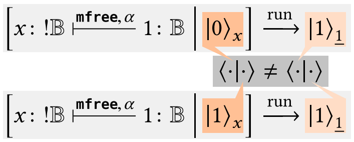

$\left[x\colon !\mathbb{B} \xmapsto{\texttt{mfree},\alpha} 1\colon \mathbb{B} \;\middle|\; |0\rangle_x\right] \xrightarrow{\text{run}} |1\rangle_{\underline{1}}$
$\langle\cdot|\cdot\rangle \neq \langle\cdot|\cdot\rangle$
$\left[x\colon !\mathbb{B} \xmapsto{\texttt{mfree},\alpha} 1\colon \mathbb{B} \;\middle|\; |1\rangle_x\right] \xrightarrow{\text{run}} |1\rangle_{\underline{1}}$

FIGURE 3.18: Non-isometry.

MFREE EXPRESSIONS We want to ensure that **mfree** expressions correspond to linear isometries, which in turn ensures we can physically implement their effect with quantum gates. However, this correspondence is non-trivial: Fig. 3.18 shows an example where $\xrightarrow{\text{run}}$ is not isometric because we drop a classical value from the input. Thm. 3.6.3 side-steps this issue, intuitively by ensuring that our semantics is isometric when the classical components of its input have fixed values.

Theorem 3.6.3 (Mfree Semantics). *If* $\mathbf{mfree} \in \alpha$, $\sigma \in [\![\Gamma, \Delta]\!]^c$,

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \iota(\sigma, \psi_1)\right] \xrightarrow{run} \psi_1' \quad for \quad \psi_1 \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right), and$$

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \iota(\sigma, \psi_2)\right] \xrightarrow{run} \psi_2' \quad for \quad \psi_2 \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right),$$

then $\langle\psi_1| \, |\psi_2\rangle = \langle\psi_1'| \, |\psi_2'\rangle$.

We provide a proof for Thm. 3.6.3 in App. A.6.

A useful interpretation of Thm. 3.6.3 states that $\xrightarrow{\text{run}}$ acts like an isometry on the subspace consistent with a fixed classical component $\sigma \in [\![\Gamma, \Delta]\!]^c$,

$$\{\iota(\sigma, \chi) \mid \chi \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right)\} \subseteq [\![\Gamma, \Delta]\!]^+ .$$

This corresponds to the intuition that in order to evaluate e on ψ_1, we can (i) extract the classical component σ from ψ_1, (ii) build a circuit C that realizes the linear isometry for this classical component and (iii) run C, yielding ψ_1'.

QFREE EXPRESSIONS Thm. 3.6.4 ensures that qfree expressions can be described by a function $\bar{f}$ on the ground sets.

Theorem 3.6.4 (Qfree Semantics). *If* $\Gamma \overset{\alpha}{\vdash} e\colon \tau''$ *for* $\mathbf{qfree} \in \alpha$ *and context* $\Gamma = \mathbf{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, *then there exists a function* $\bar{f}\colon [\![\Gamma]\!]^s \to [\![\mathbf{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'']\!]^s$ *on ground sets such that*

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \,|\sigma\rangle \otimes \tilde{\psi}_\sigma\right] \xrightarrow{run} \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \,|\bar{f}(\sigma)\rangle \otimes \tilde{\psi}_\sigma,$$

where $[\![\Gamma]\!]^s$ *is a shorthand for the ground set* $[\![\Gamma]\!]^c \times [\![\Gamma]\!]^q$ *on which the Hilbert space* $[\![\Gamma]\!]^+ = \mathcal{H}\left([\![\Gamma]\!]^s\right)$ *is defined.*

We provide a proof for Thm. 3.6.4 in App. A.6.

3.6.5 *Physicality*

Thm. 3.6.5 ensures Silq-Core programs can be physically realized on a QRAM. If we would change our semantics to abort on operations that are not physical, we could re-interpret Thm. 3.6.5 to guarantee *progress*, i.e., the absence of errors due to unphysical operations.

Theorem 3.6.5 (Physicality)**.** *The semantics of well-typed Silq programs is physically realizable on a QRAM.*

We provide a proof for Thm. 3.6.5 in App. A.6, which heavily relies on the semantics of annotations. As a key part of the proof, we show that we can uncompute temporary values by reversing the computation that computed them. Reversing a computation is possible on a QRAM (and supported by most existing quantum languages) by (i) producing the gates that perform this computation and (ii) reversing them.

3.7 EVALUATION OF SILQ

Next, we experimentally compare Silq to other languages. Our comparison focuses on Q#, because (i) it is one of the most widely used quantum programming languages, (ii) we consider it to be more high-level than Cirq or Qiskit, and (iii) the Q# coding contest [51, 52] provides a large collection of Q# implementations we can leverage for our comparison. To check if our findings generalize to other languages, we also compare Silq to Quipper (§3.7.2).

IMPLEMENTATION We implemented a publicly available parser, type-checker, and simulator for Silq as a fork of the PSI probabilistic programming language [62]. Specifically, Silq's AST and type checker are based on PSI, while Silq's simulator is independent of PSI. Our implementation handles all valid Silq code examples in this paper, while rejecting invalid programs. We also provide a development environment for Silq, in the form of a Visual Studio Code extension. [10]

Compared to Silq-Core, Silq supports an imperative fragment (including automatic uncomputation), additional primitives, additional convenience features (e.g., unpacking of tuples), additional types (e.g., arrays), dependent types (which only depend on classical values, as shown in Fig. 3.3), type equivalences (e.g., $!!\tau \equiv !\tau$), subtyping, and type conversions.

10 `https://marketplace.visualstudio.com/items?itemName=eth-sri.vscode-silq`

Table 3.1: Silq compared to Q#.

	Silq			Q#		
	S18	W19	Both	S18	W19	Both
Lines of code	**99**	**168**	**267**	251	242	493
Quantum primitives	**8**	**10**	**10**	12	19	22
Annotations	**2**	3	3	3	6	6
Low-level quantum gates	**14**	**23**	37	33	54	87

3.7.1 *Comparing Silq to Q#*

To compare Silq to Q#, we solved all 28 tasks of Microsoft's Q# Summer 2018 and Winter 2019 [51, 52] coding contest in Silq. We compared the Silq solutions to the Q# reference solutions provided by the language designers [63, 64] (Table 3.1) and the top 10 contestants (App. A.7).

Our results indicate that algorithms expressed in Silq are far more concise compared to the reference solution (−46%) and the average top 10 contestants (−59%). We stress that we specifically selected these baselines to be written by experts in Q# (for reference solutions) or strong programmers well-versed in Q# (for top 10 contestants). We did not count empty lines, comments, import statements, namespace statements, or lines that were unreachable for the method solving the task. This greatly benefits Q#, as it requires various imports.

Because the number of lines of code heavily depends on the available language features, we also counted (i) the number of different quantum primitives, (ii) the number of different annotations in both Q# (`controlled auto`, `adjoint self`, `Controlled`, . . .) and Silq (**`mfree`**, **`qfree`**, **`const`**, **`lifted`**, and !), as well as (iii) the number of low-level quantum circuit gates used to encode all programs in Q# and Silq (for details, see App. A.7).

Our results demonstrate that Silq is not only significantly more concise, but also requires only half as many quantum primitives, annotations, and low-level quantum gates compared to Q#. As a consequence, we believe Silq programs are easier to read and write. In fact, we conjecture that the code of the top 10 contestants was longer than the reference solutions because they had difficulties choosing the right tools out of Q#'s large set of quantum primitives. We further note that Silq is better in abstracting away standard low-level quantum circuit gates: they occur only half as often in Silq.

3.7.2 *Comparing Silq to Quipper*

The language designers of Quipper provide an encoding [53] of the triangle finding algorithm [65, 66]. We encoded this algorithm in Silq and found that again, we need significantly less code (-38%; Quipper: 378 LOC, Silq: 236 LOC). An excerpt of this, on which we achieve even greater reduction (-64%), was already discussed in Fig. 3.2.

The intent of the algorithm in Fig. 3.2 is naturally captured in Silq: it iterates over all `j,k` with $0 \leq j < k < 2^{\text{rbar}}$, and counts how often `ee[tau[j]][tau[k]] && eew[j] && eew[k]` is `true`, where we use quantum indexing into `ee`. In contrast, Quipper's code is cluttered by explicit uncomputation (e.g., of `eedd_k`), custom functions aiding uncomputation (e.g., `.&&.`), and separate initialization and assignment (e.g., `eedd_k`), because Quipper lacks automatic uncomputation.

Similarly to Q#, Quipper offers an abundance of built-in and library functions. It supports 76 basic gates and 8 types of `reverse`, while Silq only provides 10 basic gates and 1 type of `reverse`, without sacrificing expressivity. Some of Quippers overhead is due to double definitions for programming in declarative and imperative style, e.g., it offers both `gate_T` and `gate_T_at` or due to definition of inverse gates, e.g., `gate_T_inv`.

3.7.3 *Further Silq Implementations*

To further illustrate the expressiveness of Silq on interesting quantum algorithms, we provide Silq implementations of (i) Wiesner's quantum money scheme [67], (ii) a naive (unsuccessful) attack on it, and (iii) a recent (successful) attack on it [68] in App. A.7.3.

3.7.4 *Discussion*

Overall, our evaluation indicates that Silq programs are significantly shorter than equivalent programs in other quantum languages, while using only half the number of quantum primitives. In the future, it could be interesting to confirm this evidence that Silq is more user-friendly by performing a thorough usability study.

TABLE 3.2: Comparing Silq to previous quantum languages. Parenthesized features are partially (but not fully) supported.

Language	Type system	Autom. Uncomp.	`const`	`mfree`	`qfree`
QPL [48]	linear[11]	✗	✗	✗	✗
Quantum λ-calc. [69]	affine	✗	✗	✗	✗
Quipper [23]	non-linear	(✓)	✗	✗	✗
ReVerC [59]	non-linear	(✓)	✗	✗	(✗)
QWire [54]	linear	✗	✗	✗	✗
Q# [22]	non-linear	✗	✗	✓	✗
ReQWire [58]	linear	(✗)	(✗)	✗	(✗)
Silq (this work)	linear+	✓	✓	✓	✓

3.8 RELATED WORK

Before Silq, various quantum programming languages aimed to simplify development of quantum algorithms. Table 3.2 shows the key language features of previous languages most related to Silq.

CONST To our knowledge, Silq is the first quantum language to mark variables as constant. We note that for Q#, so-called *immutable* variables can still be modified (unlike **`const`** variables), for example by applying the Hadamard transform **`H`**.

Silq's constant annotation is related to ownership type systems guaranteeing read-only references [70]. As a concrete example, the Rust programming language supports a single mutable borrow and many const borrows [71](§4.2). However, the quantum setting induces additional challenges: only guaranteeing read-only access to variables is insufficient as we must also ensure safe uncomputation. To this end, Silq supports a combination of **`const`** and **`qfree`**.

QFREE To our knowledge, no existing quantum language annotates **`qfree`** functions. ReverC's language fragment contains **`qfree`** functions (e.g., **`X`**), and ReQWire's syntactic conditions cover some **`qfree`** operations, but neither language explicitly introduces or annotates **`qfree`** functions.

MFREE Of the languages in Table 3.2, only Q# can prevent reversing measurement and conditioning measurement (via special annotations). However, as Q# cannot detect implicit measurements, reverse and condi-

11 QPL (i) enforces no-cloning syntactically and (ii) disallows implicitly dropping variables (cp. rule `discard` in Fig. 12)

tionals may still induce unexpected semantics. For other languages, reversal may fail at runtime when reversing measurements, and control may fail at runtime on conditional measurement.

We note that QWire's **`reverse`** returns safe functions, but only when given unitary functions (otherwise, it reports a runtime error by outputting `None`). Thus, it for example cannot reverse **`dup`**, which is linearly isometric but not unitary.

SEMANTICS The semantics of Silq is conceptually inspired by Selinger and Valiron, who describe an operational semantics of a lambda calculus that operates on a separate quantum storage [69]. However, as a key difference, Silq's semantics is more intuitive due to automatic uncomputation.

All other languages in Table 3.2 support semantics in terms of circuits that are dynamically constructed by the program.

3.9 IMPACT

Since its publication, Silq has had significant impact on quantum programming language research. In the following, we highlight some key developments we believe were facilitated by Silq.

AUTOMATIC UNCOMPUTATION Silq has paved the way for more recent advancements in automatic uncomputation. Qunity [24] extends Silq's notion of automatic uncomputation to non-qfree expressions, at the cost of probabilistic errors. Specifically, Qunity develops the relevant theory to show that allowing a program to fail probabilistically enables automatically uncomputing expressions even if they are non-qfree. One advantage of this generalization is that it naturally supports non-qfree implementations of (almost) qfree functions. We note that besides failing probabilistically, one disadvantage of this approach is that the semantics of uncomputing a variable depends on the semantics of the function which computed this variable, while Silq's uncomputation semantics simply drops the uncomputed variable from the program state.

Our own works Unqomp (Chapter 4) and Reqomp [14] automatically synthesize efficient uncomputation, which is one of the key challenges when compiling Silq programs. Qrisp [25] was inspired by Silq and automates uncomputation by using Unqomp.

OTHER LANGUAGE FEATURES Many works reference Silq as a useful resource to demonstrate the benefits of a linear type system [26, 27, 28]—this helps answer the long-standing question of whether linear or non-linear type systems are preferable in quantum programming languages.

Likewise, multiple works refer to Silq to show-case a quantum language which supports classical control flow, thus allowing mixing classical and quantum computation [27, 29, 30].

Finally, Proto-Quipper [72], Qiwi [73], and others [74, 75] credit Silq as being relevant for the design of their language primitives.

COMPLEMENTARY HIGH-LEVEL LANGUAGE FEATURES In line with Silq's goal of raising the abstraction level of quantum languages, Tower [76] introduces quantum primitives for working with random-access memory, allowing the programmer to implement quantum data structures.

Also providing more high-level insights but for a different aspect, Twist [77] introduces language primitives helping to establish that some qubits are unentangled.

Such efforts are complementary to Silq: for example, Twist mentions that it benefits from automatic uncomputation as it can rely on the resulting correctness guarantees [77, §11].

COMPILATION AND INTERNAL REPRESENTATIONS Silq2Qiskit [78] is an effort to compile a fragment of Silq to Qiskit. While it handles some interesting aspects of Silq including control flow and quantum indexing, it unfortunately does not discuss how to compile automatic uncomputation (see also Chapter 4).

More broadly, Silq has highlighted the importance of intermediate representations (IR) that store, optimize, manipulate, and compile high-level quantum programs.

The intermediate representation QIRO [79] is designed to also accommodate Silq, but does not provide a compiler from Silq to QIRO. HQIR [33] is a recent effort with the goal of being sufficiently high-level to represent Silq programs at a suitable level of abstraction, while still bridging the gap to low-level quantum circuits.

VERIFICATION Silq has also been identified as an interesting target for verifying the correctness of quantum programs [46].

The ongoing work SilVer [80] aims to verify Silq programs using the Z3 SMT solver. Other verification tools such as VOQC [43] and SQIR [44] currently do not support Silq, but mention it as an interesting future extension.

3.10 CONCLUSION

We presented Silq, a new high-level statically typed quantum programming language which ensures safe uncomputation. This enables an intuitive semantics that is physically realizable on a QRAM.

Our evaluation shows that quantum algorithms expressed in Silq are significantly more concise and less cluttered compared to their version in previous quantum languages.

4 UNQOMP: SYNTHESIZING UNCOMPUTATION IN QUANTUM CIRCUITS

In this chapter, we present Unqomp, the first procedure to automatically synthesize uncomputation in a given quantum circuit. Unqomp can be readily integrated into popular quantum languages such as Qiskit, allowing the programmer to allocate and use temporary values analogously to classical computation, knowing they will be uncomputed by Unqomp. To allow a more natural integration with popular quantum languages like Qiskit, this chapter reframes the objective of uncomputation in a circuit-focused view, as opposed to the language-focused view provided in Chapter 3.

Our evaluation shows that programs leveraging Unqomp are not only shorter (-19% on average), but also generate more efficient circuits (-71% gates and -19% qubits on average).

4.1 INTRODUCTION

As discussed in Chapter 3, quantum programs often produce temporary values during execution. However, in contrast to classical values, the mere existence of temporary quantum values can lead to unexpected side effects on the remainder of the program state due to the phenomenon of quantum entanglement. Preventing such side effects typically requires resetting temporary quantum values to zero before discarding them, in a process called uncomputation [60].

SYNTHESIZING UNCOMPUTATION This need for uncomputation is a major roadblock preventing programmers from writing correct, efficient, and intuitive quantum programs in circuit description languages like Qiskit.

Such programs construct quantum circuits to be run on a quantum computer. Ideally, uncomputation would be synthesized automatically during circuit construction, allowing the programmer to simply omit it. Unfortunately, existing uncomputation synthesizers are restricted to quantum programs consisting exclusively of classical operations (e.g., [23, 58, 59, 81]).

Silq (Chapter 3) addressed uncomputation in quantum programs by introducing a type system which statically checks that temporary values can be

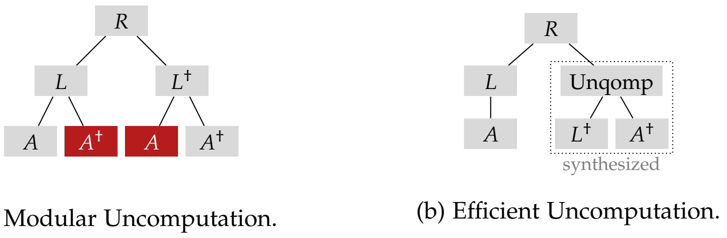


FIGURE 4.1: Manual yet modular uncomputation is inefficient.

uncomputed. However, Silq does not explicitly synthesize uncomputation as it does not include a compiler. Likewise, ReQWire [58] can verify that manually provided uncomputation is safe, but cannot synthesize it.

Consequently, most quantum languages before Unqomp require tedious manual uncomputation by explicitly reversing all operations applied to temporary values, sometimes aided by (unsafe) convenience functions (e.g., `ApplyWith`, `with_computed`, discussed in §4.7). However, this manual approach leads to tension between modularity and efficiency, discussed next.

MODULAR PROGRAMS Generally, writing complex quantum programs requires a modular approach, in particular when developing libraries. Indeed, a quantum library function L typically uncomputes all its internal temporary values, without exposing them to the caller. The programmer can then use L's inverse $L^\dagger$ to uncompute the result of L. If L computes a temporary value using an auxiliary function A, L can in turn leverage $A^\dagger$ to uncompute the result of A.

We visualize a call tree resulting from this modular approach in Fig. 4.1a, where a function R uses the library L and later uncomputes its result using $L^\dagger$, which internally recomputes A. Note that if L did not encapsulate the uncomputation of A, it would have to expose the output of A to be uncomputed by the user of L, thus breaking modularity.

INEFFICIENT CIRCUITS While the above modular construction facilitates correct uncomputation, it often results in inefficient circuits. This is a critical problem, as near-future quantum computers only support a limited number of qubits and are subject to noise limiting the number of gates [82].

In the example of Fig. 4.1a, A is uncomputed in L only to be recomputed again in $L^\dagger$. This redundant work (highlighted as 🟥) increases exponentially with the depth of the call tree, becoming prohibitive for complex programs.

In contrast, Fig. 4.1b shows the call tree of an equivalent but more efficient computation, which avoids recomputing A. Achieving this without exposing the output of A and thus breaking modularity of L's implementation is only possible if uncomputation is synthesized during circuit construction.

SACRIFICING MODULARITY In some cases, programmers sacrifice modularity for efficiency and manually build call trees similar to Fig. 4.1b. However, the resulting code is error-prone and may introduce other sources of inefficiencies (see §4.6).

These downsides are exacerbated by the fact that erroneous uncomputation is particularly hard to detect. For example, programmers often reuse the same physical qubit to first hold temporary value a and later temporary value b, in which case incorrectly uncomputing a may corrupt the computation involving b.

UNQOMP To enable writing modular yet efficient quantum programs, we introduce Unqomp, the first procedure to automatically synthesize uncomputation.

Technically, Unqomp relies on the same fundamental insight as Silq, namely that a temporary value can be safely uncomputed by inverting the operation that computed it, if the original computation can be described classically and depends on values that can be reused for uncomputation. Unfortunately, whether these values are available for uncomputation can depend on the exact order in which operations are applied, even though these operations often commute. To address this challenge, Unqomp operates on *circuit graphs*, a representation of quantum circuits which does not enforce unnecessary ordering constraints among operations.

EVALUATION RESULTS Unqomp is designed such that it can be readily integrated into existing quantum languages currently requiring manual uncomputation (see §4.3).

Our evaluation demonstrates that integrating Unqomp into Qiskit [39] allows writing code that is shorter (19% on average), more modular (preventing bugs existing in current implementations), and often significantly more efficient (71% fewer gates and 19% fewer qubits on average). We reported a set of efficiency issues revealed by our evaluation to the Qiskit developers, who have since addressed them. Even compared to the resulting enhanced version of Qiskit, Unqomp allows for significant improvements (57% for gates and 19% for qubits). Furthermore, when used on purely classical examples, Unqomp significantly outperforms other approaches, saving 40% of gates and 41% of qubits on average when compared to Quipper.

MAIN CONTRIBUTIONS The main contributions of this chapter are:

- Unqomp, a procedure synthesizing automatic uncomputation for quantum circuits (§4.3).
- A formalization of circuit graphs, Unqomp's internal representation of quantum circuits (§4.4).
- A correctness theorem for Unqomp and its proof (§4.5).
- An end-to-end implementation[1] and a thorough evaluation of Unqomp on common quantum algorithms (§4.6).

4.2 PROBLEM STATEMENT

Next, we motivate why uncomputation is critical and formally define the problem statement addressed by Unqomp.

BACKGROUND: QFREE GATES Let us first rephrase the notion of qfree from Chapter 3 in terms of quantum circuits.

Intuitively, a gate U is qfree if it can be expressed on classical bits. More precisely, for a gate U with one[2] control c and target t, U is qfree if its semantics can be described in terms of a function $f\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ as

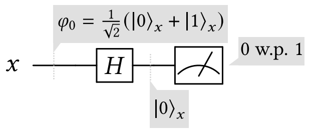
$$|i\rangle_c \, |k\rangle_t \xmapsto{U_{ct}} |i\rangle_c \, |f_i(k)\rangle_t \,, \tag{4.1}$$

writing $f_i(k)$ for $f(i,k)$ and $a \xmapsto{U} b$ when $U(a) = b$. For a gate U with no controls, the above definition simplifies to

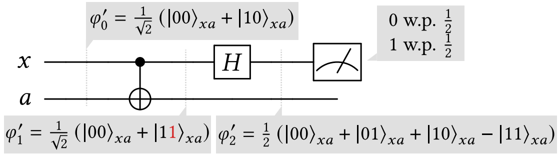
$$|k\rangle_t \xmapsto{U} |f(k)\rangle_t \,, \tag{4.2}$$

for some $f\colon \{0,1\} \to \{0,1\}$.

1 `https://github.com/eth-sri/Unqomp/tree/pldi2021`
2 The definition generalizes naturally to multiple controls.

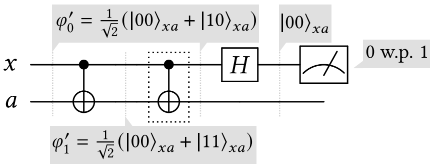


(a) Example circuit.



$\varphi_0' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |10\rangle_{xa})$
0 w.p. $\frac{1}{2}$
1 w.p. $\frac{1}{2}$
x
a
H
$\varphi_1' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |11\rangle_{xa})$
$\varphi_2' = \frac{1}{2}(|00\rangle_{xa} + |01\rangle_{xa} + |10\rangle_{xa} - |11\rangle_{xa})$

(b) Example circuit storing a temporary value into a.



$\varphi_0' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |10\rangle_{xa})$
$|00\rangle_{xa}$
0 w.p. 1
x
a
H
$\varphi_1' = \frac{1}{\sqrt{2}}(|00\rangle_{xa} + |11\rangle_{xa})$

(c) Uncomputing temporary value.

FIGURE 4.2: Uncomputation in an example quantum circuit, inspired by [83].

Examples of qfree gates include the identity I with semantics $I\,|a\rangle = |g(a)\rangle$ where g is the identity and CX with semantics $CX\,|a\rangle\,|b\rangle = |a\rangle\,|f_a(b)\rangle$ for $f_a(b) := a \oplus b$. In contrast, the Hadamard transform H is not qfree.

In this chapter, we only consider gates with exactly one target and zero or more controls. For example, CX is controlled by its first qubit (•) and targets the second qubit ($\oplus$). The gate H targets only one qubit and has no controls. As single qubit gates and CX are universal for quantum computation, considering only one target is not a restriction [47, §4.5.2].

EFFECT OF TEMPORARY VALUES To observe the effect of temporary values that are not uncomputed in terms of quantum circuits, first consider Fig. 4.2a, which applies H to x in initial state φ_0 and measures x, yielding 0 with probability 1. This circuit is extended to Fig. 4.2b, which copies[3] x into an additional temporary qubit a (called *ancilla*). To this end, Fig. 4.2b

3 Note that copying using CX does not violate the *no-cloning* theorem.

initializes a to $|0\rangle$ yielding $\varphi_0' = \varphi_0 \otimes |0\rangle_a$, and applies CX to flip the value of a if x is one. The resulting state φ_1' highlights the flipped value in red (see Fig. 4.2b). At a high level, because x is not modified by CX (x is a control), the two circuits should not differ in their effect on x. However, measuring x yields different results, as we mathematically demonstrate in Fig. 4.2b: the measurement now returns 0 or 1 with probability $\frac{1}{2}$. This difference is caused by the existence of a, which is entangled with x due to the CX gate (see φ_1').

We note that in this toy example, copying x into a is pointless, as the copy is never used. However, we can easily imagine this copy being required in the remainder of the computation (not shown). This is a common pattern in practice, where ancillae store intermediate computation results.

UNCOMPUTATION If we want to avoid the side effect of a onto x, we need to disentangle a from the remainder of the state before measuring x. This can be achieved by uncomputing a, which resets a to its initial, unentangled state $|0\rangle$. Mathematically, this amounts to transforming φ_2' to

$$\tfrac{1}{2}\left(|00\rangle_{xa} + |00\rangle_{xa} + |10\rangle_{xa} - |10\rangle_{xa}\right) = |00\rangle_{xa}\,. \tag{4.3}$$

To this end, we can insert another CX gate (the self-inverse of CX) as shown in Fig. 4.2c (dashed box). This gate reverts the original CX gate, thus uncomputing ancilla a. Then, the result of the measurement is again 0 with probability 1, as expected.

GOAL: SYNTHESIZING UNCOMPUTATION The goal of Unqomp is to automate the process of uncomputation. Given a (quantum) circuit C and a list of ancilla qubits A, our goal is to create a new circuit with the same effect as C, except that ancilla qubits are brought back to $|0\rangle$, as in Eq. (4.3).

The procedure Unqomp presented in this chapter achieves this goal, formalized in Thm. 4.2.1 below. Thm. 4.2.1 represents circuit C as a *circuit graph* G (discussed shortly), and describes the effect of G on an initial state by the semantics $[\![G]\!]$.

Theorem 4.2.1 (Correctness). *Let* $\textsc{Unqomp}(G, A) = \mathcal{G}$ *for circuit graph* G *with* n *qubits of which* m *are ancilla qubits. Without loss of generality, assume that those ancillae* $A = \left(a^{(1)}, \ldots, a^{(m)}\right)$ *are the first* m *qubits of* G*. If*

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k\in\{0,1\}^m} \gamma_k\,|k\rangle_A \qquad \otimes \phi_k, \textit{ then} \tag{4.4}$$

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![\mathcal{G}]\!]} \sum_{k\in\{0,1\}^m} \gamma_k\,|0\cdots0\rangle_A \otimes \phi_k. \tag{4.5}$$

Because $\mathcal{G}$ resets ancillae to state $|0\cdots0\rangle$, they are unentangled with the remainder of the state, and can hence be safely discarded without unexpected side effects.

We note that Thm. 4.2.1 implicitly assumes that G contains no measurement. In particular, in Fig. 4.2, G would correspond to Fig. 4.2b without the measurement, and $\mathcal{G}$ would correspond to Fig. 4.2c without the measurement.

4.3 OVERVIEW

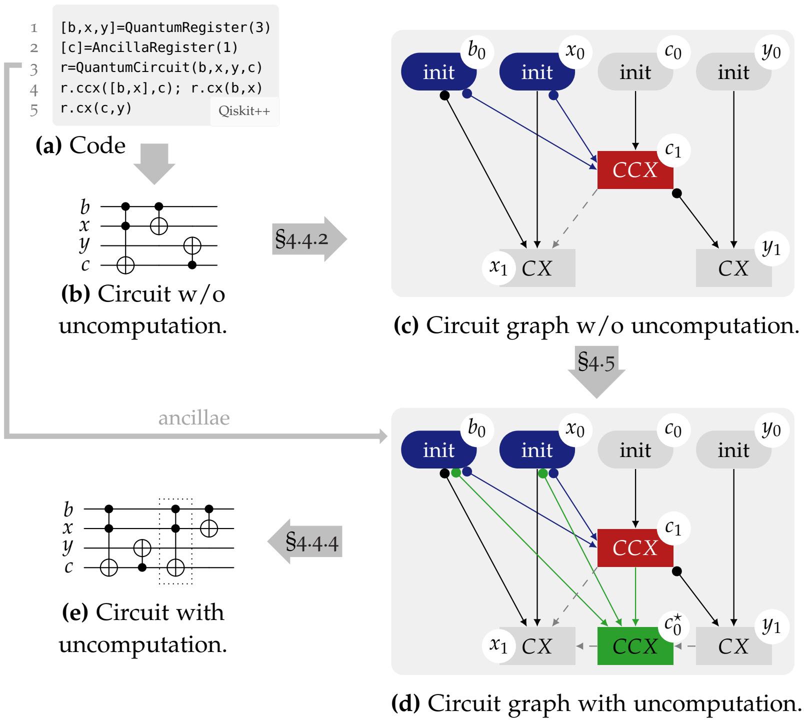
We now provide an overview of Unqomp, following Fig. 4.3.

UNQOMP FOR CIRCUIT-BASED LANGUAGES At a high level, Fig. 4.3 shows how Unqomp can be readily integrated into circuit-based programming languages such as Qiskit [39] (in the example), Cirq [57], Q# [22], or Quipper [23]. Such languages describe quantum programs (see Fig. 4.3a) which are then compiled to quantum circuits (see Fig. 4.3b).

Relying on Unqomp, we can extend Qiskit to Qiskit++, which allows declaring ancilla qubits at allocation time (see Line 2 in Fig. 4.3a). Qiskit++ (i) constructs the circuit without uncomputation, (ii) transforms the circuit to a circuit graph (§4.4.2), (iii) runs Unqomp to uncompute the ancilla qubits (§4.5), and (iv) compiles the result back to a circuit (§4.4.4). We note that the resulting circuit may be subject to post-processing such as decomposing the circuit into universal gates.

Next, we walk through the steps in Fig. 4.3 in more detail.

ADDER CIRCUIT WITHOUT UNCOMPUTATION The circuit in Fig. 4.3b (constructed from Fig. 4.3a) is an adder circuit which takes as input two qubits x and y representing the binary encoding of the number $x + 2y$. The circuit adds to this number the value of qubit b using a temporary carry qubit c.



```
[b,x,y]=QuantumRegister(3)
[c]=AncillaRegister(1)
r=QuantumCircuit(b,x,y,c)
r.ccx([b,x],c); r.cx(b,x)
r.cx(c,y)
```

(a) Code

(b) Circuit w/o uncomputation.

(c) Circuit graph w/o uncomputation.

(d) Circuit graph with uncomputation.

(e) Circuit with uncomputation.

Figure 4.3: Overview of Unqomp: A circuit incrementing $x + 2y$ by b, with carry c.

First, the circuit computes the value of c, which is initialized with $|0\rangle$, using the CCX gate (a natural generalization of the CX gate to two controls) to change c to 1 iff both x and b are 1. Next, the circuit flips the value of x if b is 1, correctly determining the least significant qubit of the result. Finally, the circuit flips the value of y if the carry c is 1. Note that this circuit does not perform uncomputation of c.

FINDING THE UNCOMPUTATION POSITION In order to uncompute c, we have to revert the CCX gate computing c. As a naive attempt, we could try to append the inverse gate of CCX (which is CCX again) at the end of the circuit in Fig. 4.3b. Unfortunately, this does not correctly uncompute c: the computation of c is controlled by x, whose value may change by the end of the circuit due to the CX gate targeting x.

A key challenge of uncomputation is therefore finding the position in the circuit to insert the inverse gate $g^\dagger$ uncomputing a gate g. This position must be (i) after all gates involving the computed value (here, after the CX gate controlled by c), but (ii) before any other gates targeting any qubit involved in $g^\dagger$ (here, before the CX gate targeting x). In Fig. 4.3b, satisfying (i–ii) is only possible when reordering the two CX gates. In Fig. 4.3e, we have reordered the CX gates, and inserted the uncomputation gate CCX in-between.

Crucially, reordering the CX gates in this example yields an equivalent circuit with the same semantics—on the same input state, the circuit produces the same output state.

4.3.1 *Circuit Graphs*

To avoid the need for gate reorderings, we introduce an alternative circuit representation called *circuit graph*, which abstracts different gate orderings with the same semantics.

Fig. 4.3c shows the circuit graph G corresponding to Fig. 4.3b.

NODES AND EDGES For every qubit in the circuit, the circuit graph contains an *init node* indicating the circuit's input (here: b_0, x_0, y_0, and c_0). The remaining nodes represent gates. For example, c_1 represents the CCX gate from Fig. 4.3b, which targets c (indicated by a *target edge* $\rightarrow$) and is controlled by b and x (indicated by *control edges* $\bullet\!\!\rightarrow$).

ANTI-DEPENDENCY EDGES Any linearization of gate nodes in G can be interpreted as a quantum circuit applying the corresponding gates in the specified order. In order to ensure that all such circuits have equivalent semantics, we introduce additional ordering constraints using *anti-dependency edges* $\dashrightarrow$.

For instance, the anti-dependency edge $c_1 \dashrightarrow x_1$ in Fig. 4.3c indicates that the CCX gate must be applied before the CX gate targeting x. This is critical, because the former uses the value of its control qubit x, which is modified by the latter.

Note that G does not enforce an ordering between gate nodes x_1 and y_1, implicitly accounting for the fact that we can swap these without affecting the semantics of the circuit.

4.3.2 *Uncomputation*

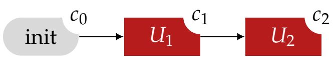
We now show how Unqomp leverages the circuit graph in Fig. 4.3c to uncompute the carry qubit c, yielding Fig. 4.3d.

ONE STEP OF UNQOMP First, Unqomp determines the last gate targeting c, which is the CCX gate in c_1 (🟥). Second, Unqomp checks that CCX is qfree (otherwise, it returns an error). We discuss this necessity in §4.5.2.

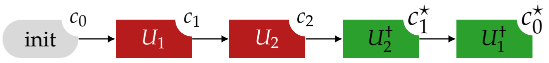
Next, Unqomp inserts a node applying the inverse of CCX (which is again CCX) into the graph (🟩). We refer to this new node as $c_0^\star$ because it resets the state of c to its state after c_0. We control $c_0^\star$ by the same controls as c_1 (🟦).

Finally, Unqomp checks that the resulting graph does not contain any cycles (otherwise, it would return an error). This check takes into account anti-dependency edges, which are also updated in Fig. 4.3d. In particular, edge $y_1 \dashrightarrow c_0^\star$ ensures that the uncomputation node $c_0^\star$ comes after gate node y_1 controlled by c_1, while edge $c_0^\star \dashrightarrow x_1$ ensures that uncomputation node $c_0^\star$ comes before gate node x_1 targeting the control x_0 of $c_0^\star$.

MULTIPLE UNCOMPUTATION STEPS If more than one gate was applied to c, Unqomp would execute multiple uncomputation steps as described above, one for each gate targeting c. For instance, assume that two gates U_1, U_2 are applied to c. The circuit graph then contains three nodes for this qubit:



Unqomp steps through all gates applied to ancilla qubits, in reverse order. To process c_2, it checks that U_2 is qfree, inserts $c_1^\star$ controlled by the same nodes as c_2, and links it to the latest node operating on c, yielding $c_2 \to c_1^\star$. Next, Unqomp processes c_1, checking that it is qfree, inserting $c_0^\star$ controlled by the same nodes as c_1, and adding an edge from the latest node operating on c, yielding $c_1^\star \to c_0^\star$ as shown below:



4.4 CIRCUIT GRAPHS

We now provide a more formal introduction to circuit graphs. In particular, we discuss their motivation and definition (§4.4.1), show how a circuit is transformed to its graph representation (§4.4.2), provide semantics for circuit graphs (§4.4.3), and show how a circuit graph is compiled back to a circuit (§4.4.4).

4.4.1 *Motivation and Definition*

We start by motivating the need for circuit graphs and presenting their formal definition.

GATE ORDERING As discussed in §4.3, quantum circuits typically enforce an unnecessarily restrictive order on their gates. In contrast, circuit graphs abstract away irrelevant ordering constraints: instead of enforcing a total order, circuit graphs use edges to record only the relevant ordering constraints between gate nodes, inducing a partial order on gates.

Specifically, circuit graphs reflect the fact that any two adjacent gates can be reordered, unless the qubit targeted by one of them is involved in the other gate (as a control or as a target). We illustrate this in Fig. 4.4, where we show two equivalent circuits and their shared circuit graph representation. In particular, the circuit graph does not contain an edge between gate nodes U and V, allowing them to be ordered arbitrarily. Formally, this equivalence can be derived from the properties of control qubits (see §2.2).

CIRCUIT GRAPHS A circuit graph is a directed acyclic graph $G = (V, E)$. Its nodes $V = V_{\text{init}} \cup V_{\text{gates}}$ consist of init nodes V_{init} and gate nodes V_{gates}. The set V_{init} contains an init node for each qubit accessed during the

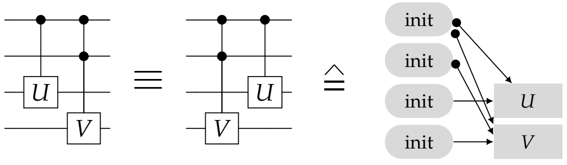


FIGURE 4.4: Valid gate reordering.

computation. For $v \in V_{\text{init}}$, we define qbit(v) to be the qubit modeled by v. The set V_{gates} contains one node for each gate in the circuit. We define gate(v) to be the gate represented by $v \in V_{\text{gates}}$, and qbit(v) to be the qubit targeted by gate(v). For example, in Fig. 4.3c, the init node b_0 models the qubit b in Fig. 4.3b, and gate node c_1 models the CCX gate targeting c.

The edges E are divided into target (visualized as $\rightarrow$), control ($\bullet\!\!\rightarrow$), and anti-dependency edges ($\dashrightarrow$). Target edges represent input-output relationships, control edges represent additional dependencies, and anti-dependency edges specify implicit ordering constraints on gate nodes. In general, anti-dependency edges can always be reconstructed from target and control edges. Specifically, circuit graphs contain an anti-dependency edge $t \dashrightarrow c$ whenever there exists a node c' such that $c' \bullet\!\!\rightarrow t$ and $c' \rightarrow c$. This anti-dependency edge models the ordering constraint that t must be applied before c.

VALID CIRCUIT GRAPHS A circuit graph is valid if it represents an actual quantum circuit. More precisely, $G = (V, E)$ is valid iff (i) its init nodes have no incoming target edge while gate nodes have exactly one, (ii) all its nodes have at most one outgoing target edge, (iii) its anti-dependency edges can be reconstructed from its control and target edges according to the rule discussed above, (iv) the number of incoming control edges of each gate node $v \in V_{\text{gates}}$ equals the number of controls of gate(v), and (v) G is acyclic. Consequently, the target edges should form disjoint paths starting at init nodes.

In the following, we only consider valid circuit graphs. All operations discussed in this chapter preserve validity.

NAMING CONVENTION We usually respect the following naming convention: for qubit a, a_0 is the init node and a_i the i^{th} gate node targeting a. For example, in Fig. 4.3c, node c_1 is the first gate targeting c. Further, we refer to gate nodes inserted by Unqomp as $a_i^\star$, indicating that $a_i^\star$ restores the state of qubit a after node a_i.

4.4.2 *From Circuits to Circuit Graphs*

We now describe how to transform a given circuit into its circuit graph representation.

BUILDING A CIRCUIT GRAPH To construct a circuit graph from a circuit, we first create an init node for each qubit.

Then, we process the gates in order. When processing a gate U, we add a fresh node u representing U to V_{gates}. To determine the incoming target edge at u, let q be the qubit targeted by U. We then introduce a target edge $v \rightarrow u$, where v is the node corresponding to the latest gate targeting q, or the init node for q if we are processing the first gate targeting q. Similarly, to determine incoming control edges at u, we consider each control qubit c of the gate. We introduce a control edge $v \bullet\!\!\rightarrow u$, where v is the node corresponding to the latest gate targeting c, or the init node for c.

Finally, we introduce anti-dependency edges based on the inserted target and control edges (see §4.4.1).

EXAMPLE When processing the first CX gate in Fig. 4.3b, we introduce gate node x_1 and add edge $x_0 \rightarrow x_1$. Because this gate is controlled by qubit b represented by init node b_0, we further add edge $b_0 \bullet\!\!\rightarrow x_1$ to the graph. The anti-dependency edge $c_1 \dashrightarrow x_1$ exists due to $x_0 \bullet\!\!\rightarrow c_1$ and $x_0 \rightarrow x_1$.

4.4.3 *Circuit Graph Semantics*

We now define the semantics of circuit graphs. We first define the semantics of individual nodes, and then extend this semantics to whole circuit graphs.

STATES The init nodes $V_{\text{init}} = \{v_1, \ldots, v_n\}$ of a circuit graph specify the qubits of the system. The state of this system is in $\mathcal{H}_{q_1,\ldots,q_n}(\{0,1\}^n)$, where $q_i := \text{qbit}(v_i)$.

SEMANTICS OF GATE NODES The semantics $[\![v]\!]$ of a gate node $v \in V_{\text{gates}}$ is defined according to the semantics of $\text{gate}(v)$, where the target and control edges ending at v determine the involved target and control qubits, respectively. For example, the semantics of x_1 in Fig. 4.3c is $[\![x_1]\!] = CX_{bx}$.

SEMANTICS OF CIRCUIT GRAPHS The semantics of a given circuit graph $G = (V, E)$ is the composition of the semantics of its gate nodes, according to the partial order specified by its edges. More precisely, we first select an arbitrary linearization $\mathcal{L}(G)$ of the gate nodes $V_{\text{gates}} \subseteq V$ consistent with the partial order induced by E. Then, we compose the node semantics in this order by function composition. Importantly, the resulting semantics is independent of the choice of $\mathcal{L}(G)$.

For example, for the circuit graph G in Fig. 4.3b, we can select $\mathcal{L}(G) = (c_1, x_1, y_1)$, yielding the semantics:

$$[\![G]\!] = [\![y_1]\!] \circ [\![x_1]\!] \circ [\![c_1]\!] \,.$$

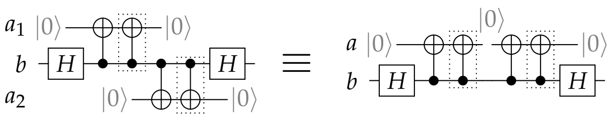
4.4.4 *Compilation to Circuit*

We now discuss how circuit graphs are compiled to circuits. As visualized in Fig. 4.3, this step is applied after introducing uncomputation in the circuit graph (discussed in §4.5). As such, the steps performed here are not part of the transformation covered by Thm. 4.2.1, which for instance assumes a constant number of ancillae.

CCX GATES OPTIMIZATION Before the actual compilation, we run a simple optimization pass suggested in [84, §6]: we replace every CCX gate targeting an ancilla qubit by a more efficient Margolus gate, which has the same semantics as CCX, except that it maps $|111\rangle$ to $-|110\rangle$ instead of $|110\rangle$. This so-called phase change does not affect the semantics of the whole circuit, as Unqomp ensures that all replaced CCX gates are paired with a gate uncomputing it, which, when also replaced, reverts the phase change.

This optimization is already selectively leveraged by experts. For instance, Qiskit uses Margolus gates in its library implementation of the MCX gate. In our case, by leveraging the uncomputation information available in the circuit graph, we can effortlessly extend this optimization to *all* uncomputed CCX gates in a circuit.

LINEARIZATION To compile $G = (V, E)$ to a circuit after applying the optimization above, we first prepare a wire for each init node and then select a linearization $\mathcal{L}(G)$ yielding $(v_1, \ldots, v_n)$. Next, we insert the gates $\text{gate}(v_1), \ldots, \text{gate}(v_n)$ into the circuit according to this linearization, deter-



(a) Naive compilation. **(b)** Efficient compilation.

FIGURE 4.5: Reusing qubits.

mining the target and control qubits involved in gate(v_i) by the incoming target and control edges at v_i.

For example, compiling Fig. 4.3d using the linearization $(c_1, y_1, c_0^\star, x_1)$ yields the circuit in Fig. 4.3e.

REUSING QUBITS Unfortunately, this strategy allocates a wire for each ancilla qubit in G. This is unnecessarily wasteful, as a wire holding a correctly uncomputed ancilla is in unentangled state $|0\rangle$ and can be safely reused to hold another ancilla without introducing unexpected side effects.

For example, Fig. 4.5a shows the result of uncomputing qubits a_1 and a_2 in a toy circuit, where dotted boxes indicate gates inserted by Unqomp. The resulting circuit requires 3 wires in total. In contrast, Fig. 4.5b shows a more efficient compilation which only requires 2 wires by reusing the same wire to hold both ancillae. This is possible because the lifetimes of the two ancillae do not overlap.

FINAL NODES To track the lifetime of ancillae, we introduce *final* nodes, which mark the end of the computation involving a qubit. Specifically, for every node $u \in V$ with no outgoing target edge, we add to G a final node v, a target edge $u \to v$, and any induced anti-dependency edges. Additionally, we extend the linearization to $\mathcal{L}^+(G)$, which includes init, gate, and final nodes, respecting the partial order induced by E. As a result, in $\mathcal{L}^+(G)$, the qubit corresponding to a final node is not involved in any gate at any later position.

GREEDY ANCILLA ALLOCATION To reduce the number of wires, we employ a simple but effective greedy strategy reusing wires whenever possible. Specifically, we process the nodes in V in the order $\mathcal{L}^+(G)$. The init and final nodes allow us to track the start and end of a qubit's lifetime. Upon visiting an ancilla init node, we greedily try to allocate it on a wire that holds an ancilla qubit which is no longer alive and thus must have been uncomputed by Unqomp. If no such wire exists, we allocate a fresh wire. This approach is an instantiation of linear scan register allocation [85], which is provably optimal for a fixed linearization [86, §17]. As finding an optimal linearization is computationally expensive, we next introduce a greedy heuristic to select a linearization that performs well empirically (see §4.6).

LINEARIZATION HEURISTIC To select a linearization $\mathcal{L}^+(G)$, we slightly modified Kahn's algorithm [87]—a standard algorithm which creates a linear order of G's vertices by iteratively removing vertices that have no incoming edges.

Specifically, when selecting the next node to be removed from G, we de-prioritize ancilla init nodes and only select them if no other choice exists. As a consequence, $\mathcal{L}^+(G)$ greedily completes computations involving ancillae before allocating new ancillae, whenever possible.

DISCUSSION: GRAPH COLORING Unfortunately, graph coloring allocation (e.g., [88, §8]) cannot be readily adapted for circuit graphs as they enforce fewer ordering constraints between operations than control flow graphs, meaning that we cannot definitively determine if two given ancillae are live simultaneously. Hence, recording pairwise conflicts is insufficient: even if each pair of ancillae (a, b), (b, c) and (c, a) can be allocated on the same wire, allocating all three of them on the same wire may be impossible.

4.5 SYNTHESIZING UNCOMPUTATION

In this section, we formalize Unqomp (§4.5.1), discuss the conditions it checks to ensure uncomputation is possible (§4.5.2), and sketch Unqomp's correctness proof (§4.5.3).

4.5.1 *Unqomp*

Algorithm 1 formalizes Unqomp, which takes a circuit graph G and a list of ancilla qubits to be uncomputed (Line 1), and returns, if possible, G extended by gate nodes uncomputing the ancilla qubits. The following discussion complements our informal presentation of Unqomp on the example in §4.3.

Algorithm 1 Unqomp: Synthesizing uncomputation.

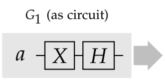
```
 1: procedure UNQOMP($G$, $a^{(1)}, \dots, a^{(n)}$: qubits)
 2:     $(v_1, \dots, v_n) \leftarrow \mathcal{L}(G)$                          ▷ linearization of gate nodes
 3:     for all $v$ in $(v_n, \dots, v_1)$ do                              ▷ iterate in reverse order
 4:         if qbit$(v) \in \{a^{(1)}, \dots, a^{(n)}\}$ then               ▷ gates operates on ancilla
 5:             $G \leftarrow$ UNCOMPUTESTEP($G$, $v$)
        return $G$
 6: procedure UNCOMPUTESTEP($G$, $a_n$: gate node)                           ▷ $G = (V, E)$
 7:     assert gate$(a_n)$ is qfree
 8:     $a_n^\star \leftarrow$ last gate node targeting qbit$(a_n)$
 9:     $ctrls \leftarrow \{c \in V \mid c \bullet\!\!\rightarrow a_n \in E\}$                          ▷ controls of $a_n$
10:     for all $c \in ctrls$ do                      ▷ in $ctrls$, replace $c$ by $c^\star$ wherever possible
11:         if $c^\star \in V$ then $ctrls \leftarrow ctrls \setminus \{c\} \cup \{c^\star\}$
12:     $a_{n-1}^\star \leftarrow$ INVERSE$(a_n)$                          ▷ fresh node with inverse gate
13:     $E_u \leftarrow \{c \bullet\!\!\rightarrow a_{n-1}^\star \mid c \in ctrls\} \cup \{a_n^\star \rightarrow a_{n-1}^\star\}$          ▷ fresh edges
14:     $E_a \leftarrow \{v \dashrightarrow a_{n-1}^\star \mid a_n^\star \bullet\!\!\rightarrow v \in E, v \in V\} \cup$             ▷ anti-dependencies
15:             $\{a_{n-1}^\star \dashrightarrow v \mid c \rightarrow v \in E, c \in ctrls\}$
16:     $G_u \leftarrow \left(V \cup \{a_{n-1}^\star\}, E \cup E_u \cup E_a\right)$                ▷ adding uncomputation
17:     assert $G_u$ has no cycles
18:     return $G_u$
```

ALL STEPS To uncompute a given list of ancillae on a given circuit graph $G = (V, E)$, Unqomp iterates over a linearization $\mathcal{L}(G)$ in reverse order (Lines 3–5), introducing an uncomputation step for every gate node targeting an ancilla.

SINGLE STEP The core of Unqomp is procedure UNCOMPUTESTEP (Line 6), which takes a circuit graph $G = (V, E)$ and a gate node a_n to be uncomputed. It first checks that the gate of a_n is qfree (Line 7). It then determines the last gate node $a_n^\star$ targeting qbit(a_n) (Line 8), which restores the state of qubit a after a_n and will be the target of the inserted uncomputation. We note that if this is the first uncomputation step on qubit a, then $a_n^\star = a_n$, as in our overview example (Fig. 4.3). Otherwise, $a_n^\star$ was inserted by Unqomp in a previous step.

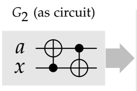
In Line 9, UNCOMPUTESTEP determines the nodes controlling a_n, which are required for controlling the uncomputation. Note that some of those control nodes c may have already been uncomputed by a previous step of Unqomp. In this case, as c and $c^\star$ can be used interchangeably, UNCOMPUTESTEP replaces the former by the latter whenever possible (Line 11). This is helpful as using $c^\star$ is more likely to result in a cycle-free graph than using c (see App. B.2).

Next, UNCOMPUTESTEP constructs the gate node $a_{n-1}^\star$ and edges E_u to be inserted into G. Specifically, the gate node $a_{n-1}^\star$ applies the inverse gate of



$$|0\rangle_a \xmapsto{[\![G_1]\!]} \tfrac{1}{\sqrt{2}}\left(|0\rangle_a - |1\rangle_a\right)$$
$$|0\rangle_a \xmapsto{[\![\mathcal{G}_1]\!]} \tfrac{1}{\sqrt{2}}\left(|0\rangle_a - |0\rangle_a\right) = 0$$

(a) Ancilla a is modified by the non-qfree gate H.



$$\tfrac{1}{\sqrt{2}}\left(|00\rangle_{ax} - |01\rangle_{ax}\right) \xmapsto{[\![G_2]\!]} \tfrac{1}{\sqrt{2}}\left(|00\rangle_{ax} - |10\rangle_{ax}\right)$$
$$\tfrac{1}{\sqrt{2}}\left(|00\rangle_{ax} - |01\rangle_{ax}\right) \xmapsto{[\![\mathcal{G}_2]\!]} \tfrac{1}{\sqrt{2}}\left(|00\rangle_{ax} - |00\rangle_{ax}\right) = 0$$

(b) Uncomputing ancilla a would result in a cyclic dependency.

FIGURE 4.6: Ancilla qubits that cannot be uncomputed.

a_n (Line 12), targets $a_n^\star$, and is controlled by *ctrls* (Line 13). UNCOMPUTESTEP further generates the anti-dependency edges induced by the new edges (Line 14), and constructs the new graph G_u by inserting all new gates and edges into G (Line 16).

Finally, UNCOMPUTESTEP reports an error if G_u contains a cycle (Line 17), and returns G_u otherwise (Line 18).

4.5.2 *Incompleteness*

Unfortunately, uncomputation is mathematically impossible in some cases. Fig. 4.6 exemplifies the two fundamental reasons for this. In both examples, satisfying Thm. 4.2.1 would require constructing a circuit graph $\mathcal{G}$ which produces the invalid state $0 = \sum_{k \in \{0,1\}^n} 0 \cdot |k\rangle$. As no quantum circuit can produce this state, uncomputation is impossible in these cases, forcing Unqomp to return an error.

NON-QFREE In Fig. 4.6a, the underlying problem is that gate H applied to ancilla a is not qfree and therefore mixes basis states: it turns the basis state $|1\rangle_a$ (the result of applying X to the initial state $|0\rangle_a$) into superposition $\frac{1}{\sqrt{2}}\left(|0\rangle_a - |1\rangle_a\right)$. Replacing $|1\rangle_a$ by $|0\rangle_a$ in this state (as required by Thm. 4.2.1) then yields the invalid state 0. Therefore, Unqomp only uncomputes qfree gates, as asserted in Line 7 of Algorithm 1.

CYCLIC DEPENDENCY Fig. 4.6b demonstrates that even for circuits containing only qfree gates, uncomputation may not be possible. The underlying problem in the example is that inserting an uncomputation gate would result in a cyclic dependency: the uncomputation node for a would have to be (i) before the CX gate targeting x, as it uses the initial value of x, but also (ii) after it, as this gate uses the updated value of a. Therefore, Unqomp asserts that the generated circuit graph contains no cycles (Line 17 in Algorithm 1).

CONSERVATIVE CRITERIA We note that Unqomp may return an error even though uncomputation would be possible in principle, as the criteria it checks (Lines 7 and 17 in Algorithm 1) are conservative. For example, applying gate H to an ancilla twice has no effect on its state (as H is self-inverse), but triggers an error in Unqomp as H is not qfree. In such rare cases, the programmer may revert to manual uncomputation, at the costs discussed in Fig. 4.1.

We note that this is not a concern in practice: in our evaluation (§4.6), Unqomp was able to uncompute all temporary values, except when they involved temporary changes of controls. This problem appeared for MCX gates with negated controls, present in two of our examples (Adder and WeightedAdder, see §4.6). To resolve this issue, we ensured that Unqomp treats these problematic gates as atomic qfree gates.

4.5.3 *Correctness of Unqomp*

Next, we discuss the key insights in our proof of Thm. 4.2.1.

Theorem 4.2.1 (Correctness). *Let* $\text{UNQOMP}(G, A) = \mathcal{G}$ *for circuit graph* G *with* n *qubits of which* m *are ancilla qubits. Without loss of generality, assume that those ancillae* $A = \left(a^{(1)}, \ldots, a^{(m)}\right)$ *are the first* m *qubits of* G. *If*

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k\in\{0,1\}^m} \gamma_k \, |k\rangle_A \qquad \otimes \phi_k, \textit{ then} \tag{4.4}$$

$$|0\cdots 0\rangle_A \otimes \varphi \xmapsto{[\![\mathcal{G}]\!]} \sum_{k\in\{0,1\}^m} \gamma_k \, |0\cdots 0\rangle_A \otimes \phi_k. \tag{4.5}$$

OUTLINE We first prove Thm. 4.2.1 in a restricted setting, where (i) there is only a single ancilla a targeted by a single gate U, (ii) the controls c of U are in a basis state $|i\rangle$, and (iii) the gate U occurs first in the computation,

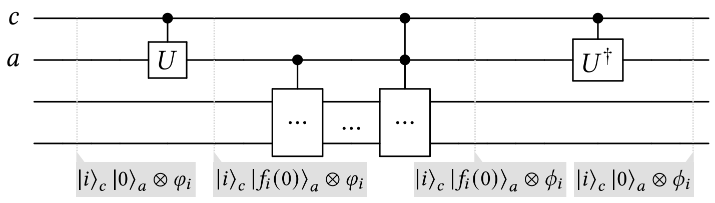

c
a
U
$U^\dagger$
$|i\rangle_c\,|0\rangle_a \otimes \varphi_i$
$|i\rangle_c\,|f_i(0)\rangle_a \otimes \varphi_i$
$|i\rangle_c\,|f_i(0)\rangle_a \otimes \phi_i$
$|i\rangle_c\,|0\rangle_a \otimes \phi_i$

FIGURE 4.7: Effect of uncomputation on quantum state.

while (iv) the gate $U^\dagger$ uncomputing U occurs last. Then, we discuss how our full proof avoids these restrictions.

PROOF SKETCH The key insight of our proof is that if an ancilla is computed using a qfree gate U, it can be uncomputed as long as its controls are still available.

Fig. 4.7 shows how the qfree gate U and its uncomputation affect the quantum state. First, the effect of U follows Eq. (4.1) in §4.2, updating qubit a to $|f_i(0)\rangle$. Then, any remaining gates preserve both a (as U is the only gate targeting a) and c (as any gate targeting the control c of $U^\dagger$ must be after $U^\dagger$ due to anti-dependency edges). Finally, applying $U^\dagger$ restores the state of ancilla qubit a to $|f_i^{-1}(f_i(0))\rangle = |0\rangle$.

This concludes our proof as the first, second to last, and last states in Fig. 4.7 correspond to the left hand side of Eq. (4.4), the right hand side of Eq. (4.4), and the right hand side of Eq. (4.5), respectively.

FULL PROOF We provide a full proof in App. B.2. It (i) handles multiple ancillae and multiple gates by induction on the number of gate nodes in G, (ii) naturally extends to controls in superposition, (iii) accounts for gates before U and after $U^\dagger$, and (iv) takes into account that nodes c and $c^\star$ can be used interchangeably (see Line 11 in Algorithm 1).

4.6 EVALUATION

We now present an extensive experimental evaluation of Unqomp on common quantum algorithms.

IMPLEMENTATION We implemented our approach as a language extension called Qiskit++, which integrates Unqomp into Qiskit as visualized in Fig. 4.3. Qiskit++ allows annotating ancilla qubits in a Qiskit program at allocation time (e.g., see Line 2 in Fig. 4.3a) and uncomputes these automatically. Like Qiskit, Qiskit++ is written in Python.

RESEARCH QUESTIONS Our evaluation addresses the following research questions, where Q1 and Q2 analyze the input Qiskit++ code and Q3 evaluates the compilation result.

Q1 Code Length: Does Unqomp reduce the amount of code, compared to manual uncomputation?

Q2 Modularity: Does Unqomp allow writing more modular and hence less complex code?

Q3 Efficiency: Does Unqomp yield more efficient circuits in terms of gates and qubits?

4.6.1 *Evaluated Algorithms*

To address these questions, we evaluated Unqomp on 10 quantum algorithms shown in Table 4.1. We used the implementations from Qiskit 0.22.0 [89] and Cirq v0.9.1 [90], where we re-implemented Cirq examples in Qiskit.

ALGORITHMS Each quantum algorithm is represented by a Python function which, given some parameters such as the input size, constructs a circuit implementing the algorithm. For example, the MCX (multi-controlled NOT) function accepts a parameter n, and constructs a circuit which takes inputs $(c_1, \ldots, c_n, t)$ to compute $(c_1, \ldots, c_n,\ t \oplus (c_1 \& \ldots \& c_n))$.

Deutsch-Jozsa [47, §1.4.4] and Grover [47, §6] are well-known quantum algorithms. Given an integer v, IntegerComparator flips a target qubit if the number encoded in a list of control qubits is greater than v. Given n and an angle θ, MCRY (multi-controlled Y rotation) rotates a target qubit by θ in the Y basis if all of n control qubits are 1. For parameter n, Multiplier computes the binary representation of $x \cdot y$ for two numbers x and y encoded in n qubits each. Similarly, Adder computes $x + y$. Given n and a piecewise linear function f, PiecewiseLinearR rotates a target qubit by the angle $f(c)$ in basis Y, where c is encoded using n control qubits. In PolynomialPauliR, the function f is polynomial. Finally, given n and a list $v_1, \ldots, v_n$ of classical values, WeightedAdder computes $\sum c_i v_i$ for control qubits c_i.

For our Qiskit++ implementations of these algorithms, we simply removed the parts performing uncomputation or manually passing ancillae

TABLE 4.1: Comparing code complexity with and without Unqomp. The table shows the lines of code (incl. relative difference), whether ancilla allocation is modular (M), and whether uncomputation is implicit (U). Algorithms marked with [c] are from Cirq [90], while all other algorithms are from Qiskit [89].

	Original			**with Unqomp**			
Algorithm	lines	M	U	lines	diff	M	U
Adder [c]	31	✓		28	−10%	✓	✓
Deutsch-Jozsa	13	✓	✓	12	−8%	✓	✓
Grover	46	✓	✓	31	−33%	✓	✓
IntegerComparator	45			38	−16%	✓	✓
MCRY	7			4	−43%	✓	✓
MCX	16	✓		13	−19%	✓	✓
Multiplier [c]	13			11	−15%	✓	✓
PiecewiseLinearR	43			29	−33%	✓	✓
PolynomialPauliR	120			118	−2%	✓	✓
WeightedAdder	70			42	−40%	✓	✓

(see §4.6.3) and instead annotated ancilla qubits to enable automatic uncomputation.

UNCOMPUTATION SYNTHESIS TIME We ran the Qiskit++ compilation pipeline on a commodity laptop with 8 GB of RAM and 8 CPU cores at 2.40 GHz. For all algorithms presented in Table 4.1, our implementation of Unqomp required less than 1 second to introduce uncomputation.

4.6.2 *Q1: Code Length*

Table 4.1 compares the code lengths of the Qiskit (original) and Qiskit++ (with Unqomp) implementations for each algorithm. We observe that Unqomp consistently reduces the number of code lines, by up to 43%.

Most Qiskit algorithm implementations include explicit uncomputation code, which reverts all gates applied to the ancillae (see non-ticked in column U). This is not required in Qiskit++, leading to a significant code reduction. For example, in WeightedAdder, 39% of the source code lines deal with explicit uncomputation, which can be omitted using Unqomp. Explicitly inserting uncomputation not only increases code length but may also require rewriting the actual computation. For example, the original implementation of Adder is convoluted by re-ordered gates and interleaved uncomputation (cp. Fig. 4.3e). In contrast, uncomputation in Unqomp is implicit and ancillae are uncomputed automatically.

```
c1 = QuantumRegister(n)
c2 = QuantumRegister(n - 1)
g1 = MCXGate(len(c1))
g2 = MCXGate(len(c2))
a = QuantumRegister(max(
   g1.num_ancillae,
   g2.num_ancillae))
c.append(g1, c1, t, a)
c.append(g2, c2, t, a)
```

(a) Ancilla reuse in Qiskit.

```
c1 = QuantumRegister(n)
c2 = QuantumRegister(n - 1)
a = QuantumRegister(n - 2)
c.mcx(c1, t, a)
c.mcx(c2, t, a)
```

(b) Hardcoded qubits.

```
c1 = QuantumRegister(n - 1)
c2 = QuantumRegister(n)
c.mcx(c1, t)
c.mcx(c2, t)
```

(c) Modularity in Qiskit++.

Figure 4.8: Exposing ancillae to the caller in Qiskit. We shorten `c.append(MCXGate(len(c)), c, t, a)` to `c.mcx(c, t, a)`.

The Deutsch-Jozsa and Grover implementations do not include any uncomputation as they leverage phase kickback for the oracle evaluation. We note that the oracle circuit (which may perform uncomputation internally) is provided as a parameter to these algorithms and hence not part of the code considered here. For this reason, Unqomp cannot save any lines for Deutsch-Jozsa. The significant reduction for Grover originates from a modularity issue discussed next.

4.6.3 *Q2: Modularity*

Next, we compare the modularity of ancilla allocation. Overall, we find that Qiskit functions often break modularity, while Qiskit++ allows for modular code.

Exposing ancillae in Qiskit Qiskit functions expose the number of required ancillae to the caller using a field `num_ancillae` and rely on the caller to allocate them. This allows developers to manually reuse ancillae across functions, relying on correct uncomputation within the functions. For example, Fig. 4.8a shows a code snippet where the developer allocates the required ancillae `a` for two `MCX` gates.

While this construction respects modularity, it requires the developer to manually reuse qubits and tediously combine the ancilla requirements of multiple functions to determine the number of ancillae to allocate. This creates significant overhead: for instance, a third of the lines in the original

Grover implementation deal with ancilla management. Indeed, this construction is only used for few examples in our benchmark (see ticked ✓ in column M of Table 4.1).

BREAKING MODULARITY To reduce developer overhead, Qiskit functions are often implemented in a non-modular manner (see column M), where ancilla requirements are hard-coded using hand-crafted formulas relying on explicit knowledge about the implementation of called functions. As a typical example, Fig. 4.8b allocates exactly $n-2$ ancillae for MCX using knowledge of its internal implementation and of the length of both `c1` and `c2`.

Clearly, this increases coupling and leads to issues if library implementations are changed. Indeed, we found multiple inefficient usages of MCX and MCRY in the Qiskit library, allocating more ancillae than necessary for those gates due to a change in their default implementation. We reported three such issues to the developers, who have subsequently fixed them. [4] Fixing these issues further ensured that using the "v-chain" variant of MCX requires only a low number of basic gates by using all allocated ancillae, as we discuss in §4.6.4.

MODULARITY IN QISKIT++ In contrast to Qiskit, Qiskit++ allows allocating ancillae in a modular manner, as indicated by ✓ in column M of Table 4.1. A library function can locally allocate ancilla qubits for internal use, without exposing them to the caller of the function. Both caller and library can rely on Unqomp to automatically uncompute ancillae, and efficiently allocate and reuse qubits. For example, in Fig. 4.8c, the caller does not need to know about the ancillae in `MCX`.

4.6.4 *Q3: Efficiency*

We now compare the efficiency of circuits generated by Qiskit++ to circuits generated by Qiskit and Quipper, where Quipper is only applicable for classical programs, i.e., programs only consisting of qfree operations. Overall, we find that Qiskit++ circuits are often significantly more efficient.

4 `https://github.com/Qiskit/qiskit-terra/issues/4786` for MCRY, `https://github.com/Qiskit/qiskit-terra/issues/5320` for WeightedAdder, and `https://github.com/Qiskit/qiskit-terra/issues/5321` for PolynomialPauliR.

TABLE 4.2: Percentage of gates and qubits (qbs) saved by Unqomp, where higher numbers are better. Implementations marked with * were improved in Qiskit due to our bug reports (see Footnote 4). Entries in parenthesis require manual intervention, and ✗ indicates that Quipper's `classical_to_reversible_optim` is not applicable.

	Qiskit			**Quipper**		
	gates		**qbs**	**gates**		**qbs**
Algorithm	all	CX		all	CX	
Adder	34	35	0	56	62	17
Deutsch-Jozsa	0	0	0	(38)	(50)	(5)
Grover	0	0	0	(40)	(50)	(5)
IntegerComparator	31	48	0	41	51	0
MCRY	99.5	99.5	−4	✗	✗	✗
MCRY *	48	48	−4	✗	✗	✗
MCX	0	0	0	41	51	5
Multiplier	36	38	2	−25	−25	34
PiecewiseLinearR	41	42	29	✗	✗	✗
PolynomialPauliR	81	86	11	✗	✗	✗
PolynomialPauliR *	44	45	11	✗	✗	✗
WeightedAdder	43	55	−12	52	53	78
WeightedAdder *	30	33	−12	52	53	78
WeightedAdder alt. impl.	31	46	0	48	50	82
WeightedAdder alt. impl. *	16	20	0	48	50	82

APPROACH For all algorithms, we ran the full compilation pipeline as shown in Fig. 4.3, followed by Qiskit's decomposition into the two basic gates CX and $U3$ as post-processing, using the "v-chain" variant of MCX. Further, we instantiated the oracle circuit in Grover and Deutsch-Jozsa with an MCX gate. For the comparison to Quipper, we manually translated the algorithms to Quipper using the `classical_to_reversible_optim` construct to insert uncomputation. We then applied the Qiskit decomposition discussed above to the resulting circuits.

We show the resulting reduction from Qiskit to Qiskit++ and Quipper to Qiskit++ for gates and qubits in Table 4.2. For completeness, Table 4.2 also shows the reduction in CX gates only (which are typically more expensive than $U3$ gates), with analogous results.

LIMITATIONS OF CLASSICAL UNCOMPUTATION As shown in Table 4.2, the uncomputation offered by Quipper is severely limited: because construct `classical_to_reversible_optim` only supports classical programs, the presence of non-qfree gates prevents directly applying it on Deutsch-Jozsa, Grover, MCRY, PiecewiseLinearR, and PolynomialPauliR. However, when ancillae are only used in qfree parts of the circuit, it is possible to isolate those

```
c.ccx(c0, c1, a0)
c.ccx(c2, a0, r)
c.h(c0)
```

(a) Separable qfree section.

```
c.ccx(c0, c1, a0)
c.ccx(c2, a0, a1)
c.cry(a1, r, 2)
```

(b) Non-qfree gate on ancilla.

FIGURE 4.9: Limitation of Classical Uncomputation. Variables `c0, c1, c2, r` are qubits and `a0, a1` are ancillae. The call `c.ccx(c0, c1, a0)` applies a CCX gate with controls `c0, c1` and target `a0`, while `c.cry(a0, r, 2)` applied a controlled rotation with control `a0`, target `r`, and angle 2.

qfree parts and apply `classical_to_reversible_optim` to these, before combining them with the non-qfree parts of the algorithm. For example, Fig. 4.9a shows an extract of the Grover implementation for input size 3. The ancilla `a0` is only used in the first two lines. Thus, we can apply Quipper `classical_to_reversible_optim` to the circuit generated by those two lines, and append to its result the gate corresponding to the third line. This strategy allows applying Quipper uncomputation to Grover and Deutsch-Jozsa. However, this approach not only requires manual intervention but also results in less efficient circuits (see Table 4.2).

When ancillae are used in parts of the circuit that are not purely qfree—as shown for instance in Fig. 4.9b on the code for MCRY with input size 3—the above separation is not possible. This is the case for MCRY, PiecewiseLinearR and PolynomialPauliR. Thus, Quipper cannot synthesize uncomputation for these examples (see ✗ in Table 4.2).

4.6.4.1 *Reduction in Gate Count*

For all but three algorithms, Qiskit++ significantly reduces the number of gates compared to Qiskit, by up to 99.5%. These extremely high savings are partially due to a regression bug in Qiskit (see Footnote 4), which Unqomp helps to avoid. Even for algorithms not affected by this regression, or after fixing this regression (* in Table 4.2), Qiskit++ allows for significant savings of up to 48%. On the three algorithms where Qiskit++ does not outperform Qiskit, both yield circuits with identical size. Being well-studied quantum algorithms, the implementations of Grover and Deutsch-Jozsa have been manually optimized by experts to reduce gate and qubit counts. Similarly, MCX has been heavily studied and optimized [84].

Compared to Quipper, Qiskit++ almost always produces fewer gates, with savings of up to 66%. Quipper outperforms Qiskit++ only on Multiplier, due to an optimization pass performed by Quipper but not Qiskit++. In particular, Quipper applies constant propagation and can for example remove CCX gates if one of their controls is known to be 0. We note that this optimization is orthogonal to our work and could in principle be integrated in Qiskit++ as well. When disabling the optimization in Quipper, Qiskit++ consistently produces fewer gates than Quipper.

ORIGIN OF REDUCTIONS Overall, the gate savings of Qiskit++ can be explained by (i) redundant uncomputation in the original implementation (see Fig. 4.1) and (ii) CCX gate optimizations performed during our compilation (see §4.4.4).

Redundant uncomputation concerns MCRY, PiecewiseLinearR, PolynomialPauliR, and WeightedAdder, as these examples rely on libraries for integrating sub-components in a modular manner.

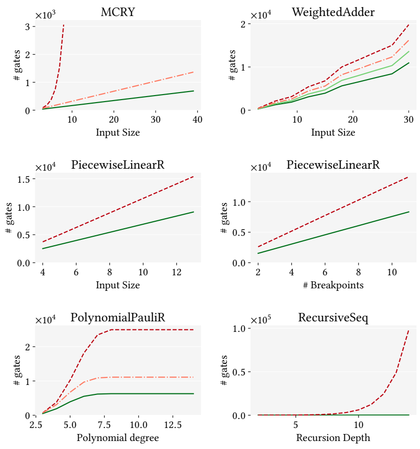
CCX gate optimizations lead to significant gains for all algorithms, except for those where no savings were observed. Note that this optimization is already partially applied in the baseline Qiskit implementations (manually to MCX, and indirectly to all examples that use MCX). Consistently applying this optimization to the Qiskit baseline would be virtually impossible without Unqomp: it would require knowing which CCX gates are later uncomputed, which is impossible to determine for library functions computing values which may or may not be used as ancillae.

For Quipper, an additional source of gate savings lies in the more fine-grained control allowed by Qiskit++. Algorithms must be implemented as Haskell functions in Quipper. Hence, many optimizations, such as temporarily changing the value of an input argument in-place, cannot be used.

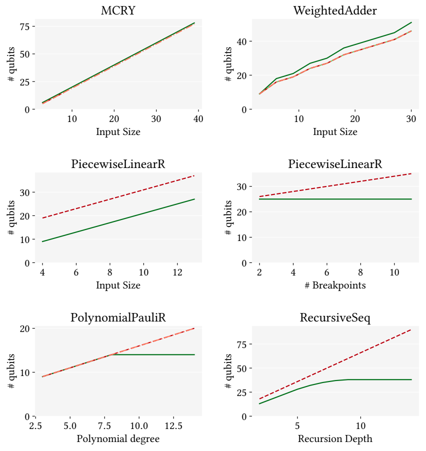
ASYMPTOTIC GAINS Fig. 4.10a shows the effect of varying some circuit parameters for selected algorithms: efficiency gains often increase with increasing complexity. Even after the fixes following our bug reports (Footnote 4), Unqomp allows for a significant reduction in all shown algorithms.

GATE EXPLOSION FOR RECURSIVE CALLS As visualized in Fig. 4.1, using library functions in Qiskit leads to redundant uncomputation. We now demonstrate that this effect is arbitrarily amplified by nested library calls. As a consequence, we should expect Unqomp to yield even larger efficiency gains when quantum algorithms become increasingly complex.

We implemented a toy algorithm RecursiveSeq that computes the sequence $x_{n+1} = 2x_n + 1$ according to its recursive definition: the function



(a) Gate counts.



(b) Qubit counts.

FIGURE 4.10: Gates and qubits for different algorithm parameters using Qiskit++ (—; this work) and Qiskit (- - -). Some implementations were improved as a result of our bug reports (see Footnote 4), shown as -·-·. For WeightedAdder, we show an alternative Qiskit++ implementation (—) trading gates for qubits. Lower values are better.

computing x_{n+1} uses a recursive function call to compute x_n on an ancilla a and returns $2a + 1$. We implemented this algorithm in Qiskit by explicitly performing uncomputation in each recursive call. In Qiskit++, uncomputation is automatic. As shown in Fig. 4.10a, this leads to an exponential increase of gate counts for Qiskit, while the count only increases linearly for Qiskit++. This suggests that more generally, Unqomp allows significant efficiency gains for complex algorithms with deep call trees.

4.6.4.2 *Reduction in Qubit Count*

Table 4.2 also compares the number of qubits for the implementations, and Fig. 4.10b shows the impact of varying algorithm parameters on the number of qubits.

Overall, Quipper yields significantly higher qubit counts. Compared to Qiskit, Qiskit++ results in an identical number of qubits for many algorithms, showing that Unqomp's ancilla allocation (§4.4.4) can often compete with manual allocation. Furthermore, for some examples, Unqomp yields a significant reduction in qubits, indicating that a completely manual approach fosters errors and missed optimizations.

For RecursiveSeq, Unqomp finds a non-trivial qubit allocation that is hard to detect manually, and thus only requires a constant number of qubits.

For PolynomialPauliR, the number of qubits in Qiskit++ remains constant for polynomial degrees d above 8, while it increases linearly for Qiskit. This is due to the Qiskit implementation allocating d qubits in a hard-coded fashion (see §4.6.3). However, a detailed analysis of the code shows that the number of required ancillae is actually only $\min(d, n)$, where n is the input size. In contrast to Qiskit, Unqomp automatically finds this improved, non-trivial qubit allocation. Similarly, for PiecewiseLinearR, Unqomp finds a more efficient qubit allocation, using n ancillae for an input size n, instead of $n + b$, where b is the number of breakpoints.

TRADING OFF QUBITS AND GATES Unfortunately, the gate count reduction for WeightedAdder comes at the cost of more qubits. This is a result of Unqomp performing uncomputation later in the circuit than the Qiskit implementation, which prevents some qubit reuse. Still, using a slightly modified alternative Qiskit++ implementation, we can trade Unqomp's gate savings for fewer qubits, resulting in identical qubit counts (see Table 4.2). Similarly, for MCRY, Qiskit++ requires exactly one more qubit than the original Qiskit implementation, as it uses slightly different code—instead of two MCX gates it uses only one MCX with its uncomputation and an extra ancilla qubit. This trade-off is only interesting when using automatic uncomputation: as MCX uses a lot of internal auxiliary values, modular manual uncomputation is quite expensive, while automatic uncomputation is cheap, as illustrated in Fig. 4.1. This cheap uncomputation allows Qiskit++ to produce half as many gates as Qiskit, at the cost of only one extra qubit.

4.7 RELATED WORK

We now discuss previous works related to Unqomp both in terms of (i) our goal of synthesizing uncomputation, and (ii) key aspects of our approach to address this goal.

AUTOMATIC UNCOMPUTATION Table 4.3 summarizes existing approaches to handle automatic uncomputation. As discussed next, these approaches differ from Unqomp in that they either do not provide a compilation to circuits, or only apply to classical computation. We note that ReQWire [58] can additionally prove that a manually provided uncomputation is safe. However, it cannot automatically synthesize uncomputation, except for

TABLE 4.3: Comparing Unqomp to related approaches.

Approach/Language	Autom. Uncomp.	Circuit
Quipper [23]	(qfree)	✓
REVS [81]	(qfree)	✓
REVERC [59]	(qfree)	✓
ReQWire [58]	(qfree)	✓
Silq [1]	✓	✗
Unqomp (this work)	✓	✓

purely classical circuits. Table 4.3 omits SQUARE [91] as it does not synthesize uncomputation: SQUARE expects the programmer to manually provide and mark the uncomputation code blocks for each ancilla, and then saves qubits or operations by interleaving those blocks. Further, SQUARE suffers from multiple shortcomings such as skipping uncomputation of some ancillae [14, §7.1].

SILQ: ENABLE SAFE UNCOMPUTATION Silq is the closest work to Unqomp in that it also promises automatic uncomputation and relies on analogous high-level insights for ensuring correctness, namely the notion of *qfree* gates and *controls* (cp. §4.5.3). However, while Silq's type system ensures that safe uncomputation is possible for all temporary values, Silq does not provide a compiler that synthesizes this uncomputation. In contrast, Unqomp synthesizes uncomputation, which allows extending arbitrary circuit-based languages (such as Qiskit [39]) to support automatic uncomputation. We can view Unqomp as a key step towards compiling Silq, which in particular requires automatically generating safe uncomputation.

QFREE PROGRAMS Quipper [23], REVS [81], REVERC [59], and ReQWire [58] support automatic uncomputation only for classical programs, i.e., programs that only use qfree functions. Further, ReverC only uncomputes boolean expressions, meaning it is not applicable to any of our examples in Table 4.1. In contrast, only Silq and Unqomp support uncomputation in quantum programs that interleave qfree with non-qfree operations. Only supporting qfree computations is a severe restriction—half of the programs we evaluated (see §4.6) are not fully qfree. Further, the proposed workflow for these approaches is to compile the classical part of a quantum program and then insert the result into the resulting quantum circuit. However, this workflow cannot always be applied, as shown in Fig. 4.9, and generally results in inefficient circuits, analogously to Fig. 4.1.

```
operation ErroneousUncomputation(x : Qubit) : Unit
{
    use ancilla = Qubit();
    ApplyWith(CopyX, ModifyX, (x, ancilla));
    // Error: Ancilla is not in zero state
}

operation CopyX(x : Qubit, ancilla : Qubit) : Unit is Adj
{
    CNOT(x, ancilla);
}

operation ModifyX(x : Qubit, ancilla : Qubit): Unit is Adj
{
    CNOT(ancilla, x);
    // Error: modifies x which is needed for uncomputation
}
```

FIGURE 4.11: `ApplyWith` producing erroneous uncomputation.

CONVENIENCE FUNCTIONS Various quantum languages offer convenience functions that simplify manual uncomputation, such as `ApplyWith` in Q# [22] or `with_computed` in Quipper [23]. However, relying on these features cannot guarantee the resulting uncomputation is safe, as incorrectly using them does not result in an error.

For example, Fig. 4.11 shows Q# code using `ApplyWith` to uncompute a in Fig. 4.6b. As uncomputing a is physically impossible (see §4.5.2) and `ApplyWith` performs no static checks, this program results in an incorrect circuit whose error is only detected at runtime (i.e., during simulation).

In addition to being unsafe, convenience functions are often tedious to use. For instance, `ApplyWith` cannot be used in combination with for-loops, forcing even expert Q# developers to resort to manual uncomputation in some cases. [5] Finally, convenience functions often generate inefficient circuits, as explained in Fig. 4.1.

CIRCUIT GRAPHS Various existing works have represented circuits in terms of circuit graphs. In classical computation, dependency graphs have long been used to represent computations without enforcing irrelevant ordering constraints (see e.g., [92], [88, §5.2]). Naturally, works in this domain do not discuss quantum computations or quantum circuits.

5 For an example, see `https://github.com/microsoft/QuantumKatas/blob/7ba83e55703fda4ff945fc6e89050f4ee179e5bc/RippleCarryAdder/ReferenceImplementation.qs#L73`

Multiple works in quantum computation operate on graph-based circuit representations. However, as none of them are geared towards uncomputation, their graphs (i) do not distinguish between target, control, and anti-dependency edges [93, 94, 95], (ii) are often limited to only a few types of gates [94, 95], and (iii) are not suitable for inserting (uncomputation) gates because they do not contain enough information to reconstruct the circuit they represent [93].

ANCILLA ALLOCATION Our approach to ancilla allocation (§4.4.4) is an instantiation of linear scan register allocation [85], with one key simplification: instead of a fixed number of registers (and the option of spilling to the heap), we have an unlimited number of potential ancillae. The cost of a specific allocation is hence simply the number of ancillae used, instead of the cost of the operations on spilled registers, allowing for an optimal allocation given a graph linearization.

4.8 IMPACT

Since the publication of Unqomp, several researchers have worked on the topic of automatic uncomputation or even built directly upon Unqomp, highlighting its importance.

As discussed in §3.9, Qunity [24] extends Silq's notion of automatic uncomputation to non-qfree expressions, at the cost of probabilistic errors. Qunity provides a compilation procedure for its programs, but unlike Unqomp does not discuss how to minimize the gate count of produced circuits. In fact, as its compilation is modular (see Fig. 4.1a), we do not expect its suggested compilation to be particularly efficient.

Further, as mentioned in §3.9, Qrisp [25] was inspired by Silq and automates uncomputation by using Unqomp.

While VQO [34] does not support automatically synthesizing uncomputation code, it can efficiently compile quantum oracles to circuits containing non-qfree gates, e.g., applying a quantum Fourier transform. Due to this improved flexibility, the resulting circuits are more efficient than Quipper implementations. Overall, we believe these approaches are complementary, i.e., we conjecture that combining both automatic uncomputation (from Unqomp) and the ability to generate non-qfree gates (from VQO) would likely yield the best results.

Our own most recent work Reqomp [14] extends Unqomp with the ability to respect a given resource constraint, allowing to synthesize circuits for space-constrained quantum computers.

Further, Unqomp naturally complements other more recent works. For example, Twist would reportedly benefit from the presence of automatic uncomputation [77, §11].

4.9 CONCLUSION

We presented Unqomp, a procedure synthesizing automatic uncomputation for quantum circuits, using an internal representation in terms of circuit graphs. Unqomp can be readily integrated into existing quantum languages, which we demonstrated by extending Qiskit to Qiskit++.

Our evaluation showed that Unqomp reduces the amount of code, improves code modularity, and yields substantially more efficient circuits in terms of number of gates and qubits.

5

ABSTRAQT: ANALYSIS OF QUANTUM CIRCUITS VIA ABSTRACT STABILIZER SIMULATION

Stabilizer simulation can efficiently simulate an important class of quantum circuits consisting exclusively of Clifford gates. However, all existing extensions of this simulation to arbitrary quantum circuits including non-Clifford gates suffer from an exponential runtime.

To address this challenge, we present a novel approach for efficient stabilizer simulation on arbitrary quantum circuits, at the cost of lost precision. Our key idea is to compress an exponential sum representation of the quantum state into a single *abstract* summand covering (at least) all occurring summands. This allows us to introduce an *abstract stabilizer simulator* that efficiently manipulates abstract summands by *over-approximating* the effect of circuit operations including Clifford gates, non-Clifford gates, and (internal) measurements.

We implemented our abstract simulator in a tool called Abstraqt and experimentally demonstrate that Abstraqt can establish circuit properties intractable for existing techniques.

5.1 INTRODUCTION

Stabilizer simulation [96] is a promising technique for efficient classical simulation of quantum circuits consisting exclusively of *Clifford* gates. Unfortunately, generalizing stabilizer simulation to arbitrary circuits including non-Clifford gates requires exponential time [36, 37, 38, 97, 98, 99]. Specifically, the first such generalization by Aaronson and Gottesman [36, §VII-C] tracks the quantum state ρ at any point in the quantum circuit as a sum whose number of summands m grows exponentially with the number of non-Clifford gates:

$$\rho = \sum_{i=1}^{m} c_i P_i \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2}. \tag{5.1}$$

Here, while c_i, P_i, b_{ij}, and Q_j can be represented efficiently (see §5.2), the overall representation is inefficient due to exponentially large m.

ABSTRACTION The key idea of ABSTRAQT is to avoid tracking the exact state ρ of a quantum system and instead only track key aspects of ρ. To this end, we rely on the established framework of abstract interpretation [100, 101], which is traditionally used to analyze classical programs [102, 103] or neural networks [104] by describing sets of possible states without explicitly enumerating all of them. Here, we use abstract interpretation to describe the set of quantum states that could occur at a specific point during execution of a circuit, by *over-approximating* the summands that could occur in any of those quantum states ρ.

MERGING SUMMANDS This allows us to curb the exponential blow-up of stabilizer simulation by merging multiple summands in Eq. (5.1) into an abstract single summand which over-approximates all summands, at the cost of lost precision. The key technical challenge addressed by our work is designing a suitable *abstract domain* to describe sets of summands, accompanied by the corresponding *abstract transformers* to over-approximate the actions performed by the original exponential stabilizer simulation on individual summands.

As a result, our approach is both efficient and exact on Clifford circuits, as these circuits never require merging summands. On non-Clifford circuits, merging summands trades precision for efficiency. Moreover, our approach naturally allows us to merge the possible outcomes of a measurement into a single abstract state, preventing an exponential path explosion when simulating multiple internal measurements.

MAIN CONTRIBUTIONS The main contributions of this chapter are:

- An abstract domain (§5.4) to over-approximate a quantum state represented by Eq. (5.1).
- Abstract transformers (§5.5) to simulate quantum circuits, including gate applications and measurements.
- An efficient implementation[1] of our approach in a tool called ABSTRAQT (§5.6), together with an evaluation showing that ABSTRAQT can establish circuit properties that are intractable for existing tools (§5.7).

RESULTS Overall, we find that ABSTRAQT is useful in scenarios where a full simulation of a given circuit is intractable, but establishing specific properties of the considered circuit is desirable.

1 Our implementation is available at `https://github.com/eth-sri/abstraqt`.

For example, in our evaluation (§5.7), we demonstrate that ABSTRAQT can establish that a circuit ultimately restores some qubits to state $|0\rangle$. As precisely simulating the entire circuit is intractable, ABSTRAQT is typically the only existing tool able to establish this fact on 12 benchmarked circuits. In contrast, existing tools typically yield incorrect results, throw errors, run out of memory, time out, or are too imprecise to establish the resulting state is $|0\rangle$.

OUTLOOK ABSTRAQT trades precision for efficiency by abstracting the stabilizer simulation from Eq. (5.1), therefore allowing to establish properties of quantum circuit outputs when full simulation is intractable. Such results may be useful for tasks like (i) establishing that an internal circuit state allows for specific optimizations, (ii) debugging quantum computers by establishing invariants that can be checked at runtime, and more generally (iii) static analysis of quantum circuits, or (iv) verification of the correctness of quantum circuits.

Further, as discussed in §5.7.4, ABSTRAQT abstracts the very first stabilizer simulation generalized to non-Clifford gates by Aaronson and Gottesman [36, §VII-C]. We believe that our encouraging results pave the way to introduce analogous abstraction to various follow-up works which improve upon this simulation [37, 38, 98, 99]. As these more recent works scale better than [36, §VII-C], we expect that a successful application of abstract interpretation to them will yield even more favorable trade-offs between precision and efficiency.

5.2 BACKGROUND

In the following, we present the background necessary for this chapter.

QUANTUM STATE As discussed in Chapter 2, a pure n-qubit quantum state can be represented as a *state vector* $\psi \in \mathbb{C}^{2^n}$. In this chapter, however, we will typically represent state ψ as a *density matrix* $\rho \in \mathbb{C}^{2^n \times 2^n}$, defined as $\rho = \psi\psi^\dagger$, where $\psi^\dagger$ denotes the conjugate transpose of ψ. For a mixed state, i.e., a distribution over pure states ψ_i with probability p_i, the corresponding density matrix is $\rho = \sum_i p_i \psi_i \psi_i^\dagger$. Because both ψ and ρ store exponentially many values, they cannot be represented explicitly for large n.

QUANTUM GATE An n-qubit quantum gate $U \in \mathcal{U}(2^n)$ thus evolves ρ to $U\rho U^\dagger$, where $\mathcal{U}(2^n)$ is the set of unitary $2^n \times 2^n$ matrices.

STABILIZER SIMULATION The key idea of stabilizer simulation [36, 96] is representing quantum states $\rho = \psi\psi^\dagger$ implicitly, by *stabilizers* Q which stabilize the state ψ, that is $Q\psi = \psi$. As shown in [36], appropriately selecting n stabilizers Q_j then specifies a unique n-qubit state $\rho = \prod_{j=1}^{n} \frac{\mathbb{I}+Q_j}{2}$.

In stabilizer simulation, all Q_j are *Pauli elements* from $\mathcal{P}_n$ of the form $\mathrm{i}^v \cdot P^{(0)} \otimes \cdots \otimes P^{(n-1)}$, where $P^{(j)} \in \{X, Y, Z, \mathbb{I}_2\}$ and $\mathrm{v} \in \mathbb{Z}_4$. This directly implies that all stabilizers Q_i for the same state ψ commute, that is $Q_iQ_j = Q_jQ_i$, as elements from the Pauli group $\mathcal{P}_n$ either commute or anti-commute. These elements can be represented efficiently in memory by storing v and $P^{(0)}, \ldots, P^{(n-1)}$. In App. C.2, we list states stabilized by Pauli matrices (Table C.1) and the results of multiplying Pauli matrices (Table C.2). Further, in this chapter we use the functions *bare* $\mathfrak{b}\colon \mathcal{P}_n \to \mathcal{P}_n$ and *prefactor* $\mathfrak{f}\colon \mathcal{P}_n \to \mathbb{Z}_4$ which extract the Pauli matrices without the prefactor and the prefactor, respectively:

$$\mathfrak{f}(\mathrm{i}^v P^{(0)} \otimes \cdots \otimes P^{(n-1)}) = v, \tag{5.2}$$

$$\mathfrak{b}(\mathrm{i}^v P^{(0)} \otimes \cdots \otimes P^{(n-1)}) = P^{(0)} \otimes \cdots \otimes P^{(n-1)}. \tag{5.3}$$

Applying gate U to state ρ can be reduced to conjugating the stabilizers Q_j with U:

$$U\rho U^\dagger = U\Big(\prod_{j=1}^{n} \frac{\mathbb{I}+Q_j}{2}\Big)U^\dagger \overset{[47,\ \text{Sec. } 10.5]}{=} \prod_{j=1}^{n} \frac{\mathbb{I}+UQ_jU^\dagger}{2}. \tag{5.4}$$

While Eq. (5.4) holds for any gate U, stabilizer simulation can only exploit it if $UQ_jU^\dagger \in \mathcal{P}_n$. *Clifford gates* such as S, H, $CNOT$, $\mathbb{I}$, X, Y, and Z satisfy this for any $Q_j \in \mathcal{P}_n$.

To also support the application of non-Clifford gates such as T gates, we follow [36, §VII.C] and represent ρ more generally as

$$\rho = \sum_{i=1}^{m} c_i P_i \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2},$$

for $c_i \in \mathbb{C}$, $P_i \in \mathcal{P}_n$, $b_{ij} \in \mathbb{B}$, and $Q_j \in \mathcal{P}_n$. Here, applying U to ρ amounts to replacing P_i by $UP_iU^\dagger$ and Q_j by $UQ_jU^\dagger$, which we can exploit if both $UP_iU^\dagger$ and $UQ_jU^\dagger$ lie in $\mathcal{P}_n$.

Otherwise, we decompose[2] U to the sum $\sum_{p=1}^{K} d_p R_p$, where $d_p \in \mathbb{C}$ and $R_p \in \mathfrak{b}(\mathcal{P}_n)$ are bare Pauli elements, which have a prefactor of $\mathrm{i}^0 = 1$. Then,

$$U\rho U^\dagger = \Big(\sum_{p=1}^{K} d_p R_p\Big)\Big(\sum_{i=1}^{m} c_i P_i \prod_{j=1}^{n} \tfrac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2}\Big)\Big(\sum_{q=1}^{K} d_q R_q\Big)^\dagger \tag{5.5}$$

$$\overset{[36,\,\S\text{VII.C}]}{=} \sum_{p=1}^{K}\sum_{i=1}^{m}\sum_{q=1}^{K} c_{piq} P_{piq} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{b_{ijq}} Q_j}{2}, \tag{5.6}$$

for $c_{piq} = d_p c_i d_q^* \in \mathbb{C}$, $P_{piq} = R_p P_i R_q \in \mathcal{P}_n$, and $b_{ijq} = b_{ij} + Q_j \diamond R_q \in \mathbb{B}$. Here, d_q^* denotes the complex conjugate of d_q, $+$ denotes addition modulo 2, and $Q_j \diamond R_q$ is the commutator defined as 0 if Q_j and R_q commute and 1 otherwise. Note that $\cdot \diamond \cdot \colon \mathcal{P}_n \times \mathcal{P}_n \to \mathbb{B}$ has the highest precedence.

Overall, the decomposition of a k-qubit non-Clifford gate results in at most $K = 4^k$ summands, thus blowing up the number of summands in our representation by at most $4^k \cdot 4^k = 16^k$. In practice, the blow-up is typically smaller, e.g., decomposing a T gate only requires 2 summands, while decomposing a $CCNOT$ gate requires 8 summands.

MEASUREMENT Recall that we introduced measurements in the computational basis in Chapter 2. In this chapter, we consider more general measurements in any *Pauli basis*.

Measuring in bare Pauli basis $P \in \mathfrak{b}(\mathcal{P}_n)$ yields one of two possible quantum states. They can be computed by applying the two *projections* $P_+ := \frac{\mathbb{I}+P}{2}$ and $P_- = \frac{\mathbb{I}-P}{2}$, resulting in states $\rho_+ = P_+\rho P_+$ and $\rho_- = P_-\rho P_-$, respectively. For example, collapsing the ith qubit to $|0\rangle$ or $|1\rangle$ corresponds to measuring in Pauli basis $Z_{(i)}$. The probability of outcome ρ_+ is $\operatorname{tr}(\rho_+)$, and analogously for ρ_-. Note that we avoid renormalization for simplicity. We discuss in §5.5 how measurements are performed in stabilizer simulation [36, Sec. VII.C].

ABSTRACT INTERPRETATION Abstract interpretation [100] is a framework for formalizing approximate but sound calculation. An *abstraction* consists of ordered sets $(2^{\mathcal{X}}, \subseteq)$ and $(\boldsymbol{\mathcal{X}}, \leq)$, where $\mathcal{X}$ and $\boldsymbol{\mathcal{X}}$ are called *concrete set* and *abstract set* respectively together with a *concretization function* $\gamma\colon \boldsymbol{\mathcal{X}} \to 2^{\mathcal{X}}$ which indicates which concrete elements $x = \gamma(\boldsymbol{x}) \subseteq \mathcal{X}$ are represented by the abstract element $\boldsymbol{x}$. Additionally, $\bot \in \boldsymbol{\mathcal{X}}$ refers to $\emptyset = \gamma(\bot) \subseteq \mathcal{X}$ and $\top \in \boldsymbol{\mathcal{X}}$ refers to $\mathcal{X} = \gamma(\top)$.

2 This decomposition always exists and is unique, as bare Pauli elements span (more than) $\mathcal{U}(2^n)$.

Table 5.1: Transformers for the interval abstraction.

Function	Abstract Transformer	Efficient Closed form
$+$	$[l_1, u_1] +^\sharp [l_2, u_2] = [l', u']$	$l' = l_1 + l_2$ and $u' = u_1 + u_2$
$\cdot$	$[l_1, u_1] \cdot^\sharp [l_2, u_2] = [l', u']$	$l' = \min(l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2)$, u' defined analogously with max
exp	$\exp^\sharp([l, u]) = [l', u']$	$l' = \exp(l)$ and $u' = \exp(u)$
cos	$\cos^\sharp([l, u]) = [l', u']$	exists, several case distinctions necessary
$\cup$	$[l_1, u_1] \sqcup [l_2, u_2] = [l', u']$	$l' = \min(l_1, l_2)$ and $u' = \max(u_1, u_2)$

An abstract transformer $f^\sharp \colon \boldsymbol{\mathcal{X}} \to \boldsymbol{\mathcal{X}}$ of a function $f \colon \mathcal{X} \to \mathcal{X}$ satisfies $\gamma \circ f^\sharp(\boldsymbol{x}) \supseteq f \circ \gamma(\boldsymbol{x})$ for all $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$, where f was lifted to operate on subsets of $\mathcal{X}$. This ensures that $f^\sharp$ (over-)approximates f, a property referred to as *soundness* of $f^\sharp$. Abstract transformers can analogously be defined for functions $f \colon \mathcal{X}^n \to \mathcal{X}$. Further, we introduce *join* $\sqcup \colon \boldsymbol{\mathcal{X}} \times \boldsymbol{\mathcal{X}} \to \boldsymbol{\mathcal{X}}$, satisfying $\gamma(\boldsymbol{x}) \cup \gamma(\boldsymbol{y}) \subseteq \gamma(\boldsymbol{x} \sqcup \boldsymbol{y})$. Throughout this chapter, we distinguish abstract objects $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$ and concrete objects $x \in \mathcal{X}$ by stylizing them in bold or non-bold respectively.

As an example, a common abstraction is the interval abstraction with $\mathcal{X} = \mathbb{R}$. The abstract set is the set of intervals

$$\boldsymbol{\mathcal{X}} = \{(l, u) \mid l, u \in \mathbb{R} \cup \{\pm\infty\}\},$$

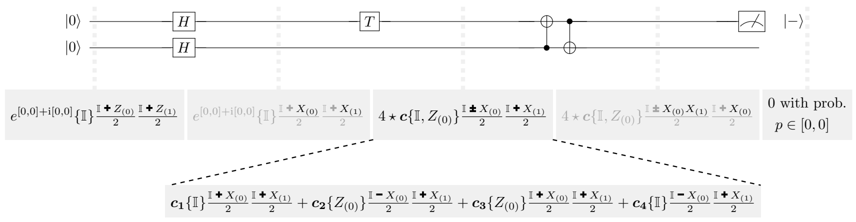
where $\boldsymbol{x} = (l, u)$ is a tuple. The concretization function $\gamma \colon \boldsymbol{\mathcal{X}} \to \mathcal{X}$ maps these tuples to sets:

$$\gamma(\boldsymbol{x}) = [l, u] = \{y \in \mathbb{R} \mid l \leq y \leq u\}.$$

Further, $\top = (-\infty, \infty)$ and $\bot = (l, u)$ for $l > u$. Common abstract transformers for the interval abstraction are shown in Table 5.1.

The transformers in Table 5.1 are *precise*, meaning that for $f \colon \mathbb{R} \to \mathbb{R}$, we have that $f^\sharp((l, u)) = (\min_{l \leq v \leq u} f(v), \max_{l \leq v \leq u} f(v))$ and analogously for $f \colon \mathbb{R}^n \to \mathbb{R}$. An abstract transformer for a composition of functions $f \circ g$ is the composition of the abstract transformers. Although this is sound, it is not necessarily precise: let $g \colon \mathbb{R} \to \mathbb{R}^2$ with $g(x) = \binom{x}{x}$ and $f \colon \mathbb{R}^2 \to \mathbb{R}$ with $f(x, y) = x \cdot y$, then $f \circ g(x) = x^2$, but $f^\sharp \circ g^\sharp((-2, 2)) = (-4, 4)$ whereas a precise transformer would map $(-2, 2)$ to $(0, 4)$.

NOTATIONAL CONVENTION In slight abuse of notation, throughout this chapter we may write the concretization of abstract elements instead of the



|0⟩
|0⟩
H
H
T
|−⟩
$e^{[0,0]+i[0,0]}\{\mathbb{I}\}\frac{\mathbb{I}+Z_{(0)}}{2}\frac{\mathbb{I}+Z_{(1)}}{2}$
$e^{[0,0]+i[0,0]}\{\mathbb{I}\}\frac{\mathbb{I}+X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}$
$4\star \boldsymbol{c}\{\mathbb{I},Z_{(0)}\}\frac{\mathbb{I}\pm X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}$
$4\star \boldsymbol{c}\{\mathbb{I},Z_{(0)}\}\frac{\mathbb{I}\pm X_{(0)}X_{(1)}}{2}\frac{\mathbb{I}+X_{(0)}}{2}$
0 with prob. $p \in [0,0]$
$\boldsymbol{c_1}\{\mathbb{I}\}\frac{\mathbb{I}+X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}+\boldsymbol{c_2}\{Z_{(0)}\}\frac{\mathbb{I}-X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}+\boldsymbol{c_3}\{Z_{(0)}\}\frac{\mathbb{I}+X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}+\boldsymbol{c_4}\{\mathbb{I}\}\frac{\mathbb{I}-X_{(0)}}{2}\frac{\mathbb{I}+X_{(1)}}{2}$

FIGURE 5.1: Overview of ABSTRAQT, where we define $\boldsymbol{c}$ and $\boldsymbol{c_1}$–$\boldsymbol{c_4}$ in §5.3.

abstract element itself. For example, for $(0,1) \in \boldsymbol{R}$, we write $[0,1]$ defined as $\{v \in \mathbb{R} \mid 0 \le v \le 1\}$ to indicate that it represents an interval. Where clear from context, we omit $\sharp$ and write f for $f^\sharp$. For example, we write $[l_1, u_1] + [l_2, u_2]$ for $[l_1, u_1] +^\sharp [l_2, u_2]$.

5.3 OVERVIEW

In this section, we showcase ABSTRAQT by applying it to the example circuit in Fig. 5.1. Overall, ABSTRAQT proceeds analogously to [36, §VII-C], but operates on abstract summands representing many concrete summands.

EXAMPLE CIRCUIT We first discuss the circuit in Fig. 5.1. Both qubits are initialized to $|0\rangle$. The circuit then applies a succession of gates. The abstract representation of the state after the application of each gate is shown in the gray boxes below the circuit. On the final state, the circuit collapses the upper qubit to $|-\rangle$ by applying the projection $M_- = \frac{\mathbb{I}-X_{(0)}}{2}$. Precise circuit simulation shows that the probability of obtaining $|-\rangle$ is 0, in this case. In the following, we demonstrate how ABSTRAQT computes an over-approximation of this probability.

INITIAL STATE The density matrix for the initial state $|0\rangle \otimes |0\rangle$ can be represented as (see [36]):

$$\rho_A = 1\mathbb{I}\frac{\mathbb{I}+(-1)^0 Z_{(0)}}{2}\frac{\mathbb{I}+(-1)^0 Z_{(1)}}{2}.$$

To translate this to an abstract density matrix, we simply replace some elements by abstract representations. This gives the following initial abstract state:

$$\boldsymbol{\rho_A} = e^{[0,0]+[0,0]\mathrm{i}}\{\mathbb{I}\}\frac{\mathbb{I}+(-1)^{\{0\}}Z_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}Z_{(1)}}{2}. \tag{5.7}$$

Here we abstract booleans as sets, for instance $\{0\}$. For conciseness, in Fig. 5.1 we write $x \mathbin{\boldsymbol{+}} y$, $x \mathbin{\boldsymbol{-}} y$, and $x \mathbin{\boldsymbol{\pm}} y$ for $x+(-1)^{\{0\}}y$, $x+(-1)^{\{1\}}y$, and $x+(-1)^{\{0,1\}}y$. Further, we represent abstract complex numbers in polar form with logarithmic length, using real intervals: 1 is represented as $e^{[0,0]+[0,0]\mathrm{i}}$, while we can over-approximate the set of complex numbers $\{1, \mathrm{i}\}$ as $e^{[0,0]+[0,\frac{\pi}{2}]\mathrm{i}}$. Finally, we abstract Pauli elements as sets, such as $\{\mathbb{I}\}$ in Fig. 5.1 and Eq. (5.7). In §5.4, we will clarify how we store these sets efficiently, for example representing $\{\mathbb{I}\}$ as $\mathrm{i}^{\{0\}} \cdot \{\mathbb{I}\} \otimes \{\mathbb{I}\}$ and $\{\mathrm{i}\cdot\mathbb{I}, \mathrm{i}\cdot Z_{(0)}\}$ as $\mathrm{i}^{\{1\}} \cdot \{\mathbb{I}, Z\} \otimes \{\mathbb{I}\}$.

We now explain how each operation in the circuit modifies this abstract state.

CLIFFORD GATE APPLICATION First, the circuit applies one Hadamard gate H to each qubit. This corresponds to the unitary operator $H_{(0)}H_{(1)}$, yielding updated abstract density matrix $\boldsymbol{\rho_B} = (H_{(0)}H_{(1)})\boldsymbol{\rho_A}(H_{(0)}H_{(1)})^\dagger$. Just as for concrete density matrices (see §5.2), this amounts to replacing

$$\begin{aligned}
\{\mathbb{I}\} &\text{ by } (H_{(0)}H_{(1)})\{\mathbb{I}\}(H_{(0)}H_{(1)})^\dagger = \{\mathbb{I}\},\\
Z_{(0)} &\text{ by } (H_{(0)}H_{(1)})Z_{(0)}(H_{(0)}H_{(1)})^\dagger = X_{(0)}, \text{ and}\\
Z_{(1)} &\text{ by } (H_{(0)}H_{(1)})Z_{(1)}(H_{(0)}H_{(1)})^\dagger = X_{(1)}.
\end{aligned}$$

We hence get $\boldsymbol{\rho_B} = e^{[0,0]+[0,0]\mathrm{i}}\{\mathbb{I}\}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}$.

NON CLIFFORD GATE APPLICATION Next, the circuit applies gate T on the upper qubit. To this end, we again follow the simulation described in §5.2. We first decompose T into Pauli elements: $T_{(0)} = d_1\mathbb{I} + d_2 Z_{(0)}$, where $d_1 \approx e^{-0.1+0.4\mathrm{i}}$ and $d_2 \approx e^{-1.0-1.2\mathrm{i}}$. Replacing T with its decomposition, we can then write $\boldsymbol{\rho_T} = T\boldsymbol{\rho_B}T^\dagger$, using Eq. (5.6), as:

$$\begin{aligned}
\boldsymbol{\rho_T} = \left(d_1\mathbb{I} + d_2 Z_{(0)}\right)&\left(e^{[0,0]+[0,0]\mathrm{i}}\{\mathbb{I}\}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}\right)\\
&\left(d_1\mathbb{I} + d_2 Z_{(0)}\right)^\dagger.
\end{aligned}$$

Analogously to §5.2, we can rewrite this to:

$$\begin{aligned}
&\boldsymbol{c_1}\{\mathbb{I}\}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}\\
&+\boldsymbol{c_2}\{Z_{(0)}\}\frac{\mathbb{I}+(-1)^{\{1\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}\\
&+\boldsymbol{c_3}\{Z_{(0)}\}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}\\
&+\boldsymbol{c_4}\{\mathbb{I}\}\frac{\mathbb{I}+(-1)^{\{1\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2},
\end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{c_1} &= d_1 e^{[0,0]+[0,0]\mathrm{i}} d_1^* \approx e^{[-0.2,-0.2]+[0,0]\mathrm{i}},\\
\boldsymbol{c_2} &= d_1 e^{[0,0]+[0,0]\mathrm{i}} d_2^* \approx e^{[-1.1,-1.1]+[1.6,1.6]\mathrm{i}},\\
\boldsymbol{c_3} &= d_2 e^{[0,0]+[0,0]\mathrm{i}} d_1^* \approx e^{[-1.1,-1.1]+[-1.6,-1.6]\mathrm{i}},\\
\boldsymbol{c_4} &= d_2 e^{[0,0]+[0,0]\mathrm{i}} d_2^* \approx e^{[-2.0,-2.0]+[0,0]\mathrm{i}}.
\end{aligned}$$

MERGING SUMMANDS Unfortunately, simply applying T gates as shown above may thus increase the number of summands in the abstract density matrix by a factor of 4. To counteract this, our key idea is to merge summands, by allowing a single abstract summand to represent multiple concrete ones, resulting in reduced computation overhead at the cost of lost precision. Our abstract representation allows for a straightforward merge: we take the union of sets and join intervals. Specifically, for complex numbers, we join the intervals in their representation, obtaining:

$$\boldsymbol{c} := \boldsymbol{c_1} \sqcup \boldsymbol{c_2} \sqcup \boldsymbol{c_3} \sqcup \boldsymbol{c_4} = e^{[-2.0,-0.2]+[-1.6,1.6]\mathrm{i}}.$$

Finally, we introduce the symbol $\star$ to denote how many concrete summands an abstract summand represents. Altogether, merging the summands in $\boldsymbol{\rho_T}$ yields:

$$\boldsymbol{\rho_C} = 4 \star e^{[-2.0,-0.2]+[-1.6,1.6]\mathrm{i}}\{\mathbb{I}, Z_{(0)}\}\frac{\mathbb{I}+(-1)^{\{0,1\}}X_{(0)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(1)}}{2}.$$

Note that for an abstract element $\boldsymbol{x}$, $r \star \boldsymbol{x}$ is not equivalent to $r \cdot \boldsymbol{x}$. For example, $2 \star \{0,1\} = \{0,1\} + \{0,1\} = \{0,1,2\}$, while $2 \cdot \{0,1\} = \{0,2\}$. [3]

3 We implicitly lift concrete elements to abstract elements: $2 \cdot \{0,1\} = \{2\} \cdot \{0,1\} = \{0,2\}$.

MEASUREMENT After the T gate, the circuit applies two additional $CNOT$ gates, resulting in the updated density matrix:

$$\boldsymbol{\rho_D} = 4 \star e^{[-2.0,-0.2]+[-1.6,1.6]i}\{\mathbb{I}, Z_{(0)}\}\frac{\mathbb{I}+(-1)^{\{0,1\}}X_{(0)}X_{(1)}}{2}\frac{\mathbb{I}+(-1)^{\{0\}}X_{(0)}}{2}.$$

Finally, the circuit applies the projection $M_- = \frac{\mathbb{I}-X_{(0)}}{2}$. To update the density matrix accordingly, we closely follow [36], which showed that measurement can be reduced to simple state updates through a case distinction on M_- and the state ρ. If (i) the measurement Pauli (here $-X_{(0)}$) commutes with the product Paulis (here $(-1)^{\{0,1\}}X_{(0)}X_{(1)}$ and $(-1)^{\{0\}}X_{(1)}$) and (ii) the measurement Pauli cannot be written as a product of the product Paulis, the density matrix after measurement is 0. We will explain in §5.5.2 how our abstract domain allows both of these checks to be performed efficiently.

Here, both conditions are satisfied, and we hence get the final state $\boldsymbol{\rho_{M1}} = 0$. We can then compute the probability of such an outcome by $p = \operatorname{tr}(\boldsymbol{\rho_{M1}}) = 0$. Thus, our abstract representation was able to provide a fully precise result.

IMPRECISE MEASUREMENT Suppose now that instead of the measurement in Fig. 5.1, we had collapsed the lower qubit to $|0\rangle$ by applying projection $M_0 = \frac{\mathbb{I}+Z_{(1)}}{2}$.

To derive the resulting state, we again follow [36] closely. We note that the measurement Pauli $+Z_{(1)}$ (i) anticommutes with the first product Pauli $(-1)^{\{0,1\}}X_{(0)}X_{(1)}$ and commutes with the second one $(-1)^{\{0\}}X_{(0)}$ and (ii) commutes with the initial Paulis $\{\mathbb{I}, Z_{(0)}\}$. In this case, we get that the density matrix is unchanged, thus $\boldsymbol{\rho_{M2}} = \boldsymbol{\rho_D}$. To compute the trace of this matrix, we follow the procedure outlined in §5.5.4. We omit intermediate steps here and get: [4]

$$p = \operatorname{tr}(\boldsymbol{\rho_{M2}}) = 4\Re(\boldsymbol{c}) \approx [0, 1.7].$$

Thus, our abstraction here is highly imprecise and does not yield any information on the measurement result (we already knew that the probability must lie in $[0, 1]$).

4 We used the precise interval bounds for $\boldsymbol{c}$ here, not the rounded values provided earlier.

Table 5.2: Example elements on abstract domains.

Dom.	Example element	Concretization
$\boldsymbol{B}$	$\{0,1\}$	$\{0,1\}$
$\boldsymbol{Z}_4$	$\{0,3\}$	$\{0,3\}$
$\boldsymbol{R}$	$(0,1)$	$[0,1]=\{r \mid 0 \leq r \leq 1\}$
$\boldsymbol{C}$	$(0,1,\pi,2\pi)$	$e^{[0,1]+[\pi,2\pi]\mathrm{i}}$ $=\{e^{r+\varphi \mathrm{i}} \mid 0 \leq r \leq 1, \pi \leq \varphi \leq 2\pi\}$
$\boldsymbol{P}_2$	$(\{0,3\},\{Z,Y\},\{X\})$	$\mathrm{i}^{\{0,3\}} \cdot \{Z,Y\} \otimes \{X\}$ $=\left\{\mathrm{i}^b \cdot P^{(1)} \otimes P^{(2)} \;\middle|\; \begin{array}{l} b \in \{0,3\}, \\ P^{(1)} \in \{Z,Y\}, P^{(2)} \in \{X\} \end{array}\right\}$

5.4 ABSTRACT DOMAINS

In the following, we formalize all abstract domains (Table 5.2) underlying our abstract representation of density matrices $\boldsymbol{\rho}$ along with key abstract transformers operating on them (Table 5.3). We note that all abstract transformers introduced here naturally also support (partially) concrete arguments.

EXAMPLE ELEMENTS Table 5.2 provides an example element $\boldsymbol{x}$ of each abstract domain, along with an example of its concretization $\gamma(\boldsymbol{x})$, where $\gamma\colon \boldsymbol{\mathcal{X}} \to 2^{\mathcal{X}}$. While Table 5.2 correctly distinguishes abstract elements from their concretization, in the following, when describing operators we write concretizations instead of abstract elements (as announced in §5.2).

BOOLEANS AND $\mathbb{Z}_4$ Abstract booleans $\boldsymbol{b} \in \boldsymbol{B} = 2^{\mathbb{B}}$ are subsets of $\mathbb{B}$, as exemplified in Table 5.2. The addition of two abstract booleans naturally lifts boolean addition to sets and is clearly sound:

$$\boldsymbol{b}+\boldsymbol{c}=\{b+c \mid b \in \boldsymbol{b}, c \in \boldsymbol{c}\}. \tag{5.8}$$

We define multiplication of abstract booleans analogously. Further, we define the join of two abstract booleans as their set union.

Analogously to booleans, our abstract domain $\boldsymbol{Z}_4$ consists of subsets of $\mathbb{Z}_4$, where addition, subtraction, multiplication, and joins works analogously to abstract booleans. Further, we can straight-forwardly embed abstract booleans into $\boldsymbol{Z}_4$ by mapping 0 to 0 and 1 to 1.

Table 5.3: Summary of abstract transformers.

Transformers	**Domains**	**Definition**
$\boldsymbol{b}+\boldsymbol{c} \in \boldsymbol{B}, \boldsymbol{b} \cdot \boldsymbol{c} \in \boldsymbol{B}$	$\boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{B}$	Lifting to sets, Eq. (5.8)
$\boldsymbol{b} \sqcup \boldsymbol{c} \in \boldsymbol{B}$	$\boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{B}$	$\boldsymbol{b} \cup \boldsymbol{c}$
$\boldsymbol{b}+\boldsymbol{c} \in \boldsymbol{Z_4}, \boldsymbol{b}-\boldsymbol{c} \in \boldsymbol{Z_4}, \boldsymbol{b} \cdot \boldsymbol{c} \in \boldsymbol{Z_4}$	$\boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{Z_4}$	Lifting to sets
$\boldsymbol{b} \sqcup \boldsymbol{c} \in \boldsymbol{Z_4}$	$\boldsymbol{b}, \boldsymbol{c} \in \boldsymbol{Z_4}$	$\boldsymbol{b} \cup \boldsymbol{c}$
$\boldsymbol{b} \in \boldsymbol{Z_4}$	$\boldsymbol{b} \in \boldsymbol{B}$	Embedding
$\boldsymbol{c} \cdot \boldsymbol{d} \in \boldsymbol{C}$	$\boldsymbol{c}, \boldsymbol{d} \in \boldsymbol{C}$	Eq. (5.9)
$\boldsymbol{c} \sqcup \boldsymbol{d} \in \boldsymbol{C}$	$\boldsymbol{c}, \boldsymbol{d} \in \boldsymbol{C}$	Eq. (5.10)
$\Re(\boldsymbol{c}) \in \boldsymbol{R}$	$\boldsymbol{c} \in \boldsymbol{C}^n$	Eq. (5.11)
$\mathrm{i}^{\boldsymbol{b}} \in \boldsymbol{C}$	$\boldsymbol{b} \in \boldsymbol{B}$	Eq. (5.12)
$\boldsymbol{PQ} \in \boldsymbol{P}_n$	$\boldsymbol{P}, \boldsymbol{Q} \in \boldsymbol{P}_n$	Eq. (5.13)
$\mathsf{f}(\boldsymbol{PQ}) \in \boldsymbol{Z_4}$	$\boldsymbol{P}, \boldsymbol{Q} \in \boldsymbol{P}_n$	Eq. (5.14)
$U_{(i)}\boldsymbol{P}U_{(i)}^{\dagger} \in \boldsymbol{P}_n$	$U \in \mathcal{U}(2^k), \boldsymbol{P} \in \boldsymbol{P}_n$	Eq. (5.15)
$\boldsymbol{P} \diamond \boldsymbol{Q} \in \boldsymbol{B}$	$\boldsymbol{P}, \boldsymbol{Q} \in \boldsymbol{P}_n$	Eq. (5.16)
$\boldsymbol{P} \sqcup \boldsymbol{Q} \in \boldsymbol{P}_n$	$\boldsymbol{P}, \boldsymbol{Q} \in \boldsymbol{P}_n$	Eq. (5.17)
$(-1)^{\boldsymbol{b}} \cdot \boldsymbol{P}$	$\boldsymbol{b} \in \boldsymbol{B}, \boldsymbol{P} \in \boldsymbol{P}_n$	Eq. (5.18)

REAL NUMBERS We abstract real numbers by intervals of the form $[\underline{a}, \overline{a}] \subseteq \mathbb{R} \cup \{\pm\infty\}$, and denote the set of such intervals by $\boldsymbol{R}$. Here, $\underline{a}$ and $\overline{a}$ indicate the lower and upper bounds of the interval, respectively. Interval addition, interval multiplication, and the cosine and exponential transformer on intervals are defined in their standard way, see §5.2.

COMPLEX NUMBERS We parametrize complex numbers $c \in \mathbb{C}$ in polar coordinates (with magnitude in log-space), as $c = e^{r+\varphi\mathrm{i}}$ for $r, \varphi \in \mathbb{R}$. For example, we parametrize 0 as $e^{-\infty+0\mathrm{i}}$.

Based on this parametrization, we abstract complex numbers using two real intervals for r and φ respectively, as exemplified in Table 5.2. Formally, we interpret $\boldsymbol{c} \in \boldsymbol{C}$ as the set of all possible outcomes when instantiating both intervals:

$$\gamma(\boldsymbol{c}) = e^{[\underline{r},\overline{r}]+[\underline{\varphi},\overline{\varphi}]\mathrm{i}} = \left\{ e^{r+\varphi\mathrm{i}} \,\middle|\, r \in [\underline{r},\overline{r}], \varphi \in [\underline{\varphi},\overline{\varphi}] \right\}.$$

We can compute the multiplication and join of two abstract complex numbers $\boldsymbol{c} = e^{[\underline{r},\overline{r}]+[\underline{\varphi},\overline{\varphi}]\mathrm{i}}$ and $\boldsymbol{c}' = e^{[\underline{r}',\overline{r}']+[\underline{\varphi}',\overline{\varphi}']\mathrm{i}}$ as

$$\boldsymbol{c} \cdot \boldsymbol{c}' = e^{[\underline{r}+\underline{r}',\overline{r}+\overline{r}']+[\underline{\varphi}+\underline{\varphi}',\overline{\varphi}+\overline{\varphi}']\mathrm{i}} \text{ and} \tag{5.9}$$

$$\boldsymbol{c} \sqcup \boldsymbol{c}' = e^{[\min(\underline{r},\underline{r}'),\max(\overline{r},\overline{r}')]+[\min(\underline{\varphi},\underline{\varphi}'),\max(\overline{\varphi},\overline{\varphi}')]\mathrm{i}}. \tag{5.10}$$

Again, simple arithmetic shows that Eqs. (5.9)–(5.10) are sound. We note that to increase precision, we could map complex numbers to a canonical representation before joining them, by exploiting $e^{r+\phi\mathrm{i}} = e^{r+(\phi+2\pi)\mathrm{i}}$ to ensure that $\underline{\varphi}$ lies in $[0, 2\pi]$.

We compute the real part of an abstract complex number $\boldsymbol{c} = e^{[\underline{r},\overline{r}]+[\underline{\varphi},\overline{\varphi}]\mathrm{i}}$ as

$$\Re(\boldsymbol{c}) = \exp([\underline{r}, \overline{r}]) \cdot \cos([\underline{\varphi}, \overline{\varphi}]), \tag{5.11}$$

where we rely on interval transformers to evaluate the right-hand side. The soundness of Eq. (5.11) follows from the standard formula to extract the real part from a complex number in polar coordinates. We will later use Eq. (5.11) to compute tr $(\boldsymbol{\rho})$. To this end, we also need the transformer

$$\mathrm{i}^{\boldsymbol{b}} = \bigsqcup_{b \in \boldsymbol{b}} \{\mathrm{i}^b\} \in \boldsymbol{C}. \tag{5.12}$$

PAULI ELEMENTS Recall that a Pauli element $P \in \mathcal{P}_n$ has the form $P = \mathrm{i}^v \cdot P^{(0)} \otimes \cdots \otimes P^{(n-1)}$, for v in $\mathbb{Z}_4$ and $P^{(k)} \in \{\mathbb{I}, X, Y, Z\}$. We therefore parametrize P as a prefactor v (in $\log_\mathrm{i}$ space) and n bare Paulis $P^{(k)}$.

Accordingly, we parametrize abstract Pauli elements $\boldsymbol{P} \in \boldsymbol{P}_n$ as $\mathrm{i}^{\boldsymbol{v}} \cdot \boldsymbol{P}^{(0)} \otimes \cdots \otimes \boldsymbol{P}^{(n-1)}$, where $\boldsymbol{v} \in \boldsymbol{Z}_4$ is a set of possible prefactors and $\boldsymbol{P}^{(k)} \subseteq \{X, Y, Z, \mathbb{I}_2\}$ are sets of possible Pauli matrices. Formally, we interpret $\boldsymbol{P}$ as the set of all possible outcomes when instantiating all sets:

$$\gamma(\boldsymbol{P}) = \left\{ \mathrm{i}^v \cdot \bigotimes_{i=0}^{n-1} P^{(i)} \,\middle|\, v \in \boldsymbol{v}, P^{(i)} \in \boldsymbol{P}^{(i)} \right\}.$$

We define the product of two abstract Pauli elements as:

$$\boldsymbol{P}\boldsymbol{Q} = \mathrm{i}^{\mathsf{f}(\boldsymbol{P}\boldsymbol{Q})} \bigotimes_{i=0}^{n-1} \mathfrak{b}\Big(\boldsymbol{P}^{(i)} \boldsymbol{Q}^{(i)}\Big). \tag{5.13}$$

To this end, we evaluate the prefactor induced by multiplying Paulis as

$$\mathfrak{f}(\boldsymbol{P}\boldsymbol{Q}) = \mathfrak{f}(\boldsymbol{P}) + \mathfrak{f}(\boldsymbol{Q}) + \sum_{i=1}^{n} \mathfrak{f}(\boldsymbol{P}^{(i)}\boldsymbol{Q}^{(i)}), \tag{5.14}$$

where we can evaluate the summands in the right-hand side of Eq. (5.14) by precomputing them for all possible sets of Pauli matrices $\boldsymbol{P}^{(i)}$ and $\boldsymbol{Q}^{(i)}$. Then, we compute the sum using Eq. (5.8). Analogously, we can evaluate $\mathfrak{b}\left(\boldsymbol{P}^{(i)}\boldsymbol{Q}^{(i)}\right)$ by precomputation. The soundness of Eq. (5.13) follows from applying the multiplication component-wise, and then separating out prefactors from bare Paulis.

We also define the conjugation of an abstract Pauli element $\boldsymbol{P}$ with k-qubit gate U padded to n qubits as:

$$\begin{aligned} U_{(i)}\boldsymbol{P}U_{(i)}^{\dagger} &= U_{(i)}\left(\mathrm{i}^{v} \cdot \boldsymbol{P}^{(0:i)} \otimes \boldsymbol{P}^{(i:i+k)} \otimes \boldsymbol{P}^{(i+k:n)}\right)U_{(i)}^{\dagger} \\ &= \mathrm{i}^{v+\mathfrak{f}(U\boldsymbol{P}^{(i:i+k)}U^{\dagger})} \cdot \boldsymbol{P}^{(0:i)} \otimes \mathfrak{b}(U\boldsymbol{P}^{(i:i+k)}U^{\dagger}) \otimes \boldsymbol{P}^{(i+k:n)}, \end{aligned} \tag{5.15}$$

where $\boldsymbol{P}^{(i:j)}$ denotes $\boldsymbol{P}^{(i)} \otimes \cdots \otimes \boldsymbol{P}^{(j-1)}$. Because k is typically small, and all possible gates U are known in advance, we can efficiently precompute $\mathfrak{f}(U\boldsymbol{P}^{(i:i+k)}U^{\dagger})$ and $\mathfrak{b}(U\boldsymbol{P}^{(i:i+k)}U^{\dagger})$. We note that this only works if the result of conjugation is indeed an (abstract) Pauli element—if not, this operation throws an error[5]. The soundness from Eq. (5.15) follows from applying U to qubits i through $i+k$, and then separating out prefactors from bare Paulis.

We define the commutator $\boldsymbol{P} \diamond \boldsymbol{Q}$ of two abstract Pauli elements $\boldsymbol{P}$ and $\boldsymbol{Q}$ as

$$\left(\mathrm{i}^{v} \cdot \bigotimes_{i=0}^{n-1} \boldsymbol{P}^{(i)}\right) \diamond \left(\mathrm{i}^{w} \cdot \bigotimes_{i=0}^{n-1} \boldsymbol{Q}^{(i)}\right) = \sum_{i=1}^{n} \boldsymbol{P}^{(i)} \diamond \boldsymbol{Q}^{(i)}. \tag{5.16}$$

Here, we evaluate the sum using Eq. (5.8), and efficiently evaluate $\boldsymbol{P}^{(i)} \diamond \boldsymbol{Q}^{(i)} \in \boldsymbol{B}$ by precomputing:

$$\boldsymbol{P}^{(i)} \diamond \boldsymbol{Q}^{(i)} = \left\{ P^{(i)} \diamond Q^{(i)} \;\middle|\; P^{(i)} \in \boldsymbol{P}^{(i)}, Q^{(i)} \in \boldsymbol{Q}^{(i)} \right\}.$$

5 We can recover from this error by decomposing U as a sum of bare Pauli elements, as mentioned in §5.2, see also Eqs. (5.21)–(5.22).

The soundness of Eq. (5.16) can be derived from the corresponding concrete equation, which can be verified using standard linear algebra.

We define the join of abstract Pauli elements as

$$\left(\mathrm{i}^{\boldsymbol{v}} \bigotimes_{i=0}^{n-1} \boldsymbol{P}^{(i)}\right) \sqcup \left(\mathrm{i}^{\boldsymbol{w}} \bigotimes_{i=0}^{n-1} \boldsymbol{Q}^{(i)}\right) = \mathrm{i}^{\boldsymbol{v} \sqcup \boldsymbol{w}} \bigotimes_{i=0}^{n-1} \left(\boldsymbol{P}^{(i)} \cup \boldsymbol{Q}^{(i)}\right), \tag{5.17}$$

where $\boldsymbol{P}^{(i)} \cup \boldsymbol{Q}^{(i)} \subseteq \{\mathbb{I}, X, Y, Z\}$. Clearly, this join is sound.

Finally, we define an abstract transformer for modifying the sign of an abstract Pauli element $\boldsymbol{P}$ by:

$$(-1)^{\boldsymbol{b}} \cdot \left(\mathrm{i}^{\boldsymbol{v}} \cdot \bigotimes_{i=0}^{n-1} \boldsymbol{P}^{(i)}\right) = \mathrm{i}^{\boldsymbol{v}+2\cdot\boldsymbol{b}} \cdot \bigotimes_{i=0}^{n-1} \boldsymbol{P}^{(i)} \tag{5.18}$$

The soundness of Eq. (5.18) follows directly from $(-1)^v = \mathrm{i}^{2v}$.

ABSTRACT DENSITY MATRICES The concrete and abstract domains introduced previously allow us to represent an abstract density matrix $\boldsymbol{\rho} \in \boldsymbol{D}$ as follows:

$$\boldsymbol{\rho} = r \star \boldsymbol{c} \cdot \boldsymbol{P} \cdot \prod_{j=1}^{n} \tfrac{\mathbb{I}+(-1)^{\boldsymbol{b}_j} Q_j}{2}. \tag{5.19}$$

Here, $r \in \mathbb{N}$, $\boldsymbol{c} \in \boldsymbol{C}$, $\boldsymbol{P} \in \boldsymbol{P}_n$, $\boldsymbol{b}_j \in \boldsymbol{B}$, and $Q_j \in \mathcal{P}_n$. Note that Q_j are concrete Pauli elements, while $\boldsymbol{P}$ is abstract. Further, both $\boldsymbol{P}$ and Q_j can have a prefactor, i.e., are not necessarily bare Paulis. Here, the integer counter r records how many concrete summands were abstracted. Specifically, $r \star \boldsymbol{x}$ is defined as $\sum_{i=1}^{r} \boldsymbol{x}$. Overall, we interpret $\boldsymbol{\rho}$ as:

$$\gamma(\boldsymbol{\rho}) = \left\{ \sum_{i=1}^{r} c_i P_i \prod_{j=1}^{n} \tfrac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2} \;\middle|\; c_i \in \gamma(\boldsymbol{c}), P_i \in \gamma(\boldsymbol{P}), b_{ij} \in \gamma(\boldsymbol{b}_j) \right\}, \tag{5.20}$$

relying on the previously discussed interpretations of $\mathbb{C}$, $\mathcal{P}_n$, and $\mathbb{B}$.

5.5 ABSTRACT TRANSFORMERS

We now formalize the abstract transformers used by ABSTRAQT to simulate quantum circuits. The soundness of all transformers is straightforward, except for the trace transformer (§5.5.4) which we discuss in App. C.1.

INITIALIZATION We start from initial state $\otimes_{i=1}^{n} |0\rangle$, which corresponds to density matrix

$$\boldsymbol{\rho} = \prod_{j=1}^{n} \frac{\mathbb{I}+Z_{(j)}}{2} = 1 \star e^{[0,0]+\mathrm{i}[0,0]} \cdot \mathrm{i}^{\{0\}}\{\mathbb{I}\} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\{0\}}Z_{(j)}}{2},$$

as established in [36, Sec. III]. We note that we can prepare other starting states by applying appropriate gates to the starting state $\otimes_{i=1}^{n} |0\rangle$.

5.5.1 *Gate Application*

Analogously to the concrete case discussed in §5.2, applying a unitary gate U to $\boldsymbol{\rho}$ yields:

$$U\boldsymbol{\rho}U^{\dagger} = r \star \boldsymbol{cP'} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_j}Q'_j}{2}, \tag{5.21}$$

for $\boldsymbol{P'} = U\boldsymbol{P}U^{\dagger}$ and $Q'_j = UQ_jU^{\dagger}$.

If either $U\boldsymbol{P}U^{\dagger} \not\subseteq \mathcal{P}_n$ or $UQ_jU^{\dagger} \not\subseteq \mathcal{P}_n$, Eq. (5.21) still holds, but we cannot represent the resulting matrices efficiently. In this case, again analogously to §5.2, we instead decompose the offending gate as $U = \sum_p d_pR_p$, with $R_p \in \mathcal{P}_n$ and obtain

$$U\boldsymbol{\rho}U^{\dagger} = \sum_{pq} r \star \boldsymbol{c_{pq}P_{pq}} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_{jq}}Q_j}{2}, \tag{5.22}$$

for $\boldsymbol{c}_{pq} = d_p\boldsymbol{c}d_q^{*}$, $\boldsymbol{P}'_{pq} = R_p\boldsymbol{P}R_q$, and $\boldsymbol{b_{jq}} = \boldsymbol{b_j} + Q_j \diamond R_q$.

Overall, we can evaluate Eqs. (5.21)–(5.22) by relying on the abstract transformers from §5.4.

COMPRESSION To prevent an exponential blow-up of the number of summands and to adhere to the abstract domain of $\boldsymbol{\rho}$ which does not include a sum, we compress all summands to a single one. Two summands can be joined as follows:

$$\left(r_1 \star \boldsymbol{c}_1\boldsymbol{P}_1 \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_{1j}}Q_j}{2}\right) \sqcup \left(r_2 \star \boldsymbol{c}_2\boldsymbol{P}_2 \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_{2j}}Q_j}{2}\right)$$
$$= r \star \boldsymbol{cP} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_j}Q_j}{2},$$

where $r = r_1 + r_2$, $\boldsymbol{c} = \boldsymbol{c}_1 \sqcup \boldsymbol{c}_2$, $\boldsymbol{b}_j = \boldsymbol{b}_{1j} \sqcup \boldsymbol{b}_{2j}$, and $\boldsymbol{P} = \boldsymbol{P}_1 \sqcup \boldsymbol{P}_2$. The key observation here is that the concrete Q_j are independent of the summand, and thus need not be joined.

We note that we could also only merge *some* summands and leave the others precise—investigating the effect of more flexible merging strategies could be interesting future research.

5.5.2 *Measurement*

We now describe how to perform Pauli measurements, by extending the (concrete) stabilizer simulation to abstract density matrices. The correctness of the concrete simulation was previously established in [36, Sec. VII.C], while the correctness of the abstraction is immediate.

SIMULATING MEASUREMENT Applying a Pauli measurement in basis $R \in \mathfrak{b}(\mathcal{P}_n)$ has a probabilistic outcome and transforms ρ to $\rho_+ = \frac{\mathbb{I}+R}{2}\rho\frac{\mathbb{I}+R}{2}$ with probability $\mathrm{tr}(\rho_+)$ or $\rho_- = \frac{\mathbb{I}-R}{2}\rho\frac{\mathbb{I}-R}{2}$ with probability $\mathrm{tr}(\rho_-)$. We describe how to compute ρ_+. Computing ρ_- works analogously by using $-R$ instead of R.

In the following, we will consider a concrete state ρ as defined in §5.2 and an abstract state $\boldsymbol{\rho}$ as defined in Eq. (5.19):

$$\rho = \sum_{i=1}^{m} c_i P_i \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2} \quad \text{and} \quad \boldsymbol{\rho} = r \star \boldsymbol{c}\boldsymbol{P} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_j} Q_j}{2}. \tag{5.23}$$

Concrete simulation of measurement distinguishes two cases: either (i) R commutes with all Q_j or (ii) R anti-commutes with at least one Q_j. Note that as the Q_j are concrete in an abstract state $\boldsymbol{\rho}$, those two cases translate directly to the abstract setting. We now describe both cases for concrete and abstract simulation.

BACKGROUND: CONCRETE CASE (I) In this case, we assume R commutes with all Q_j. Focusing on a single summand ρ_i of ρ, measurement maps it to (see [36]):

$$\rho_{i,+} = c_i \frac{\mathbb{I}+R}{2} P_i \frac{\mathbb{I}+R}{2} \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{b_{ij}} Q_j}{2}. \tag{5.24}$$

Let us first introduce the notation $\{(-1)^{b_{ij}} Q_j\} \rightsquigarrow R$, denoting that R can be written as a product of selected Pauli elements from $\{(-1)^{b_{ij}} Q_j\}$.

Symmetrically, we write $\{(-1)^{b_{ij}}Q_j\} \not\rightsquigarrow R$ if R cannot be written as such a product. As shown in [36], if $\{(-1)^{b_{ij}}Q_j\} \rightsquigarrow R$ then $\frac{\mathbb{I}+R}{2}\prod_{j=1}^{n}\frac{\mathbb{I}+(-1)^{b_{ij}}Q_j}{2}$ is equal to $\prod_{j=1}^{n}\frac{\mathbb{I}+(-1)^{b_{ij}}Q_j}{2}$ and if $\{(-1)^{b_{ij}}Q_j\} \not\rightsquigarrow R$ then $\frac{\mathbb{I}+R}{2}\prod_{j=1}^{n}\frac{\mathbb{I}+(-1)^{b_{ij}}Q_j}{2}$ is null. Further, using that $R^2 = \mathbb{I}$, we get from Eq. (5.24) that if P_i commutes with R, $\rho_{i,+}$ is equal to ρ_i, otherwise, P_i anti-commutes with R and $\rho_{i,+}$ is null. Putting it all together, we finally get:

$$\rho_+ = \sum_{i=1}^{m}\rho_{i,+} = \sum_{i=1}^{m}\begin{cases} c_iP_i\prod\limits_{j=1}^{n}\frac{\mathbb{I}+(-1)^{b_{ij}}Q_j}{2} & \text{if } \{(-1)^{b_{ij}}Q_j\} \rightsquigarrow R \text{ and } R\diamond P_i = 0,\\ 0 & \text{if } \{(-1)^{b_{ij}}Q_j\} \not\rightsquigarrow R \text{ or } R\diamond P_i = 1.\end{cases} \tag{5.25}$$

ABSTRACT CASE (I) Let us first define $\rightsquigarrow^u$ and $\not\rightsquigarrow^u$ for a concrete R, concrete Q_j and abstract $\boldsymbol{b_j}$. We say $\{(-1)^{\boldsymbol{b_j}}Q_j\} \rightsquigarrow^u R$ if for all j, for all $b_j \in \gamma(\boldsymbol{b_j})$, we have $\{(-1)^{b_j}Q_j\} \rightsquigarrow R$. Similarly, we say $\{(-1)^{\boldsymbol{b_j}}Q_j\} \not\rightsquigarrow^u R$ if for all j, for all $b_j \in \gamma(\boldsymbol{b_j})$, we have $\{(-1)^{b_j}Q_j\} \not\rightsquigarrow R$. Note that $\rightsquigarrow^u$ and $\not\rightsquigarrow^u$ are under-approximations, and there can exist some R and $\{(-1)^{\boldsymbol{b_j}}Q_j\}$ such that neither apply. Using those two abstract relations, we get the abstract transformer for $\boldsymbol{\rho}_+$:

$$r \star \begin{cases} \boldsymbol{cP}\prod\limits_{j=1}^{n}\frac{\mathbb{I}+(-1)^{\boldsymbol{b_j}}Q_j}{2} & \text{if } \{(-1)^{\boldsymbol{b_j}}Q_j\} \rightsquigarrow^u R \text{ and } R\diamond \boldsymbol{P} = \{0\},\\ 0 & \text{if } \{(-1)^{\boldsymbol{b_j}}Q_j\} \not\rightsquigarrow^u R \text{ or } R\diamond \boldsymbol{P} = \{1\},\\ (\boldsymbol{c}\sqcup\{0\})\,\boldsymbol{P}\prod\limits_{j=1}^{n}\frac{\mathbb{I}+(-1)^{\boldsymbol{b_j}}Q_j}{2} & \text{otherwise.}\end{cases} \tag{5.26}$$

We can evaluate Eq. (5.26) by relying on the abstract transformers from Table 5.3 and by evaluating $\rightsquigarrow^u$ as discussed shortly.

BACKGROUND: CONCRETE CASE (II) We now suppose R anti-commutes with at least one Q_j. In this case, we can rewrite ρ such that R anti-commutes with Q_1, and commutes with all other Q_j. Specifically, we can select any Q_{j^*} which anti-commutes with R, swap b_{ij^*} and Q_{j^*} with b_{i1} and Q_1, and replace all other Q_j anti-commuting with R by Q_1Q_j (and analogously b_{ij}

by $b_{ij} + b_{i1}$), which leaves ρ invariant (see [36]). Assuming ρ is the result after this rewrite, we have:

$$\rho_+ = \sum_i \tfrac{1}{2} c_i P_i' \frac{\mathbb{I} + (-1)^0 R}{2} \prod_{j=2}^{n} \frac{\mathbb{I} + (-1)^{b_{ij}} Q_j}{2}, \tag{5.27}$$

$$\text{where } P_i' = \begin{cases} P_i & \text{if } R \diamond P_i = 0, \\ (-1)^{b_{i1}} P_i Q_1 & \text{if } R \diamond P_i = 1. \end{cases}$$

Overall, after rewriting ρ as above, Eq. (5.27) replaces c_i by $\frac{1}{2}c_i$, P_i by P_i', b_{i1} by 0, and Q_1 by R.

ABSTRACT CASE (II) In the abstract case, we first apply the same rewrite as in the concrete case, where we pick j^* as the first j for which Q_j anti-commutes with R. [6] Then, directly abstracting Eq. (5.27) yields:

$$\boldsymbol{\rho}_+ = r \star \tfrac{1}{2} \boldsymbol{c} \boldsymbol{P}' \frac{\mathbb{I} + (-1)^{\{0\}} R}{2} \prod_{j=2}^{n} \frac{\mathbb{I} + (-1)^{\boldsymbol{b}_j} Q_j}{2}, \tag{5.28}$$

$$\text{where } \boldsymbol{P}' = \begin{cases} \boldsymbol{P} & \text{if } R \diamond \boldsymbol{P} = \{0\}, \\ (-1)^{\boldsymbol{b}_1} \boldsymbol{P} Q_1 & \text{if } R \diamond \boldsymbol{P} = \{1\}, \\ \boldsymbol{P} \sqcup (-1)^{\boldsymbol{b}_1} \boldsymbol{P} Q_1 & \text{otherwise.} \end{cases}$$

Here, we replace $\boldsymbol{c}$ by $\frac{1}{2}\boldsymbol{c}$, $\boldsymbol{P}$ by $\boldsymbol{P}'$, $\boldsymbol{b}_1$ by $\{0\}$, and Q_1 by R. When defining $\boldsymbol{P}'$, we follow the two cases from Eq. (5.27) when our abstraction is precise enough to indicate which case we should choose, or join the results of both cases otherwise. Again, we can evaluate Eq. (5.28) by relying on the abstract transformers from Table 5.3.

JOINING BOTH MEASUREMENT RESULTS For measurements occurring within a quantum circuit, stabilizer simulation generally requires randomly selecting either ρ_+ or ρ_- with probability $\mathrm{tr}(\rho_+)$ and $\mathrm{tr}(\rho_-)$, respectively, and then continues only with the selected state. In contrast, ABSTRAQT can join both measurement outcomes into a single abstract state $\boldsymbol{\rho}_+ \sqcup \boldsymbol{\rho}_-$, as the Q_j are the same in both. This allows us to pursue both measurement outcomes simultaneously, as we demonstrate in §5.7.

6 We could also consider other strategies than picking the first possible j, for example picking a j for which $\boldsymbol{b}_j$ is precise whenever possible, to increase precision.

5.5.3 *Efficiently computing* $\rightsquigarrow$

To simulate the result of a measurement, we introduced the new operator $\{(-1)^{b_j}Q_j\} \rightsquigarrow R$, denoting that some Pauli R can be written as a product of $\{(-1)^{b_j}Q_j\}$. We now show how to compute $\rightsquigarrow$ efficiently.

BACKGROUND: CONCRETE CASE We first note that $\{(-1)^{b_j}Q_j\} \rightsquigarrow R$ holds if and only if there exist some $x \in \mathbb{B}^n$ such that:

$$R \overset{!}{=} \prod_{j=1}^{n} \left((-1)^{b_j}Q_j\right)^{x_j}. \tag{5.29}$$

Further, this solution x would satisfy:

$$\mathfrak{b}(R) \overset{!}{=} \mathfrak{b}\left(\prod_{j=1}^{n} \left((-1)^{b_j}Q_j\right)^{x_j}\right) \tag{5.30}$$

Eq. (5.30) has a solution if and only if R commutes with all the Q_j, in which case this solution x is unique (see [36]). Hence, to check if $\{(-1)^{b_j}Q_j\} \rightsquigarrow R$, we can first verify whether $R \diamond Q_j = 0$ for all j, and if so, check if the unique x satisfying Eq. (5.30) also satisfies Eq. (5.29).

BACKGROUND: FINDING x FOR Eq. (5.30) To compute this solution x, the stabilizer simulation relies critically on an isomorphism $\mathfrak{g}$ between Pauli matrices $\{\mathbb{I}, X, Y, Z\}$ and $\mathbb{B}^2$.

Specifically, $\mathfrak{g}$ maps I to $\binom{0}{0}$, X to $\binom{1}{0}$, Y to $\binom{1}{1}$, and Z to $\binom{0}{1}$. Further, $\mathfrak{g}$ extends naturally to bare Pauli elements $R \in \mathfrak{b}(\mathcal{P}_n)$ and tuples $Q = (Q_1, \dots, Q_n) \in \mathfrak{b}(\mathcal{P}_n)^n$ by:

$$\mathfrak{g}(R) = \begin{pmatrix} \mathfrak{g}(R^{(0)}) \\ \vdots \\ \mathfrak{g}(R^{(n-1)}) \end{pmatrix} \text{ and } \mathfrak{g}(Q) = \begin{pmatrix} \mathfrak{g}(Q_1^{(0)}) & \cdots & \mathfrak{g}(Q_n^{(0)}) \\ \vdots & \ddots & \vdots \\ \mathfrak{g}(Q_1^{(n-1)}) & \cdots & \mathfrak{g}(Q_n^{(n-1)}) \end{pmatrix},$$

where $\mathfrak{g}(R) \in \mathbb{B}^{2n\times 1}$ and $\mathfrak{g}(Q) \in \mathbb{B}^{2n\times n}$. We can naturally extend $\mathfrak{g}$ to $\mathcal{P}_n$, by defining $\mathfrak{g}(R) = \mathfrak{g}(\mathfrak{b}(R))$.

This isomorphism $\mathfrak{g}$ is designed so that the product of bare Pauli elements ignoring prefactors corresponds to a component-wise addition of encodings:

$$\mathfrak{g}(P_1P_2) = \mathfrak{g}(P_1) + \mathfrak{g}(P_2). \tag{5.31}$$

Using Eq. (5.31), we can obtain solution candidates x for Eq. (5.30) by solving a system of linear equations using Gaussian elimination modulo 2:

$$\mathfrak{g}(R) \overset{!}{=} \mathfrak{g}\left(\prod_{j=1}^{n} Q_j^{x_j}\right) = \sum_{j=1}^{n} \mathfrak{g}(Q_j)x_j = \mathfrak{g}(Q)x. \tag{5.32}$$

Because in our case, $\mathfrak{g}(Q)$ is over-determined and has full rank, Eq. (5.32) either has no solution, or a unique solution x.

BACKGROUND: CHECKING PREFACTORS Once we have found the unique x (if it exists) satisfying Eq. (5.30) as described above, we need to check if it also satisfies Eq. (5.29). It is enough to check if the prefactors match:

$$\mathfrak{f}(R) \overset{!}{=} \mathfrak{f}\left(\prod_j (-1)^{b_j x_j} Q_j^{x_j}\right),$$

or equivalently:

$$\mathfrak{f}(R) - \mathfrak{f}\left(\prod_j Q_j^{x_j}\right) - 2\sum_j b_j x_j \overset{!}{=} 0,$$

where the subtraction and sum operations are over $\mathbb{Z}_4$.

Putting it all together, we can define $\mathfrak{F}\colon \mathcal{P}_n \times \mathcal{P}_n^n \times \mathbb{B}^n \to \mathbb{Z}_4 \cup \{\lightning\}$ with

$$\mathfrak{F}(R,Q,b) = \begin{cases} \lightning & \text{if } \exists j, R \diamond Q_j = 1, \\ \mathfrak{f}(R) - \mathfrak{f}\left(\prod_{j=1}^{n} Q_j^{x_j}\right) - 2\sum_{j=1}^{n} x_j b_j & \text{otherwise,} \end{cases} \tag{5.33}$$

where x is the unique value such that $\mathfrak{g}(R) = \mathfrak{g}(Q)x$ and $\lightning$ indicates there is no such x. We then have that $\{(-1)^{b_j} Q_j\} \rightsquigarrow R$ if and only if $\mathfrak{F}(R,Q,b) = 0$, or equivalently, $\{(-1)^{b_j} Q_j\} \not\rightsquigarrow R$ if and only if $\mathfrak{F}(R,Q,b) \neq 0$.

$\mathfrak{F}$ FOR ABSTRACT $\boldsymbol{b_j}$ For abstract values $\boldsymbol{b_j}$, we define $\mathfrak{F}\colon \mathcal{P}_n \times \mathcal{P}_n^n \times \boldsymbol{B}^n \to 2^{\mathbb{Z}_4 \cup \{\lightning\}}$ as follows:

$$\mathfrak{F}(R,Q,\boldsymbol{b}) = \begin{cases} \{\lightning\} & \text{if } \exists j, R \diamond Q_j = 1, \\ \mathfrak{f}(R) - \mathfrak{f}\left(\prod_{j=1}^{n} Q_j^{x_j}\right) - 2\sum_{j=1}^{n} x_j \boldsymbol{b_j} & \text{otherwise.} \end{cases} \tag{5.34}$$

Following the same reasoning as above, we have $\{(-1)^{\boldsymbol{b}_j} Q_j\} \rightsquigarrow^u R$ if and only if $\mathfrak{S}(R, Q, \boldsymbol{b}) = \{0\}$ and $\{(-1)^{\boldsymbol{b}_j} Q_j\} \not\rightsquigarrow^u R$ if and only if $\mathfrak{S}(R, Q, \boldsymbol{b}) \cap \{0\} = \emptyset$.

$\mathfrak{S}$ FOR ABSTRACT $\boldsymbol{b}_j$ AND $\boldsymbol{R}$ To compute the trace of a state (see §5.5.4), we further extend Eq. (5.33) to abstract $\boldsymbol{b}_j$ and abstract $\boldsymbol{R}$, and define $\mathfrak{S}\colon \boldsymbol{P}_n \times \mathcal{P}_n^n \times \boldsymbol{B}^n \to 2^{\mathbb{Z}_4 \cup \{\lightning\}}$ as:

$$\mathfrak{S}(\boldsymbol{R}, Q, \boldsymbol{b}) = \begin{cases} \{\lightning\} & \text{if } \exists j. \boldsymbol{R} \diamond Q_j = \{1\}, \\ \mathfrak{f}(\boldsymbol{R}) - \mathfrak{f}\left(\prod_{j=1}^{n} Q_j^{\boldsymbol{x}_j}\right) - 2\sum_{j=1}^{n} \boldsymbol{x}_j \boldsymbol{b}_j & \text{if } \forall j. \boldsymbol{R} \diamond Q_j = \{0\}, \\ \mathfrak{f}(\boldsymbol{R}) - \mathfrak{f}\left(\prod_{j=1}^{n} Q_j^{\boldsymbol{x}_j}\right) - 2\sum_{j=1}^{n} \boldsymbol{x}_j \boldsymbol{b}_j \cup \{\lightning\} & \text{otherwise,} \end{cases} \tag{5.35}$$

$$\text{for } \mathfrak{g}(\boldsymbol{R}) = \mathfrak{g}(Q)\boldsymbol{x}. \tag{5.36}$$

Here, evaluating Eq. (5.35) requires evaluating $Q_j^{\boldsymbol{b}}$ for an abstract boolean $\boldsymbol{b}$, which we define naturally as

$$Q_j^{\boldsymbol{b}} := \begin{cases} \{Q_j\} & \text{if } \boldsymbol{b} = \{1\}, \\ \{\mathbb{I}\} & \text{if } \boldsymbol{b} = \{0\}, \\ \{Q_j, \mathbb{I}\} & \text{if } \boldsymbol{b} = \{0,1\}. \end{cases}$$

Further, Eq. (5.36) requires over-approximating all x which satisfy the linear equation $\mathfrak{g}(\boldsymbol{R}) = \mathfrak{g}(Q)x$. Here, we naturally extend $\mathfrak{g}$ to abstract Paulis by joining their images. For instance, we have that $\mathfrak{g}(\{X, Y\}) = \{\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)\} \sqcup \{\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\} = \left(\begin{smallmatrix}\{1\}\\\{0,1\}\end{smallmatrix}\right)$. We then view $\mathfrak{g}(\boldsymbol{R}) = \mathfrak{g}(Q)x$ as a system of linear equations $\boldsymbol{b} = Ax$, where the left-hand side consists of abstract booleans $\boldsymbol{b} \in \boldsymbol{B}^{2n}$. We then drop all equations in this equation system where the left-hand side is $\{0, 1\}$, as they do not constrain the solution space. This updated system is fully concrete, hence we can solve it using Gaussian elimination. We get either no solution, or a solution space $y + \sum_{k=1}^{p} \lambda_k u_k$, where y is a possible solution and $u_1, \ldots, u_p$ is a possibly empty basis of the null solution space. In the case of no solution, $\boldsymbol{x}$ is not needed in Eq. (5.35). Otherwise, we can compute $\boldsymbol{x}_j$ as $\{y_j + \sum_{k=1}^{m} \lambda_k u_{k,j} \mid \lambda_k \in \mathbb{B}\}$.

5.5.4 *Trace*

Recall that the probability of obtaining state ρ_+ when measuring ρ is $\operatorname{tr}(\rho_+)$. We now describe how to compute this trace using $\mathfrak{F}$ defined above.

BACKGROUND: CONCRETE TRACE Following [36], we compute the trace of a density matrix ρ by:

$$\operatorname{tr}(\rho) = \sum_{i=1}^{m} \Re\Big(c_i \mathrm{i}^{\mathfrak{F}(P,Q,b_i)}\Big), \tag{5.37}$$

where we define $\mathrm{i}^{\lightning} := 0$. Because the trace of a density matrix is always real, $\Re(\cdot)$ is redundant, but will be convenient to avoid complex traces in our abstraction.

ABSTRACT TRACE For an abstract state $\boldsymbol{\rho}$, we define:

$$\operatorname{tr}(\boldsymbol{\rho}) = r \cdot \Re\Big(\boldsymbol{c}\mathrm{i}^{\mathfrak{F}(\boldsymbol{P},Q,\boldsymbol{b})}\Big), \tag{5.38}$$

where we use $\mathfrak{F}(\cdot)$ as defined in Eq. (5.35).

5.6 IMPLEMENTATION

In the following, we discuss our implementation of the abstract transformers from §5.4 and §5.5 in ABSTRAQT.

LANGUAGE AND LIBRARIES We implemented ABSTRAQT in Python 3.8, relying on Qiskit 0.40.0 [39] for handling quantum circuits, and a combination of NumPy 1.20.0 [105] and Numba 0.54 [106] to handle matrix operations.

BIT ENCODINGS An abstract density matrix $\boldsymbol{\rho} = r \star \boldsymbol{c} \cdot \boldsymbol{P} \cdot \prod_{j=1}^{n} \frac{\mathbb{I}+(-1)^{\boldsymbol{b}_j} Q_j}{2}$ is encoded as a tuple $(r, \boldsymbol{c}, \boldsymbol{P}, \boldsymbol{b_1}, ..., \boldsymbol{b_n}, Q_1, \ldots, Q_n)$. To encode the concrete Pauli matrices Q_j, we follow concrete stabilizer simulation encodings such as [107] and encode Pauli matrices P using two bits $\mathfrak{g}(P)$ (see §5.5.3). To encode abstract elements of a finite set we use bit patterns. For example, we encode $\boldsymbol{b_1} = \{1, 0\} \in \mathcal{B}$ as 11_2, where the least significant bit (i.e. the right-most bit) indicates that $0 \in \boldsymbol{b_1}$. Analogously, we encode

$v = \{3,0\} \in \mathbf{Z_4}$ as 1001_2. Further, we encode $\{Z, Y\}$ as 1100_2, where the indicator bits correspond to Z, Y, X, and $\mathbb{I}$, respectively, from left to right. Hence the abstact Pauli $\boldsymbol{P} = (\{0,3\}, \{Z, Y\}, \{X\})$ would be represented as $(1001_2, 1100_2, 0010_2)$.

IMPLEMENTING TRANSFORMERS The abstract transformers on abstract density matrices can be implemented using operations in $\mathbf{B}, \mathbf{Z_4}, \mathbf{C}$, and $\mathbf{P_1}$. As $\mathbf{B}, \mathbf{Z_4}$, and $\mathbf{P_1}$ are small finite domains, we can implement operations in these domains using lookup tables, which avoids the need for bit manipulation tricks. While such tricks are applicable in our context (e.g., [36] uses bit manipulations to compute $H_{(i)} P H_{(i)}^\dagger$ for $P \in \mathcal{P}_n$), they are generally hard to come up with [108]. In contrast, the efficiency of our lookup tables is comparable to that of bit manipulation tricks, without requiring new insights for new operations.

For example, to evaluate $\{\} + \{0\}$ over $\mathbf{B}$ using Eq. (5.8), we encode the first argument $\{\}$ as 00 and the second argument $\{0\}$ as 01. Looking up entry $(00, 01)$ in a two-dimensional pre-computed table then yields 00, the encoding of the correct result $\{\}$. We note that we cannot implement this operation directly using a XOR instruction on encodings, as this would yield incorrect results: 00 XOR 01 $= 01 \simeq \{0\}$, which is incorrect.

GAUSSIAN ELIMINATION To efficiently solve equations modulo two as discussed in §5.5, we implemented a custom Gaussian elimination relying on bit-packing (i.e., storing 32 boolean values in a single 32-bit integer). In the future, it would be interesting to explore if Gaussian elimination could be avoided altogether, as suggested by previous works [36, 107].

TESTING To reduce the likelihood of implementation errors, we have complemented ABSTRAQT with extensive automated tests. We test that abstract transformers $f^\sharp$ are sound with respect to concrete functions f, that is to say that

$$\forall x_1 \in \gamma(\boldsymbol{x_1}) \cdots \forall x_k \in \gamma(\boldsymbol{x_k}). f(x_1, \ldots, x_n) \in f^\sharp(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}).$$

We check this inclusion for multiple selected samples of $\boldsymbol{x_i}$ and $x_i \in \boldsymbol{x_i}$ (typically corner cases).

This approach is highly effective at catching implementation errors, which we have found in multiple existing tools as shown in §5.7.

5.7 EVALUATION

We now present our evaluation of ABSTRAQT, demonstrating that it can establish circuit properties no existing tool can establish.

5.7.1 *Benchmarks*

To evaluate ABSTRAQT, we generated 12 benchmark circuits, summarized and visualized in Table 5.4.

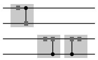
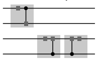
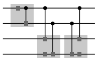
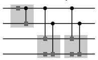
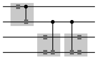
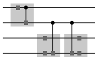
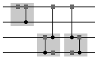
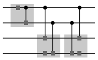
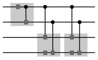
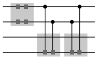
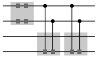
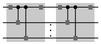
BENCHMARK CIRCUIT GENERATION Each circuit operates on 62 qubits, partitioned into 31 *upper* qubits and 31 *lower* qubits. We picked the limit of 62 qubits because our baseline ESS (discussed shortly) only supports up to 63 qubits; ABSTRAQT is not subject to such a limitation.

Each circuit operates on initial state $|0\rangle$ and is constructed to ensure that all lower qubits are eventually reverted to state $|0\rangle$. We chose this invariant as it can be expressed for most of the evaluated tools, as we will discuss in §5.7.2. Further, as some tools can only check this for one qubit at a time, we only check if the very last qubit is reverted to $|0\rangle$, instead of running 31 independent checks (which would artificially slow down some baselines). Note that this check is of equivalent difficulty for all lower qubits.

BENCHMARK DETAILS Table 5.4 details how each benchmark circuit was generated. Most of the circuits are built from three concatenated subcircuits. First, c_1 modifies the upper qubits, then c_2 modifies the lower qubits (potentially using gates controlled by the upper qubits) and finally c_3 reverts all lower qubits to $|0\rangle$, but in a non-trivial way. Circuit `CCX+H;Cliff` slightly deviates from this pattern, as it also modifies the upper qubits using gates controller by lower qubits. Further, circuits `Cliff+T;H;CZ+RX` and `Cliff+T;H;CZ+RX'` additionally apply two layers of H gates to the lower qubits. Finally, circuit `MeasureGHZ` applies internal measurements, as discussed below.

The majority of circuits revert the lower qubits to $|0\rangle$ by applying c_3, the inverse of c_2 but optimized using PyZX [109]—this obfuscates the fact that c_2 and c_3 cancel out. Four circuits, marked with a trailing prime ('), generate c_3 by optimizing the un-inverted c_2. They still reset all lower qubits to $|0\rangle$, but establishing this requires advanced reasoning. Specifically, `RZ`$_2$`+H;CX'`

TABLE 5.4: Description of benchmark circuits, where upper $= \{1, \ldots, 31\}$ and lower $= \{32, \ldots, 62\}$.

Circuit	**Generation**	**Gates** (approx.)
`Cliff;Cliff` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{o(q) \mid o \in \{H,S\}, q \in \text{lower}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	26k $\times$ Clifford
`Cliff+T;Cliff` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S,T\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{o(q) \mid o \in \{H,S\}, q \in \text{lower}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	23k $\times$ Clifford, 2.5k $\times$ T
`Cliff+T;CX+T` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S,T\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{T(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	18k $\times$ Clifford, 9k $\times$ T, 40 $\times$ $T^\dagger$
`Cliff+T;CX+T'` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S,T\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{T(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2)$	18k $\times$ Clifford, 9k $\times$ T, 40 $\times$ $T^\dagger$
`Cliff+T;H;CZ+RX` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S,T\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_h = H(32); \ldots; H(62)$ $c_2 \in \left(\{CZ(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{RX_{\frac{\pi}{4}}(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_h; c_2; \text{opt}(c_2^\dagger); c_h$	18k $\times$ Clifford, 5k $\times$ $RX_{\frac{\pi}{4}}$, 3k $\times$ T, 1k $\times$ $T^\dagger$
`Cliff+T;H;CZ+RX'` 	$c_1 \in \left(\{o(q) \mid o \in \{H,S,T\}, q \in \text{upper}\} \cup \{CX(q_1,q_2) \mid q_1,q_2 \in \text{upper}\}\right)^{10^4}$ $c_h = H(32); \ldots; H(62)$ $c_2 \in \left(\{CZ(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{RX_{\frac{\pi}{4}}(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_h; c_2; \text{opt}(c_2); c_h$	18k $\times$ Clifford, 5k $\times$ $RX_{\frac{\pi}{4}}$, 4k $\times$ T, 40 $\times$ $T^\dagger$
`CCX+H;Cliff` 	$c_1 \in \left(\{CCX(q_1,q_2,q_3) \mid q_1,q_2,q_3 \in \text{upper}\} \cup \{H(q) \mid q \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{o(q) \mid o \in \{H,S\}, q \in \text{lower}\} \cup \right.$ $\left.\{CX(q_1,q_2) \mid q_1 \in \text{lower}, q_2 \in \text{lower} \cup \text{upper}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	22k $\times$ Clifford, 5k $\times$ CCX
`CCX+H;CX+T` 	$c_1 \in \left(\{CCX(q_1,q_2,q_3) \mid q_1,q_2,q_3 \in \text{upper}\} \cup \{H(q) \mid q \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{T(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	16k $\times$ Clifford, 5k $\times$ CCX, 5k $\times$ T, 1k $\times$ $T^\dagger$
`CCX+H;CX+T'` 	$c_1 \in \left(\{CCX(q_1,q_2,q_3) \mid q_1,q_2,q_3 \in \text{upper}\} \cup \{H(q) \mid q \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\} \cup \{T(q) \mid q \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2)$	16k $\times$ Clifford, 5k $\times$ CCX, 7k $\times$ T, 30 $\times$ $T^\dagger$
`RZ`$_2$`+H;CX` 	$c_1 \in \left(\{o(q) \mid o \in \{RZ_2, H\}, q \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2^\dagger)$	16k $\times$ Clifford, 5k $\times$ RZ_2
`RZ`$_2$`+H;CX'` 	$c_1 \in \left(\{o(q) \mid o \in \{RZ_2, H\}, q \in \text{upper}\}\right)^{10^4}$ $c_2 \in \left(\{CX(q_1,q_2) \mid q_1 \in \text{upper}, q_2 \in \text{lower}\}\right)^{10^4}$ return $c_1; c_2; \text{opt}(c_2)$	16k $\times$ Clifford, 5k $\times$ RZ_2
`MeasureGHZ` 	$c_1 = CX(1,2); \ldots; CX(1,62)$ $c_2 = H(1); c_1; \text{measure}(1), c_1$ return $c_2; \ldots; c_2$ (100 times)	12k $\times$ Clifford, 100 $\times$ measure

flips each lower qubit an even number of times. [7] Similarly, `Cliff+T;CX+T'` and `CCX+H;CX+T'` additionally modify the phase but still flip each lower qubit an even number of times. Finally, `Cliff+T;H;CZ+RX'` flips between states $|+\rangle$ and $|-\rangle$ an even number of times, where $RX_{\frac{\pi}{4}}$ only modifies the phase.

The last benchmark `MeasureGHZ` first generates a GHZ state $\frac{1}{\sqrt{2}}|0\cdots0\rangle + \frac{1}{\sqrt{2}}|1\cdots1\rangle$, and collapses it to $|0\cdots0\rangle$ or $|1\cdots1\rangle$ by measuring the first qubit. Then, it resets all qubits to $|0\rangle$ except for the first one. It then repeats this process, with the first qubit starting in either $|0\rangle$ or $|1\rangle$. Thus, the state before measurement is either $\frac{1}{\sqrt{2}}|0\cdots0\rangle + \frac{1}{\sqrt{2}}|1\cdots1\rangle$ or $\frac{1}{\sqrt{2}}|0\cdots0\rangle - \frac{1}{\sqrt{2}}|1\cdots1\rangle$, but every repetition still resets all lower qubits to $|0\rangle$.

DISCUSSION Our benchmark covers a wide variety of gates, with all applying Clifford gates, seven applying T gates, three applying CCX gates, two applying $RX_{\frac{\pi}{4}}$ gates (one qubit gate, rotation around the X axis of $\frac{\pi}{4}$ radians), and two applying RZ_2 gates (one qubit gate, rotation around the Z axis of 2 radians).

All benchmarks are constructed to revert the lower qubits to $|0\rangle$, but in a non-obvious way. As fully precise simulation of most benchmarks is unrealistic, we expect that over-approximation is typically necessary to establish this fact.

5.7.2 *Baselines*

We now discuss how we instantiated existing tools to establish that a circuit c evolves a qubit q to state $|0\rangle$. Overall, we considered two tools based on stabilizer simulation (ESS [37] and QuiZX [38]), one tool based on the Feynman path integral (Feynman [110]), one tool based on abstract interpretation (YP21 [40], in two different modes), and one tool based on state vectors (Statevector as implemented by Qiskit [39]).

ESS Qiskit [39] provides an extended stabilizer simulator implementing the ideas published in [37] which (i) decomposes quantum circuits into Clifford circuits, (ii) simulates these circuits separately, and (iii) performs measurements by an aggregation across these circuits. To check if a circuit c consistently evolves a qubit q to $|0\rangle$, we check if c extended by a

7 More precisely, when representing the quantum state as a sum over computational basis states, an even number of flips are applied to each qubit of each summand.

measurement of q always yields 0. To run our simulation, we used default parameters.

QUIZX QuiZX [38] improves upon [37] by alternating between decomposing circuits (splitting non-Clifford gates into Clifford gates) and optimizing the decomposed circuits (which may further reduce non-Clifford gates). We can use QuiZX to establish that a qubit is in state $|0\rangle$ by "plugging" output q as $|1\rangle$ and establishing that the probability of this output is zero. [8]

FEYNMAN Feynman [110] allows to verify quantum circuits based on the Feynman path integral. Its implementation[9] supports two main use cases, namely optimization and checking the equivalence of two circuits. While these use cases cannot prove that a circuit resets a qubit to $|0\rangle$, we can use Feynman's equivalence check to check whether the circuits in Table 5.4 are equivalent to a simplified version which performs no operation at all on lower qubits. We check this equivalence for all circuits, even for those where we know it does not hold (namely all whose name ends with a prime), allowing us to confirm that Feynman cannot scale to any of our benchmarks (see §5.7.3).

We note that Feynman currently does not support internal measurements. [10]

YP21 Like ABSTRAQT, YP21 [40] also uses abstract interpretation, but relies on projectors instead of stabilizer simulation. Specifically, it encodes the abstract state of selected (small) subsets of qubits as projectors $\{P_j\}_{j \in \mathcal{J}}$, which constrain the state of these qubits to the range of P_j.

To check if a qubit q is in state $|0\rangle$, we check if the subspace resulting from intersecting the range of all P_j is a subset of the range of $\mathbb{I} + Z_{(q)}$—an operation which is natively supported by YP21.

When running YP21, we used the two execution modes suggested in its original evaluation [40]. The first mode tracks the state of all pairs of qubits, while the second considers subsets of 5 qubits that satisfy a particular condition (for details, see [40, §9]). Because [40] does not discuss which execution mode to pick for new circuits, we evaluated all circuits in both modes.

We note that because YP21 does not support $CX(a, b)$ for $a > b$, we instead encoded such gates as $H(a); H(b); CX(b, a); H(b); H(a)$.

8 The use of plugging is described on `https://github.com/Quantomatic/quizx/issues/9`.
9 Tool available at `https://github.com/meamy/feynman`
10 `https://github.com/meamy/feynman/issues/8`

STATEVECTOR Qiskit [39] further provides a simulator based on state vectors, which we also used for completeness.

ABSTRAQT In ABSTRAQT, we can establish that a qubit is in state $|0\rangle$ by measuring the final abstract state $\boldsymbol{\rho}$ in basis $Z_{(i)}$ and checking if the probability of obtaining $|1\rangle$ is 0.

EXPERIMENTAL SETUP We executed all experiments on a machine with 110 GB RAM and 56 cores at 2.6 GHz, running Ubuntu 22.04. Because some tools consumed excessive amounts of memory, we limited them to 12 GB of RAM. This was not necessary for ABSTRAQT, which never required more than 600 MB of RAM. We limited each tool to a single thread.

5.7.3 *Results*

Table 5.5 summarizes the results when using all tools discussed in §5.7.2 to establish that the last qubit in 10 randomly selected instantiations of each benchmark from Table 5.4 is in state $|0\rangle$. Overall, it demonstrates that while ABSTRAQT can establish this for all benchmarks within minutes, QuiZX can only establish it for a few instances, and all other tools cannot establish it for any benchmark. Further, we found that for some circuits the established simulation tool ESS yields incorrect results. We now discuss the results of each tool in more details.

MEASUREGHZ Importantly, no baseline tool except ABSTRAQT can simultaneously simulate both outcomes of a measurement, without incurring an exponential blow-up. Therefore, for `MeasureGHZ`, we consider internal measurements as an unsupported operation in these tools. We note that we could randomly select one measurement outcome and simulate the remainder of the circuit for it, but then we can only establish that the final state is $|0\rangle$ *for a given sequence of measurement outcomes*. In contrast, a single run of ABSTRAQT can establish that the final state is $|0\rangle$ for all possible measurement outcomes (see also §5.5.2).

QUIZX As QuiZX is the only baseline tool solving some of our benchmark instances, we provide a detailed comparison to it in Table 5.6.

Overall, QuiZX cannot consistently handle any of the benchmarks from Table 5.4, Instead, it often either times out or runs out of memory. Further, QuiZX consistently runs into an internal error when simulating

TABLE 5.5: Success rates when running simulators on benchmarks from Table 5.4.

Label	**Abstraqt**	**QuiZX**	**ESS**	**Feynman**	**YP21 (mode 1)**	**YP21 (mode 2)**	**Statevec.**
Cliff;Cliff	**100%**	0% (E)	0% (I)	0% (T,M)	0% (T,P)	0% (I)	0% (M)
Cliff+T;Cliff	**100%**	70% (T)	0% (I)	0% (T,M)	0% (T,P)	0% (I)	0% (M)
Cliff+T;CX+T	**100%**	80% (M)	0% (M)	0% (T)	0% (T,E,P)	0% (I)	0% (M)
Cliff+T;CX+T′	**100%**	0% (M)	0% (M)	0% (T)	0% (T,E,P)	0% (I)	0% (M)
Cliff+T;H+CZ+RX	**100%**	60% (M)	0% (M)	0% (T)	0% (T,P)	0% (I)	0% (M)
Cliff+T;H+CZ+RX′	**100%**	0% (T,M)	0% (M)	0% (T)	0% (T,P)	0% (I)	0% (M)
CCX+H;Cliff	**100%**	0% (T)	0% (M)	0% (M)	0% (T)	0% (T)	0% (M)
CCX+H;CX+T	**100%**	50% (T)	0% (M)	0% (T)	0% (T)	0% (T)	0% (M)
CCX+H;CX+T′	**100%**	0% (T,M)	0% (M)	0% (T)	0% (T)	0% (T)	0% (M)
RZ$_2$+H;CX	**100%**	0% (E)	0% (T,M)	0% (U)	0% (U)	0% (U)	0% (M)
RZ$_2$+H;CX′	**100%**	0% (E)	0% (M)	0% (U)	0% (U)	0% (U)	0% (M)
MeasureGHZ	**100%**	0% (U)	0% (U)	0% (U)	0% (U)	0% (U)	0% (U)
Overall success	**100%**	22%	0%	0%	0%	0%	0%

T: timeout (6h), M: out of memory, U: unsupported operation in the circuit,
I: incorrect simulation results, P: too imprecise, E: internal error

TABLE 5.6: Detailed comparison of outcomes from ABSTRAQT and QuiZX, including runtimes of successful runs.

Label	**Abstraqt**			**QuiZX**		
	Outcomes	min [s]	max [s]	Outcomes	min [s]	max [s]
Cliff;Cliff	$10 \times$ ✓	24	33	$0 \times$ ✓, $10 \times$ E	-	-
Cliff+T;Cliff	$10 \times$ ✓	32	47	$7 \times$ ✓, $3 \times$ T	$5.5 \cdot 10^3$	$2.0 \cdot 10^4$
Cliff+T;CX+T	$10 \times$ ✓	46	63	$8 \times$ ✓, $2 \times$ M	$2.0 \cdot 10^3$	$9.4 \cdot 10^3$
Cliff+T;CX+T′	$10 \times$ ✓	47	65	$0 \times$ ✓,$10 \times$ T	-	-
Cliff+T;H+CZ+RX	$10 \times$ ✓	58	69	$6 \times$ ✓, $4 \times$ M	$3.6 \cdot 10^3$	$1.4 \cdot 10^4$
Cliff+T;H+CZ+RX′	$10 \times$ ✓	52	71	$0 \times$ ✓, $1 \times$ T, $9 \times$ M	-	-
CCX+H;Cliff	$10 \times$ ✓	143	155	$0 \times$ ✓,$10 \times$ T	-	-
CCX+H;CX+T	$10 \times$ ✓	155	173	$5 \times$ ✓, $5 \times$ T	$5.9 \cdot 10^3$	$7.9 \cdot 10^3$
CCX+H;CX+T′	$10 \times$ ✓	155	173	$0 \times$ ✓, $1 \times$ T, $9 \times$ M	-	-
RZ$_2$+H;CX	$10 \times$ ✓	37	47	$0 \times$ ✓, $10 \times$ E	-	-
RZ$_2$+H;CX′	$10 \times$ ✓	37	46	$0 \times$ ✓, $10 \times$ E	-	-
MeasureGHZ	$10 \times$ ✓	23	32	$0 \times$ ✓, $10 \times$ U	-	-

T: timeout (6h), M: out of memory, U: unsupported operation in the circuit,
E: internal error

`RZ`$_2$`+H;CX` and `RZ`$_2$`+H;CX'`. Surprisingly, QuiZX also consistently fails to simulate `Cliff;Cliff`, which we conjecture is due to a bug for circuits that do not contain non-Clifford gates. After adding a single T gate, simulation is successful.

Importantly, even when QuiZX succeeds, it is significantly slower than ABSTRAQT, sometimes by more than two orders of magnitude.

ESS Surprisingly, ESS simulates circuits `Cliff;Cliff` and `Cliff+T;Cliff` incorrectly. Specifically, it samples the impossible measurement of 1 around 50% of cases. Interestingly, smaller circuits generated with the same process are handled correctly. It is reassuring to see that ABSTRAQT allows us to discover such instabilities in established tools.

It may be surprising that ESS returns an incorrect result for `Cliff+T;Cliff` instead of timing out, although the circuit contains many T gates—this is because Qiskit can establish that the Clifford+T part of the circuit is irrelevant when measuring the last qubit. For all remaining circuits, ESS runs out of memory or times out, as it decomposes the circuit into exponentially many Clifford circuits.

FEYNMAN Feynman consistently either times out, runs out of memory, or does not support a relevant operation (namely measurement and RZ_2).

YP21 YP21 typically either times out, throws an internal error, does not support a relevant operation (e.g., measurements or RZ_2), or returns incorrect results. The latter is because on some circuits, mode 2 choses an empty set of projectors, which leads to trivially unsound results. When YP21 does terminate, it is too imprecise to establish that the last qubit is in state $|0\rangle$.

STATEVECTOR Unsurprisingly, statevector simulation cannot handle the circuits in Table 5.5. This is because it requires space exponential in the number of qubits, which precludes simulating any of the benchmarks.

5.7.4 *Limitations and Discussion*

We note that our benchmarks are designed to showcase successful applications of ABSTRAQT where it outperforms existing tools. Of course, ABSTRAQT is not precise on all circuits—e.g., ABSTRAQT quickly loses precision on general Clifford+T circuits (analogously to the imprecise measurement discussed in §5.3).

FUTURE ABSTRACTIONS We expect that for many real-world circuits, existing approaches work better than the current implementation of ABSTRAQT. However, as ABSTRAQT only abstracts the first stabilizer simulation generalized to non-Clifford gates [36, §VII-C], we believe it paves the way to also abstract more recent stabilizer simulators. For example, ESS [37] operates on so-called *CH-forms* which, like the generalized stabilizer simulation underlying ABSTRAQT, can be encoded using bits and complex numbers. Hence, it seems plausible that our ideas could be adapted to abstract ESS. QuiZX operates on *ZX-diagrams* consisting of graphs whose nodes are parametrized by rotation angles α. Again, a promising direction for future research is introducing abstract ZX-diagrams that support abstract rotation angles. This is particularly promising because both ESS and QuiZX scale better in number of T gates than [36, §VII-C]: with 2^n instead of 4^n.

We note however that not all concrete simulation techniques are directly amenable to abstraction. For example, when naively abstracting the Clifford simulation by Aaronson and Gottesmann, applying a measurement requires selecting an entry in an boolean matrix that definitively equals one [36, Case I in §III]—it is unclear how to generalize this to abstract boolean matrices whose entries may be $\{0, 1\}$.

IMPROVING ABSTRAQT Another promising route towards better abstractions in incrementally improving ABSTRAQT itself. For example, it would be interesting to consider the effect of keeping more than one abstract summand, abstracting P_i or b_{ij} using a custom relational domain (which retains information about the relationship between different values) [111], or a more precise abstraction for complex numbers by taking into account that restricted gate sets such as Clifford+T only induce matrices over finite sets of values.

SUMMARY Overall, we believe that all tools in Table 5.5 are valuable to analyze quantum circuits. We are hoping that addressing some limitations of the considered baselines (e.g., fixing bugs in QuiZX and ESS) and cross-pollinating ideas (e.g., extending QuiZX by abstract interpretation) will allow the community to benefit from the fundamentally different mathematical foundations of all tools.

5.8 RELATED WORK

Here, we discuss works related to the goal and methods of ABSTRAQT.

QUANTUM ABSTRACT INTERPRETATION Some existing works have investigated abstract interpretation for simulating quantum circuits [40, 112, 113]. As [40] is not specialized for Clifford circuits, it is very imprecise on the circuits investigated in §5.7: it cannot derive that the lower qubits are $|0\rangle$ for any of them. While [112, 113] are inspired by stabilizer simulation, they only focus on determining if certain qubits are entangled or not, whereas ABSTRAQT can extract more precise information about the state. Further, both tools are inherently imprecise on non-Clifford gates—in contrast, a straight-forward extension of ABSTRAQT can treat some non-Clifford gates precisely at the exponential cost of not merging summands.

STABILIZER SIMULATION The Gottesman-Knill theorem [96] established that stabilizers can be used to efficiently simulate Clifford circuits. Stim [107] is a recent implementation of such a simulator, which only supports Clifford gates and Pauli measurements.

Stabilizer simulation was extended to allow for non-Clifford gates at an exponential cost, while still allowing efficient simulation of Clifford gates [36, §VII-C]. Various works build upon this insight, handling Clifford gates efficiently but suffering from an exponential blow-up on non-Clifford gates [37, 38, 97, 98, 99]. In our evaluation, we demonstrate that ABSTRAQT extends the reach of state-of-the-art stabilizer simulation by comparing to two tools from this category, ESS [37] (chosen because it is implemented in the popular Qiskit library) and QuiZX [38] (chosen because it is a recent tool reporting favorable runtimes).

VERIFYING QUANTUM PROGRAMS Another approach to establishing circuit properties is end-to-end formal program verification, as developed in [44] for instance. However, this approach often requires new insights for each program it is applied to. Even though recent works have greatly improved verification automation, proving even the simplest programs still requires a significant time investment [114], whereas our approach can analyze it without any human time investment.

The work [115] automatically generates rich invariants, but is exponential in the number of qubits, limiting its use to small circuits. Finally, [110] can automatically verify the equivalence of two given circuits, but times out on the benchmarks considered in §5.7.

5.9 CONCLUSION

In this chapter, we have demonstrated that combining abstract interpretation with stabilizer simulation allows to establish circuit properties that are intractable otherwise.

Our key idea was to over-approximate the behavior of non-Clifford gates in the generalized stabilizer simulation of Aaronson and Gottesman [36] by merging summands in the sum representation of the quantum states density matrix. Our carefully chosen abstract domain allows us to define efficient abstract transformers that approximate each of the concrete stabilizer simulation functions, including measurement.

6 CONCLUSION AND OUTLOOK

In conclusion, this thesis has introduced the three novel tools Silq, Unqomp, and Abstraqt with the aim of advancing the field of quantum computation.

These tools substantially lower the entrance barrier for non-expert quantum programmers and enable experts to realize the full potential of quantum computing. As a result, they can expedite the development of more efficient and correct quantum algorithms, as well as facilitate further progress in the field of quantum computing.

For future work, several directions can be explored to further enhance the capabilities of these tools. Developing a compiler for Silq is an ongoing effort [33], which could lead to significantly more efficient compilation results by exploiting the high-level information available in Silq. Similarly, the high-level nature of Silq programs could also simplify the formal verification of quantum algorithms, by relying on guarantees offered by the language [43, 44]. Additionally, we could improve the versatility of uncomputation synthesis by taking into account additional information such as resource constraints (as demonstrated in our follow-up work Reqomp [14]), or circuit identities (to generate more efficient circuits). Further, Abstraqt lays the groundwork to explore the effect of abstract interpretation on stabilizer simulation or other simulation techniques.

Overall, by continuing to leverage techniques from the programming language community, we believe the quantum computation community can further bridge the gap between classical and quantum programming, thereby unlocking the full potential of quantum computing.

BIBLIOGRAPHY

[1] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. “Silq: A High-level Quantum Language with Safe Uncomputation and Intuitive Semantics”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, 286. DOI: 10.1145/3385412.3386007. URL: https://doi.org/10.1145/3385412.3386007 (visited on 09/14/2020).

[2] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. “Unqomp: synthesizing uncomputation in Quantum circuits”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, 222. URL: https://doi.org/10.1145/3453483.3454040 (visited on 06/14/2022).

[3] Benjamin Bichsel, Maximilian Baader, Anouk Paradis, and Martin Vechev. *Abstraqt: Analysis of Quantum Circuits via Abstract Stabilizer Simulation*. arXiv:2304.00921 [quant-ph]. 2023. DOI: 10.48550/arXiv.2304.00921. URL: http://arxiv.org/abs/2304.00921 (visited on 04/17/2023).

[4] Benjamin Bichsel, Timon Gehr, and Martin Vechev. “Fine-Grained Semantics for Probabilistic Programs”. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, 145. DOI: 10.1007/978-3-319-89884-1_6.

[5] Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K. Mansinghka. “Incremental inference for probabilistic programs”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*. Philadelphia, PA, USA: ACM Press, 2018, 571. DOI: 10.1145/3192366.3192399. URL: http://dl.acm.org/citation.cfm?doid=3192366.3192399 (visited on 09/27/2018).

[6] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. “DP-Finder: Finding Differential Privacy Violations by Sampling and Optimization”. In: *Proceedings of the*

2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada: ACM, 2018, 508. DOI: 10.1145/3243734.3243863. URL: http://dl.acm.org/doi/10.1145/3243734.3243863 (visited on 04/07/2020).

[7] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. “Scalable Taint Specification Inference with Big Code”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. event-place: Phoenix, AZ, USA. New York, NY, USA: ACM, 2019, 760. DOI: 10.1145/3314221.3314648. URL: http://doi.acm.org/10.1145/3314221.3314648 (visited on 11/15/2019).

[8] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. “zkay: Specifying and Enforcing Data Privacy in Smart Contracts”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, 1759. DOI: 10.1145/3319535.3363222. URL: https://doi.org/10.1145/3319535.3363222 (visited on 12/02/2020).

[9] Pesho Ivanov, Benjamin Bichsel, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin Vechev. “AStarix: Fast and Optimal Sequence-to-Graph Alignment”. In: *Research in Computational Molecular Biology*. Ed. by Russell Schwartz. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, 104. DOI: 10.1007/978-3-030-45257-5_7.

[10] Benjamin Bichsel, Samuel Steffen, Ilija Bogunovic, and Martin Vechev. “DP-Sniper: Black-Box Discovery of Differential Privacy Violations using Classifiers”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. 2021, 391. DOI: 10.1109/SP40001.2021.00081.

[11] Pesho Ivanov, Benjamin Bichsel, and Martin Vechev. “Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds”. In: *Research in Computational Molecular Biology*. Ed. by Itsik Pe’er. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, 306. DOI: 10.1007/978-3-031-04749-7_22.

[12] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. “ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs”. In: *2022 IEEE Symposium on*

Security and Privacy (SP). ISSN: 2375-1207. 2022, 179. DOI: 10.1109/SP46214.2022.9833732.

[13] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. “Zapper: Smart Contracts with Data and Identity Privacy”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. New York, NY, USA: Association for Computing Machinery, 2022, 2735. DOI: 10.1145/3548606.3560622. URL: https://dl.acm.org/doi/10.1145/3548606.3560622 (visited on 04/17/2023).

[14] Anouk Paradis, Benjamin Bichsel, and Martin Vechev. *Reqomp: Space-constrained Uncomputation for Quantum Circuits*. arXiv:2212.10395 [quant-ph]. 2022. DOI: 10.48550/arXiv.2212.10395. URL: http://arxiv.org/abs/2212.10395 (visited on 04/25/2023).

[15] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. “Quantum Algorithm for Linear Systems of Equations”. In: *Phys. Rev. Lett.* 103 (15 2009), 150502. DOI: 10.1103/PhysRevLett.103.150502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502.

[16] Guang Hao Low, Theodore J. Yoder, and Isaac L. Chuang. “Quantum inference on Bayesian networks”. In: *Phys. Rev. A* 89.6 (2014), 062315. DOI: 10.1103/PhysRevA.89.062315. URL: https://link.aps.org/doi/10.1103/PhysRevA.89.062315.

[17] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. “Quantum principal component analysis”. In: *Nature Physics* 10.9 (2014), 631. DOI: 10.1038/nphys3029.

[18] Nathan Wiebe, Daniel Braun, and Seth Lloyd. “Quantum Algorithm for Data Fitting”. In: *Physical Review Letters* 109.5 (2012). DOI: 10.1103/physrevlett.109.050505.

[19] Patrick Rebentrost, M Mohseni, and Seth Lloyd. “Quantum Support Vector Machine for Big Data Classification”. In: *Physical Review Letters* 113 (2013).

[20] Vedran Dunjko, Jacob M. Taylor, and Hans J. Briegel. “Quantum-Enhanced Machine Learning”. In: *Physical Review Letters* 117.13 (2016). DOI: 10.1103/physrevlett.117.130501.

[21] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa

Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019). Number: 7779 Publisher: Nature Publishing Group, 505. DOI: 10.1038/s41586-019-1666-5. URL: https://www.nature.com/articles/s41586-019-1666-5 (visited on 03/17/2021).

[22] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL". In: *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. Vienna, Austria: ACM Press, 2018. DOI: 10.1145/3183895.3183901.

[23] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. "Quipper: a scalable quantum programming language". In: *PLDI'13*. Seattle, Washington, USA: ACM Press, 2013. DOI: 10.1145/2491956.2462177.

[24] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. "Qunity: A Unified Language for Quantum and Classical Computing". In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), 32:921. DOI: 10.1145/3571225. URL: https://dl.acm.org/doi/10.1145/3571225 (visited on 04/25/2023).

[25] Raphael Seidel, Sebastian Bock, Nikolay Tcholtchev, and Manfred Hauswirth. *Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers (PLanQC 2022) - ICFP 2022*. 2022. URL: https://icfp22.sigplan.org/details/planqc-2022/6/Qrisp-

A-Framework-for-Compilable-High-Level-Programming-of-Gate-Based-Quantum-Comput (visited on 04/25/2023).

[26] Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. *Q# as a Quantum Algorithmic Language*. arXiv:2206.03532 [quant-ph]. 2022. DOI: 10.48550/arXiv.2206.03532. URL: http://arxiv.org/abs/2206.03532 (visited on 04/25/2023).

[27] Christophe Chareton, Sébastien Bardin, Dongho Lee, Benoît Valiron, Renaud Vilmart, and Zhaowei Xu. *Formal Methods for Quantum Programs: A Survey*. arXiv:2109.06493 [cs]. 2022. DOI: 10.48550/arXiv.2109.06493. URL: http://arxiv.org/abs/2109.06493 (visited on 04/25/2023).

[28] Kartik Singhal. *Quantum Hoare Type Theory*. arXiv:2012.02154 [quant-ph]. 2021. DOI: 10.48550/arXiv.2012.02154. URL: http://arxiv.org/abs/2012.02154 (visited on 04/25/2023).

[29] Wang Fang, Mingsheng Ying, and Xiaodi Wu. *Differentiable Quantum Programming with Unbounded Loops*. arXiv:2211.04507 [quant-ph]. 2022. DOI: 10.48550/arXiv.2211.04507. URL: http://arxiv.org/abs/2211.04507 (visited on 04/25/2023).

[30] Evandro Chagas Ribeiro Da Rosa and Rafael De Santiago. “Ket Quantum Programming”. In: *ACM Journal on Emerging Technologies in Computing Systems* 18.1 (2021), 12:1. DOI: 10.1145/3474224. URL: https://dl.acm.org/doi/10.1145/3474224 (visited on 04/25/2023).

[31] Jens Palsberg. *Discussions on Silq*. Private conversation. 2023.

[32] Srinjoy Ganguly and Thomas Cambier. *Quantum Computing with Silq Programming: Get up and running with quantum computing with the simplicity of this new high-level programming language*. Packt Publishing, 2021.

[33] Hristo Venev. *Compiling and Running High-level Quantum Programs*. 2023. URL: https://popl23.sigplan.org/details/POPL-2023-student-research-competition/7/Compiling-and-Running-High-level-Quantum-Programs.

[34] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. *Verified Compilation of Quantum Oracles*. arXiv:2112.06700 [quant-ph]. 2022. DOI: 10.48550/arXiv.2112.06700. URL: http://arxiv.org/abs/2112.06700 (visited on 06/28/2023).

[35] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. "Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), 150:1. DOI: 10.1145/3428218. URL: https://doi.org/10.1145/3428218 (visited on 11/18/2020).

[36] Scott Aaronson and Daniel Gottesman. "Improved Simulation of Stabilizer Circuits". In: *Physical Review A* 70.5 (2004). arXiv: quant-ph/0406196, 052328. DOI: 10.1103/PhysRevA.70.052328. URL: http://arxiv.org/abs/quant-ph/0406196 (visited on 08/28/2020).

[37] Sergey Bravyi, Dan Browne, Padraic Calpin, Earl Campbell, David Gosset, and Mark Howard. "Simulation of quantum circuits by low-rank stabilizer decompositions". In: *Quantum* 3 (2019). arXiv:1808.00128 [quant-ph], 181. DOI: 10.22331/q-2019-09-02-181. URL: http://arxiv.org/abs/1808.00128 (visited on 03/08/2023).

[38] Aleks Kissinger and John van de Wetering. "Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions". In: *Quantum Science and Technology* 7.4 (2022). arXiv:2109.01076 [quant-ph], 044001. DOI: 10.1088/2058-9565/ac5d20. URL: http://arxiv.org/abs/2109.01076 (visited on 01/24/2023).

[39] Qiskit contributors. *Qiskit: An Open-source Framework for Quantum Computing*. 2023. DOI: 10.5281/zenodo.2573505.

[40] Nengkun Yu and Jens Palsberg. "Quantum abstract interpretation". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, 542. DOI: 10.1145/3453483.3454061. URL: https://doi.org/10.1145/3453483.3454061 (visited on 06/30/2021).

[41] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. "Quartz: superoptimization of Quantum circuits". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, 625. DOI: 10.1145/3519939.3523433. URL: https://dl.acm.org/doi/10.1145/3519939.3523433 (visited on 06/28/2023).

[42] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. *Quanto: Optimizing Quantum Circuits with Automatic Generation of Circuit Identities*. arXiv:2111.11387 [quant-ph]. 2021. DOI: `10.48550/arXiv.2111.11387`. URL: `http://arxiv.org/abs/2111.11387` (visited on 06/28/2023).

[43] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. "A verified optimizer for Quantum circuits". In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), 37:1. DOI: `10.1145/3434318`. URL: `https://doi.org/10.1145/3434318` (visited on 03/18/2021).

[44] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. "Proving Quantum Programs Correct". In: *Leibniz International Proceedings in Informatics (LIPIcs)* 193 (2021). Ed. by Liron Cohen and Cezary Kaliszyk, 21:1. DOI: `10.4230/LIPIcs.ITP.2021.21`. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/13916`.

[45] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. *A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm*. arXiv:2204.07112 [quant-ph]. 2022. DOI: `10.48550/arXiv.2204.07112`. URL: `http://arxiv.org/abs/2204.07112` (visited on 06/28/2023).

[46] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. *Formal Verification of Quantum Programs: Theory, Tools and Challenges*. arXiv:2110.01320 [quant-ph]. 2022. DOI: `10.48550/arXiv.2110.01320`. URL: `http://arxiv.org/abs/2110.01320` (visited on 04/25/2023).

[47] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. 10th anniversary ed. Cambridge ; New York: Cambridge University Press, 2010.

[48] Peter Selinger. "Towards a Quantum Programming Language". In: *Mathematical. Structures in Comp. Sci.* 14.4 (2004), 527. DOI: `10.1017/S0960129504004256`. URL: `https://doi.org/10.1017/S0960129504004256`.

[49] Emmanuel Knill. *Conventions for quantum pseudocode*. Tech. rep. Citeseer, 1996.

[50] Steven Roman. *Advanced Linear Algebra*. OCLC: 730328666. New York, NY: Springer Science+Business Media, LLC, 2008. URL: `http://site.ebrary.com/id/10230315` (visited on 06/14/2018).

[51] Microsoft. "Public submissions of the Microsoft Q# Coding Contest - Summer 2018". In: (2018). URL: https://codeforces.com/contest/1002/.

[52] Microsoft. "Public submissions of the Microsoft Q# Coding Contest - Winter 2019". In: (2019). URL: https://codeforces.com/contest/1116/.

[53] Neil J. Ross and Peter LeFanu Lumsdaine. *Algorithms.TF.Main*. https://www.mathstat.dal.ca/~selinger/quipper/doc/Algorithms-TF-Main.html. 2015.

[54] Jennifer Paykin, Robert Rand, and Steve Zdancewic. "QWIRE: a core language for quantum circuits". In: *POPL'17*. Paris, France: ACM Press, 2017. DOI: 10.1145/3009837.3009894.

[55] Dave Wecker and Krysta M Svore. "LIQUi|>: A software design architecture and domain-specific language for quantum computing". In: *arXiv preprint arXiv:1402.4467* (2014).

[56] Damian S. Steiger, Thomas Häner, and Matthias Troyer. "ProjectQ: an open source software framework for quantum computing". In: *Quantum* 2 (2018), 49. DOI: 10.22331/q-2018-01-31-49. URL: https://quantum-journal.org/papers/q-2018-01-31-49/ (visited on 07/06/2019).

[57] Google AI Quantum Team. "Cirq". In: (2017). URL: https://github.com/quantumlib/Cirq.

[58] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. "ReQWIRE: Reasoning about Reversible Quantum Circuits". In: *Electronic Proceedings in Theoretical Computer Science* 287 (2019). arXiv: 1901.10118, 299. DOI: 10.4204/EPTCS.287.17. URL: http://arxiv.org/abs/1901.10118 (visited on 07/05/2019).

[59] Matthew Amy, Martin Roetteler, and Krysta M. Svore. "Verified Compilation of Space-Efficient Reversible Circuits". In: *CAV'17*. Vol. 10427. Cham, 2017, 3. DOI: 10.1007/978-3-319-63390-9_1. URL: http://link.springer.com/10.1007/978-3-319-63390-9_1 (visited on 11/14/2018).

[60] Charles H Bennett. "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6 (1973), 525. DOI: 10.1147/rd.176.0525. URL: http://ieeexplore.ieee.org/document/5391327/ (visited on 07/06/2018).

[61] Lov K Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, 212.

[62] Timon Gehr, Sasa Misailovic, and Martin Vechev. “Psi: Exact symbolic inference for probabilistic programs”. In: *International Conference on Computer Aided Verification*. Springer, 2016, 62.

[63] Mariia Mykhailova and Martin Roetteler. “Microsoft Q# Coding Contest - Summer 2018 - Main Contest July 6-9, 2018”. In: (2018). URL: https://assets.codeforces.com/rounds/997-998/main-contest-editorial.pdf.

[64] Mariia Mykhailova and Martin Roetteler. “Microsoft Q# Coding Contest - Winter 2019 - Main Contest March 1-4, 2019”. In: (2019). URL: https://assets.codeforces.com/rounds/1116/contest-editorial.pdf.

[65] F. Magniez, M. Santha, and M. Szegedy. “Quantum Algorithms for the Triangle Problem”. In: *SIAM Journal on Computing* 37.2 (2007), 413. DOI: 10.1137/050643684. URL: https://epubs.siam.org/doi/abs/10.1137/050643684 (visited on 07/01/2019).

[66] Andrew M. Childs and Robin Kothari. “Quantum query complexity of minor-closed graph properties”. In: *STACS'11*. Vol. 9. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany, 2011, 661. DOI: 10.4230/LIPIcs.STACS.2011.661. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3052 (visited on 07/01/2019).

[67] Stephen Wiesner. “Conjugate Coding”. In: *SIGACT News* 15.1 (1983), 78. DOI: 10.1145/1008908.1008920. URL: http://doi.acm.org/10.1145/1008908.1008920 (visited on 10/31/2019).

[68] Daniel Nagaj, Or Sattath, Aharon Brodutch, and Dominique Unruh. “An Adaptive Attack on Wiesner's Quantum Money”. In: *Quantum Info. Comput.* 16.11-12 (2016), 1048. URL: http://dl.acm.org/citation.cfm?id=3179330.3179337 (visited on 10/31/2019).

[69] Peter Selinger and Benoit Valiron. “A lambda calculus for quantum computation with classical control”. In: *Mathematical Structures in Computer Science* 16.03 (2006), 527. DOI: 10.1017/S0960129506005238. URL: http://www.journals.cambridge.org/abstract_S0960129506005238 (visited on 11/02/2018).

[70] Peter Müller and Arnd Poetzsch-Heffter. “Universes: a type system for controlling representation exposure”. In: 1999.

[71] Steve Klabnik and Carol Nichols. *The Rust programming language*. San Francisco: No Starch Press, 2018.

[72] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. *Proto-Quipper with dynamic lifting*. arXiv:2204.13041 [quant-ph]. 2022. DOI: 10.48550/arXiv.2204.13041. URL: http://arxiv.org/abs/2204.13041 (visited on 04/25/2023).

[73] Abhinandan Pal and Anubhab Ghosh. "Qiwi: A Beginner Friendly Quantum Language". In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. New York, NY, USA: Association for Computing Machinery, 2022, 78. DOI: 10.1145/3563768.3563959. URL: https://dl.acm.org/doi/10.1145/3563768.3563959 (visited on 04/25/2023).

[74] Chris Heunen and Robin Kaarsgaard. "Quantum information effects". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), 2:1. DOI: 10.1145/3498663. URL: https://dl.acm.org/doi/10.1145/3498663 (visited on 04/25/2023).

[75] David M. Kahn and Jan Hoffmann. *Automatic Amortized Resource Analysis with the Quantum Physicist's Method*. arXiv:2106.13936 [cs]. 2021. DOI: 10.48550/arXiv.2106.13936. URL: http://arxiv.org/abs/2106.13936 (visited on 04/25/2023).

[76] Charles Yuan and Michael Carbin. "Tower: data structures in Quantum superposition". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), 134:259. DOI: 10.1145/3563297. URL: https://dl.acm.org/doi/10.1145/3563297 (visited on 04/25/2023).

[77] Charles Yuan, Christopher McNally, and Michael Carbin. "Twist: sound reasoning for purity and entanglement in Quantum programs". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), 30:1. DOI: 10.1145/3498691. URL: https://doi.org/10.1145/3498691 (visited on 01/31/2022).

[78] Julian Hans and Sven Groppe. "Silq2Qiskit - Developing a quantum language source-to-source translator". In: *Proceedings of the 5th International Conference on Computer Science and Software Engineering*. CSSE '22. New York, NY, USA: Association for Computing Machinery, 2022, 581. DOI: 10.1145/3569966.3570114. URL: https://doi.org/10.1145/3569966.3570114 (visited on 05/01/2023).

[79] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. “QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization”. In: *ACM Transactions on Quantum Computing* 3.3 (2022), 14:1. DOI: 10.1145/3491247. URL: https://dl.acm.org/doi/10.1145/3491247 (visited on 04/25/2023).

[80] Marco Lewis. *SilVer*. original-date: 2021-03-23T13:22:06Z. 2021. URL: https://github.com/marco-lewis/verif-silq (visited on 04/25/2023).

[81] Alex Parent, Martin Roetteler, and Krysta M. Svore. “Reversible Circuit Compilation with Space Constraints”. In: *arXiv:1510.00377 [quant-ph]* (2015). arXiv: 1510.00377. URL: http://arxiv.org/abs/1510.00377 (visited on 10/27/2020).

[82] John Preskill. “Quantum Computing in the NISQ era and beyond”. In: *Quantum* 2 (2018), 79.

[83] Umesh Vazirani. *Quantum Mechanics and Quantum Computation (CS191x)*. Online Lecture (Lecture 7). 2013. URL: https://www.youtube.com/watch?v=XPkKRBk71TY&list=PLDAjb_zu5aoFazE31_8yT0OfzsTcmvAVg&index=30.

[84] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. “Elementary gates for quantum computation”. In: *Physical Review A* 52.5 (1995). arXiv: quant-ph/9503016, 3457. DOI: 10.1103/PhysRevA.52.3457. URL: http://arxiv.org/abs/quant-ph/9503016 (visited on 11/09/2020).

[85] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5 (1999), 895. DOI: 10.1145/330249.330250.

[86] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[87] Arthur B Kahn. “Topological sorting of large networks”. In: *Communications of the ACM* 5.11 (1962), 558. DOI: 10.1145/368996.369025.

[88] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Boston: Addison Wesley, 2006.

[89] Qiskit Development Team. *Qiskit Circuit Library*. Accessed: 2020-10-27. 2020. URL: https://qiskit.org/documentation/apidoc/circuit_library.html.

[90] Cirq Development Team. *Cirq Circuit Examples*. Accessed: 2020-10-29. 2020. URL: https://github.com/quantumlib/Cirq.

[91] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. "SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020). arXiv: 2004.08539, 570. DOI: 10.1109/ISCA45697.2020.00054. URL: http://arxiv.org/abs/2004.08539 (visited on 11/20/2020).

[92] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), 319. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041 (visited on 10/23/2020).

[93] Atsushi Matsuo and Shigeru Yamashita. "Changing the Gate Order for Optimal LNN Conversion". In: *Reversible Computation*. Ed. by Alexis De Vos and Robert Wille. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, 89. DOI: 10.1007/978-3-642-29517-1_8.

[94] Ross Duncan and Simon Perdrix. "Rewriting Measurement-Based Quantum Computations with Generalised Flow". In: *Automata, Languages and Programming*. Ed. by Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, 285. DOI: 10.1007/978-3-642-14162-1_24.

[95] Lucas Dixon and Ross Duncan. "Graphical Reasoning in Compact Closed Categories for Quantum Computation". In: *arXiv:0902.0514 [cs]* (2009). arXiv: 0902.0514. URL: http://arxiv.org/abs/0902.0514 (visited on 10/23/2020).

[96] Daniel Gottesman. *The Heisenberg Representation of Quantum Computers*. Tech. rep. arXiv:quant-ph/9807006. arXiv:quant-ph/9807006 type: article. arXiv, 1998. DOI: 10.48550/arXiv.quant-ph/9807006. URL: http://arxiv.org/abs/quant-ph/9807006 (visited on 03/30/2023).

[97] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. "Extending Gottesman Types Beyond the Clifford Group". In: *The Second International Workshop on Programming Languages for Quantum Computing (PLanQC 2021)*. 2021. URL: https://pldi21.sigplan.

org/details/planqc-2021-papers/9/Extending-Gottesman-Types-Beyond-the-Clifford-Group.

[98] Hakop Pashayan, Oliver Reardon-Smith, Kamil Korzekwa, and Stephen D. Bartlett. "Fast estimation of outcome probabilities for quantum circuits". In: *PRX Quantum* 3.2 (2022). arXiv:2101.12223 [quant-ph], 020361. DOI: 10.1103/PRXQuantum.3.020361. URL: http://arxiv.org/abs/2101.12223 (visited on 03/09/2023).

[99] *Classical Simulation of Quantum Circuits with Partial and Graphical Stabiliser Decompositions*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/LIPICS.TQC.2022.5. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16512/.

[100] 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 ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. event-place: Los Angeles, California. New York, NY, USA: ACM, 1977, 238. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973 (visited on 08/30/2019).

[101] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks". In: *Journal of logic and computation* 2.4 (1992), 511. DOI: 10.1093/logcom/2.4.511.

[102] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software". In: *ACM SIGPLAN Notices* 38.5 (2003), 196. DOI: 10.1145/780822.781153. URL: https://doi.org/10.1145/780822.781153 (visited on 03/14/2023).

[103] Francesco Logozzo and Manuel Fähndrich. "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses". In: *Science of Computer Programming*. Special Issue on Object-Oriented Programming Languages and Systems (OOPS 2008), A Special Track at the 23rd ACM Symposium on Applied Computing 75.9 (2010), 796. DOI: 10.1016/j.scico.2009.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0167642309000719 (visited on 03/14/2023).

[104] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *2018 IEEE Symposium on Security and Privacy (SP)*.

San Francisco, CA: IEEE, 2018, 3. DOI: 10.1109/SP.2018.00058. URL: https://ieeexplore.ieee.org/document/8418593/ (visited on 12/18/2018).

[105] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (2020), 357. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[106] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015, 1. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162 (visited on 07/01/2021).

[107] Craig Gidney. "Stim: a fast stabilizer circuit simulator". In: *Quantum* 5 (2021). arXiv: 2103.02202, 497. DOI: 10.22331/q-2021-07-06-497. URL: http://arxiv.org/abs/2103.02202 (visited on 11/18/2021).

[108] Henry S. Warren. *Hacker's Delight*. 2nd. Addison-Wesley Professional, 2012. DOI: 10.5555/2462741.

[109] Aleks Kissinger and John van de Wetering. "PyZX: Large Scale Automated Diagrammatic Reasoning". In: *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Ed. by Bob Coecke and Matthew Leifer. Vol. 318. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2020, 229. DOI: 10.4204/EPTCS.318.14.

[110] Matthew Amy. "Towards Large-scale Functional Verification of Universal Quantum Circuits". In: *Electronic Proceedings in Theoretical Computer Science* 287 (2019). arXiv:1805.06908 [quant-ph], 1. DOI: 10.4204/EPTCS.287.1. URL: http://arxiv.org/abs/1805.06908 (visited on 09/01/2023).

[111] Antoine Miné. "Weakly Relational Numerical Abstract Domains". In: (2004).

[112] Simon Perdrix. “Quantum Entanglement Analysis Based on Abstract Interpretation”. In: *Proceedings of the 15th International Symposium on Static Analysis*. SAS ’08. event-place: Valencia, Spain. Berlin, Heidelberg: Springer-Verlag, 2008, 270. DOI: 10.1007/978-3-540-69166-2_18. URL: http://dx.doi.org/10.1007/978-3-540-69166-2_18 (visited on 09/20/2019).

[113] Kentaro Honda. “Analysis of Quantum Entanglement in Quantum Programs using Stabilizer Formalism”. In: *Electronic Proceedings in Theoretical Computer Science* 195 (2015). DOI: 10.4204/EPTCS.195.19.

[114] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. “An Automated Deductive Verification Framework for Circuit-building Quantum Programs”. In: *Programming Languages and Systems* 12648 (2021), 148. DOI: 10.1007/978-3-030-72019-3_6. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7984546/ (visited on 04/18/2023).

[115] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. “Invariants of Quantum Programs: Characterisations and Generation”. In: *SIGPLAN Not.* 52.1 (2017), 818. DOI: 10.1145/3093333.3009840. URL: https://doi.org/10.1145/3093333.3009840.

A

APPENDIX: SILQ

A.1 COMPARING SILQ TO QUIPPER AND QWIRE

Fig. A.1 and Fig. A.2 provide full versions of the programs shown in Fig. 3.2.

```
cTri <- foldM (\cTri j -> do
  let tau_j = tau ! j
  eed <- qinit (intMap_replicate rr False)
  -- computing eed = ee[tau[j]]
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  cTri <- foldM (\cTri k -> do
    let tau_k = tau ! k
    eedd_k <- qinit False
    -- eedd_k=eed[tau[k]]=ee[tau[j]][tau[k]]
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
    -- using eedd_k as ctrl
    cTri <- increment cTri ‘controlled‘ eedd_k .&&. (eew ! j) .&&. (eew ! k)
    -- uncomputing eedd_k
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
    qterm False eedd_k
    return cTri)
   cTri [j+1..rrbar-1]
  -- uncomputing eed
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  qterm (intMap_replicate rr False) eed
  return cTri)
 cTri [0..rrbar-1]
```

FIGURE A.1: Quipper code from Fig. 3.2.

A.2 GROVER'S ALGORITHM

Fig. A.3 shows an implementation of Grover's algorithm, including Grover's diffusion operator in Silq.

```
index  : ∏(n:Nat,i:Nat) . CIRC(t[n]          ,t[n]⊗         t) = ...
qindex : ∏(n:Nat,m:Nat) . CIRC(t[n]⊗qubit[m],t[n]⊗qubit[m]⊗t) = ...
controlledInc : ∏(n:Nat). CIRC(qubit[n]⊗qubit,qubit[n]⊗qubit) = ...

EvalCondition : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar],...⊗qubit
) = box(ee,tau,eew) =>
  (tau,tauj) <- unbox (index rrbar j) tau; -- tauj=tau[j]
  (tau,tauk) <- unbox (index rrbar k) tau; -- tauk=tau[k]
  (ee, tauj, eed) <- unbox (qindex rrbar r) ee tauj; -- eed=ee[tauj]
  (eed,tauk,eedd_k) <- unbox (qindex rrbar r) eed tauk; -- eedk=eed[tauk]
  (eew,eewj) <- unbox (index rrbar j) eew; -- eewj=eew[j]
  (eew,eewk) <- unbox (index rrbar k) eew; -- eewk=eew[k]
  (eedd_k,eewj,eewk,c) <- unbox and eedd_k eewj eewk; -- condition
  output (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) --output

LoopBody : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar],
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar]
) = box (ee,tau,eew,cTri) =>
  (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) <- unbox (EvalCondition r rrbar j k)
      ee tau eew; -- evaluate condition
  (cTri,c) <- unbox (controlledInc rrbar) cTri c; -- controlled increment
  (ee,tau,eew) <- unbox (reverseIsometric EvalCondition r rrbar j k) ee tau eew tauj
      tauk eed eedd_k eewj eewk c -- uncompute
  output (ee,tau,eew,cTri) -- output
```

FIGURE A.2: QWire code from Fig. 3.2.

```
def groverDiff[n:!ℕ](cand:uint[n]){
  for k in [0..n) { cand[k] := H(cand[k]); }
  if cand!=0 {
    phase(π);
  }
  for k in [0..n) { cand[k] := H(cand[k]); }
  return cand;
}

def grover[n:!ℕ](f:uint[n]! → lifted 𝔹){
  nIterations:= floor(π/4/asin(2^(-n/2)));
  cand:=0:uint[n];
  for k in [0..n) { cand[k] := H(cand[k]); }

  for k in [0..nIterations){
    if f(cand) { phase(π); }
    cand:=groverDiff(cand);
  }
  return measure(cand);
}
```

FIGURE A.3: Grover's diffusion operator in Silq.

A.3 UNCOMPUTING NON-QFREE EXPRESSIONS

Here, we show why uncomputing the condition in function `nonQfree` in Fig. 3.5 is not possible (in particular also not by following Bennet's construction). Fig. A.4a provides a rewritten version of `nonQfree` that makes its individual operations more explicit.

Without uncomputation, `nonQfree` produces `x` (implicitly duplicated before applying **`H`**), a modified `y`, and a temporary control `t`, hence uncomputation should remove `t` without uncomputing `x` or the modified `y`.

The most natural way to try to uncompute `t` is running Bennett's construction by (i) running `nonQfree`, (ii) duplicating the modified `y`, and (iii) reversing `nonQfree`. However, this would result in `x`, the original `y`, and the modified `y`, instead of just `x` and the modified `y`.

Fig. A.4 shows that more generally, dropping `t` from the state is unphysical. Specifically, dropping `t` from the state (which is the goal of correct uncomputation) can result in the invalid state 0.

```
1  def nonQfree(const x:𝔹,y:𝔹){
2    t := dup(x);
3    t := H(t);
4    if t{
5      y := X(y);
6    }
7    // uncompute t
8  }
```

(a) Rewritten version of `nonQfree` that makes its individual operations more explicit.

$$
\begin{aligned}
&\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y\\
\xrightarrow{\text{Line 2}}\;&\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t\\
\xrightarrow{\text{Line 3}}\;&\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y\left|-\right\rangle_t+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y\left|-\right\rangle_t\\
=&\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t\\
\xrightarrow{\text{Lines 4–6}}\;&\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t\\
\xrightarrow{\text{Line 7}}\;&\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y=0
\end{aligned}
$$

(b) Semantics of `nonQfree`.

FIGURE A.4: Semantics of `nonQfree` on input $\frac{1}{\sqrt{2}}\left|1\right\rangle_{\mathtt{x}}\left|0\right\rangle_{\mathtt{y}}+\frac{1}{\sqrt{2}}\left|1\right\rangle_{\mathtt{x}}\left|1\right\rangle_{\mathtt{y}}$ when uncomputing the condition.

$$\frac{\Gamma_i \vdash e_i \colon \tau_i \qquad \Gamma' \vdash e' \colon \times_{i=1}^{n} \tau_i \to \tau'}{\Gamma_1, \dots, \Gamma_n, \Gamma' \vdash e'(e_1, \dots, e_n) \colon \tau'}$$

$$\frac{\vec{x} \colon \vec{\tau}, \vec{y} \colon \vec{\tau}' \vdash e \colon \tau''}{\vec{y} \colon \vec{\tau}' \vdash \lambda(\vec{x} \colon \vec{\tau}).e \colon \times_{i=1}^{n} \tau_i \to \tau''}$$

$$\frac{\Gamma_c \vdash e \colon \mathbb{B} \qquad \Gamma \vdash e_1 \colon \tau \quad \Gamma \vdash e_2 \colon \tau}{\Gamma_c, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \colon \tau}$$

FIGURE A.5: The basic patterns of our typing rules (ignoring annotations) are standard for a linear type system.

A.4 TYPING RULES

In the following, we provide additional information on typing rules of Silq-Core.

A.4.1 *Basic Pattern of Typing Rules*

Fig. A.5 shows the basic patterns of our typing rules without annotations.

$$\begin{array}{crcl}
\texttt{H} & \mathbb{B} & !\xrightarrow{\texttt{mfree}} & \mathbb{B} \\
\texttt{phase} & !\texttt{float} & !\xrightarrow{\texttt{mfree}} & \mathbb{1} \\
\texttt{rotX}, \texttt{rotY}, \texttt{rotZ} & !\texttt{float} \times \mathbb{B} & !\xrightarrow{\texttt{mfree}} & \mathbb{B} \\
\texttt{X} & \mathbb{B} & !\xrightarrow{\texttt{qfree,mfree}} & \mathbb{B} \\
\texttt{Y}, \texttt{Z} & \mathbb{B} & !\xrightarrow{\texttt{mfree}} & \mathbb{B} \\
\texttt{dup} & \texttt{const}\ \tau & !\xrightarrow{\texttt{qfree,mfree}} & \tau \\
(\cdot, \dots, \cdot) & \times_{i=1}^{n} \tau_i & !\xrightarrow{\texttt{qfree,mfree}} & \times_{i=1}^{n} \tau_i \\
\texttt{forget}(\cdot = \cdot) & \tau \times \texttt{const}\ \tau & !\xrightarrow{\texttt{qfree,mfree}} & \mathbb{1} \\
\cdot \oplus \cdot & \texttt{const uint} \times \texttt{const uint} & !\xrightarrow{\texttt{qfree}} & \texttt{uint}
\end{array}$$

Figure A.6: Types of selected built-in functions.

A.4.2 *Types of Selected Built-in Functions*

Fig. A.6 shows the type of some built-in functions. In the following, we only discuss its most interesting aspects.

HADAMARD, PHASE The parameter of **H** is not **const**, meaning that evaluating **H** consumes its argument (the argument is not available after the call). In contrast, the parameters of $\oplus$ are **const**, meaning that adding two expressions preserves them. Further, **H** is only **mfree**, while $\oplus$ is **qfree** and **mfree**. The function **phase** requires a classical phase (!**float**), and does not return anything (indicated by $\mathbb{1}$).

DUP, TUPLING Function **dup** returns a copy of its argument, without changing the argument (indicated by **const** τ). For tupling $(\cdot, \dots, \cdot)$, our typing rule relies on the implicit tupling by function calls (see Fig. 3.10). It consumes its arguments and is **qfree**. As an implicit consequence, $(e_1, \dots, e_n)$ is classical if all e_i's are classical.

FORGET Function **forget**$(e_1 = e_2)$ leaves its second argument constant, but consumes its first. This allows us to *forget* e_1.

A.5 SEMANTICS

In the following, we provide additional information on semantics of Silq-Core.

$$\frac{}{\left[\varnothing \overset{\alpha}{\vdash} c\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi \otimes |c\rangle_{\underline{c}}}\ \text{const} \qquad \frac{}{\left[x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \mathbb{I}_{x \to \underline{x}}(\psi)}\ \text{var}$$

$$\frac{}{\left[\texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \texttt{dup}_{x \to x, \underline{x}}(\psi)}\ \text{var-const}$$

Figure A.7: Semantics of constants and variables.

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \text{drop}^{(x)}\,(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{!W} \quad \frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{W}$$

$$\frac{\left[\Gamma, x\colon !\tau, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \texttt{dup}_{x \to x, x}\,(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{!C}$$

$$\frac{\left[\Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \texttt{dup}_{x \to x, x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \text{drop}^{(x)}\,(\psi')}\ \text{C}$$

Figure A.8: Semantics of contraction and weakening.

$$\dfrac{\begin{array}{c|c}\text{evaluate arguments} & \text{determine function} \\ \left[\Gamma_i \overset{\alpha_i}{\vdash} e_i \colon \tau_i \,\middle|\, \psi_{i-1}\right] \xrightarrow{\text{run}} \psi_i' & \left[\Gamma' \overset{\alpha'}{\vdash} e' \colon \bigtimes_{i=1}^{n} \beta_i \tau_i ! \xrightarrow{\alpha'_{\text{func}}} \tau' \,\middle|\, \psi_n\right] \xrightarrow{\text{run}} \psi_{n+1} \otimes |e'', \overbrace{\sigma}^{\text{captured in } e''}\rangle_{\underline{e'}} \\ \psi_i = \psi_i' & \text{evaluate function} \quad \left[e''(\vec{e}) \colon \bigtimes_{i=1}^{n} \beta_i \tau_i ! \xrightarrow{\alpha'_{\text{func}}} \tau' \,\middle|\, \sigma \colon \Gamma' \,\middle|\, \psi_{n+1}\right] \xrightarrow{\text{eval}} \psi_{n+2} \end{array}}{\left[\vec{\Gamma}, \Gamma' \overset{\alpha''}{\vdash} e'(\vec{e}) \colon \tau' \,\middle|\, \psi_0\right] \xrightarrow{\text{run}} \underbrace{\mathbb{I}_{\underline{\text{ret}} \to \underline{e'(\vec{e})}}}_{\text{name result}} \circ \underbrace{\text{drop}^{(\{\underline{e_i} | \mathtt{const} \in \beta_i\})}}_{\text{uncompute}} (\psi_{n+2})} \ \text{func-eval}$$

$$\dfrac{\overset{\text{name arguments}}{\left[\vec{\beta}\vec{x} \colon \vec{\tau}, \Gamma \overset{\alpha}{\vdash} e'' \colon \tau' \,\middle|\, \mathbb{I}_{\underline{\vec{e}} \to \vec{x}}\Big(\psi \otimes |\sigma\rangle\Big)\right] \xrightarrow{\text{run}} \psi'}}{\left[\Big(\lambda(\vec{\beta}\vec{x} \colon \vec{\tau}).e''\Big)(\vec{e}) \colon \bigtimes_{i=1}^{n} \beta_i \tau_i \,! \xrightarrow{\alpha} \tau' \,\middle|\, \sigma \colon \Gamma \,\middle|\, \psi\right] \xrightarrow{\text{eval}} \underbrace{\mathbb{I}_{\underline{e''} \to \underline{\text{ret}}}}_{\text{name return value}} \circ \underbrace{\mathbb{I}_{\underline{x_i} \to \underline{e_i} | \mathtt{const} \in \beta_i}}_{\text{reset variable names}} (\psi')} \ \text{eval-}\lambda\text{-abs}$$

$$\dfrac{\psi = \sum_{v} \gamma_v \, |v\rangle_{\underline{e}} \otimes \tilde{\psi}_v \quad v' \in [\![\tau]\!]^{\mathsf{c}} \times [\![\tau]\!]^{\mathsf{q}}}{\left[\mathtt{measure}(e) \colon \tau \,! \xrightarrow{\alpha} ! \tau \,\middle|\, \varnothing \colon \varnothing \,\middle|\, \psi\right] \xrightarrow{\text{eval}} \gamma_{v'} \, |v'\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{v'}} \ \text{measure} \qquad \dfrac{}{\left[c(\vec{e}) \colon \times_{i=1}^{n} \beta_i \tau_i \,! \xrightarrow{\alpha} \tau' \,\middle|\, \varnothing \colon \varnothing \,\middle|\, \psi\right] \xrightarrow{\text{eval}} [\![c]\!]_{\underline{\vec{e}} \to (\underline{e_i})_{i | \mathtt{const} \in \beta_i}, \underline{\text{ret}}} (\psi)} \ \text{built-in-eval}$$

FIGURE A.9: Semantics of function calls.

$$\dfrac{\psi = |e_{\text{func}}, \sigma\rangle_{\underline{e}} \otimes \tilde{\psi}}{\left[\texttt{reverse}(e)\colon \left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{j=1}^{m} \tau'_j ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{k=1}^{l} \tau''_k\right) ! \xrightarrow{\texttt{mfree},\texttt{qfree}} \left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau''_k ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{j=1}^{m} \tau'_j\right) \middle| \varnothing \middle| \psi\right] \xrightarrow{\text{eval}} |\texttt{reverse}(e_{\text{func}}), \sigma\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}}\ \text{rev}$$

$$\dfrac{\left[e_{\text{func}}(\vec{e}^{\text{c}}, \vec{t})\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{j=1}^{m} \tau'_j ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{k=1}^{l} \tau''_k \middle| \sigma\colon \Gamma \middle| \psi'\right] \xrightarrow{\text{eval}} \mathbb{I}_{\underline{\vec{e}^{\cancel{\text{c}}}} \to \underline{\text{ret}}}(\psi) \quad \psi' \in \left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i, \vec{t}\colon \bigtimes_{j=1}^{m} \tau'_j, \Delta\right]\!\right]^{+} \quad \Delta \text{ according to } \psi}{\left[\texttt{reverse}(e_{\text{func}})(\vec{e}^{\text{c}}, \vec{e}^{\cancel{\text{c}}})\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau''_k ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{j=1}^{m} \tau'_j \middle| \sigma\colon \Gamma \middle| \psi\right] \xrightarrow{\text{eval}} \mathbb{I}_{\underline{\vec{t}} \to \underline{\text{ret}}}(\psi')}\ \text{call-rev}$$

FIGURE A.10: Semantics of reverse.

$$\dfrac{\left[\Gamma_c \overset{\alpha_c}{\vdash} e_c\colon \mathbb{B} \middle| \psi\right] \xrightarrow{\text{run}} \psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}} \quad \left[\Gamma \overset{\alpha_t}{\vdash} e_t\colon \tau \middle| \psi_t\right] \xrightarrow{\text{run}} \psi'_t \quad \left[\Gamma \overset{\alpha_f}{\vdash} e_f\colon \tau \middle| \psi_f\right] \xrightarrow{\text{run}} \psi'_f}{\left[\Gamma_c, \Gamma \overset{\alpha}{\vdash} \underbrace{\texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_f}_{e}\colon \tau \middle| \psi\right] \xrightarrow{\text{run}} \mathbb{I}_{\underline{e_t} \to \underline{e}}(\psi'_t) + \mathbb{I}_{\underline{e_f} \to \underline{e}}\left(\psi'_f\right)}\ \text{ite-q}$$

FIGURE A.11: Semantics of control flow. The rule is analogous for $e_c\colon !\mathbb{B}$.

$$\begin{array}{rllr}
 & \Big(\llbracket \mathbf{X} \rrbracket_{x \to y}\Big)\Big(|b\rangle_{x} \otimes \quad |\vec{w}\rangle_{\vec{z}}\Big) & & (A.1) \\
= & \llbracket \mathbf{X} \rrbracket_{x \to y}\Big(|b\rangle_{x}\Big) \otimes \mathbb{I}\Big(|\vec{w}\rangle_{\vec{z}}\Big) & & (A.2) \\
= & \llbracket \mathbf{X} \rrbracket_{x \to y}\Big(|b\rangle_{x}\Big) \otimes \quad |\vec{w}\rangle_{\vec{z}} & \mathbb{I} & (A.3) \\
= & \Big(\llbracket \mathbf{X} \rrbracket\,\big(|b\rangle\big)\Big)_{y} \quad \otimes \quad |\vec{w}\rangle_{\vec{z}} & x \to y & (A.4) \\
= & |1-b\rangle_{y} \quad \otimes \quad |\vec{w}\rangle_{\vec{z}} & \mathbf{X} & (A.5)
\end{array}$$

FIGURE A.12: Operating on named states with context.

A.5.1 *Semantics of Expressions*

In the following, we provide formal semantics for Silq-Core expressions.

CONSTANTS, VARIABLES Fig. A.7 first shows the rule for constants, which adds the constant to the state. Then, it shows the rules for variables. For consumed variables, $\mathbb{I}_{x \to \underline{x}}$ renames x to $\underline{x}$ in ψ without affecting other variables in ψ (shortly discussed in more detail). In contrast, the rule for constant variables preserves x and introduces an explicit duplicate $\underline{x}$ by $\llbracket \mathbf{dup} \rrbracket$, which maps $|v\rangle$ to $|v, v\rangle$ (cp. Fig. A.14).

OPERATING ON NAMED STATES WITH CONTEXT We provide a more detailed example demonstrating the effect of subscript "$x \to \underline{x}$" in Fig. A.12, which shows how to apply $\llbracket \mathbf{X} \rrbracket_{x \to y}$ to state $|b\rangle_{x} \otimes |\vec{w}\rangle_{\vec{z}}$, where the formal definition of $\llbracket \mathbf{X} \rrbracket$ is $\llbracket \mathbf{X} \rrbracket\,(|b\rangle) = |1-b\rangle$ (see App. A.5.2).

Here, the subscript $x \to y$ of $\mathbf{X}$ ensures that we (i) preserve $|\vec{w}\rangle_{\vec{z}}$ (cp. Eqs. (A.2)–(A.3)) and (ii) run $\mathbf{X}$ on $|b\rangle_{x}$ and name the output y (cp. Eq. (A.4)).

Here, it is crucial that we assume the standard representation introduced in Fig. 3.16, which ensures that classical and quantum components of variable x are stored together as $|(v, v')\rangle_{x}$. As a consequence, we know that if $\vec{z}$ contains one or more occurrences of x, these represent duplicates of x, as opposed to classical or quantum components of x.

CONTRACTION, WEAKENING Next, Fig. A.8 shows the semantics of contraction and weakening.

If the weakening rule drops a classical variable x from the context (rule !W), the semantics drops x from the state, using $\text{drop}^{(x)}\left(|v\rangle_x |\vec{w}\rangle_{\vec{y}}\right) = |\vec{w}\rangle_{\vec{y}}$. If the context contains multiple occurrences of x, only the first occurrence of x is dropped.

If the rule drops a constant variable (rule W), the semantics ignores this. Instead, it waits until the end of the function to uncompute all constant variables.

The contraction rule for classical variables (rule !C) duplicates the contracted variable x. In contrast, the contraction rule for quantum variables (rule C), duplicating the contracted variable x, and removes the duplicate after evaluating e. This removal of duplicates is not needed for classical variables, as only constant variables are preserved after their last usage.

FUNCTION CALLS The first rule in Fig. A.9 shows the semantics of a generic function call $e'(\vec{e})$. First, the rule evaluates all arguments, resulting in state ψ_n. Second, the rule evaluates e', resulting in state ψ_{n+1} containing the function e'' to be evaluated, which may capture variables σ. We note that the rule implicitly assumes that the function to be evaluated is classically known — a property guaranteed by our type system. Third, it evaluates the function using a transition rule of the form $[e''(\vec{e}) \mid \sigma \mid \psi_{n+1}] \xrightarrow{\text{eval}} \psi_{n+2}$. In contrast to run-transitions, eval-transitions assume that all arguments $\vec{e}$ are already evaluated in ψ_{n+1} (as guaranteed by run-transitions). Finally, the rule drops the **`const`** arguments of e'' by uncomputing them, and renames the output value from $\underline{\text{ret}}$ to $\underline{e'(\vec{e})}$.

EVAL-TRANSITIONS All remaining rules in Fig. A.9 are eval-transitions. The rules modify their input state ψ according to the called function, and then store the result in $\underline{\text{ret}}$.

MEASUREMENT The rule for measurement selects one possible measurement v' and collapses the state to this value. Note that measurement allows multiple transitions, one for each possible measured value v'. Here, and in various other eval-transitions, the state of captured variables is $\sigma = \varnothing$, as measurement cannot capture variables.

BUILT-IN FUNCTIONS The rule for evaluating built-in functions c relies on the semantics $[\![c]\!]$ of these functions, as discussed in App. A.5.2. The subscript to $[\![c]\!]$ ensures that the function operates on input values named

$\underline{e_1}, \ldots, \underline{e_n}$, and names the output values $\underline{e_i}$ (if the i^{th} argument of c is **const**) and $\underline{\text{ret}}$ (to indicate the return value).

EVALUATING LAMBDA ABSTRACTION The last rule in Fig. A.9 evaluates a lambda abstraction. First, it adds the variables captured in e'' to the current state ψ. Second, it renames the values of the evaluated arguments to the names of the parameters of e''. Third, it runs e'' on the resulting state, obtaining ψ'. Finally, it resets the variable names of constant parameters to e'' back to $\underline{e_i}$ and names the return value $\underline{\text{ret}}$.

REVERSE Fig. A.10 shows the semantics of **reverse**. Expression **reverse**(e) does not immediately reverse e (which evaluates to function e_{func}), but instead records that e_{func} should be reversed, by storing **reverse**(e_{func}) and the state σ captured by e_{func} under $\underline{\text{ret}}$.

The actual reversal is performed upon a call to the reversed function, also shown in Fig. A.10. Here, we explicitly split the arguments into $\vec{e}^{c}$ (the **const** arguments) and $\vec{e}^{\cancel{c}}$ (the non-**const** arguments), as assumed by Fig. 3.12. Intuitively, rule call-reversed maps ψ to ψ', if running e_{func} on ψ' yields ψ. However, it must also account for naming mismatches: Running e_{func} on ψ' yields $\underline{\text{ret}}$ instead of $\underline{\vec{e}^{\cancel{c}}}$, and the name of the returned value must be $\underline{\text{ret}}$.

We note that it is possible that there is no ψ' satisfying the premise of call-reversed, when e_{func} is not surjective. In this case, **reverse**(e_{func}) is undefined on ψ, which intuitively happends if ψ is not in the range of e_{func}).

CONTROL FLOW Fig. A.11 shows the semantics of control flow, handling both classical and quantum control flow. The rule (i) evaluates condition e and (ii) splits the resulting state into two states based on the value of e. Then, it evaluates e_1 in the first state and e_2 in the second. Finally, it adds both resulting states and drops e from the state.

A.5.2 *Semantics of Built-in Functions*

Fig. A.13a shows the semantic space of built-in functions f in terms of partial linear functions $[\![f]\!]$, where being a partial function allows us to support undefined behavior for some inputs.

Note that the function space of $[\![f]\!]$ in principle admits functions (i) violating **const** by modifying constant arguments and even (ii) violating the rules of quantum physics as in $\alpha\,|0\rangle + \beta\,|1\rangle \mapsto (\alpha + \beta)\,|0\rangle$. Thus, we must

For $f\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{i=1}^{m} \tau'_i \xrightarrow{\alpha} \tau''$, we have

$$[\![f]\!] : \left[\!\!\left[\bigtimes_{i=1}^{n} \tau_i \times \bigtimes_{i=1}^{m} \tau'_i \right]\!\!\right]^{+} \overset{\text{lin.}}{\not\to} \left[\!\!\left[\bigtimes_{i=1}^{n} \tau_i \times \tau'' \right]\!\!\right]^{+}$$

(a) Semantics of a general built-in function f.

$$[\![\mathtt{X}]\!] \, |b\rangle = |1-b\rangle \tag{A.6}$$

$$[\![\mathtt{X}]\!] \left(\sum_{b=0}^{1} \gamma_b \, |b\rangle \right) = \sum_{b=0}^{1} \gamma_b \, [\![\mathtt{X}]\!] \, |b\rangle = \sum_{b=0}^{1} \gamma_b \, |1-b\rangle \tag{A.7}$$

(b) Semantics of $\mathtt{X}$.

FIGURE A.13: Semantic of built-in functions.

ensure that these violations do not occur for the built-in functions defined by Silq-Core.

As an example, Fig. A.13b shows the semantics of $\mathtt{X}$ on basis states (Eq. (A.6)), the quantum semantics are given by linear extension (Eq. (A.7)). For simplicity, the semantics in Fig. A.13a (i) operates on states with unnamed indices and (ii) does not take context into account. However, our operational semantics operates on states with named indices involving context. Fig. A.12 shows how to bridge this gap when applying $\mathtt{X}$ to state $|b\rangle_{x} \otimes |\vec{w}\rangle_{\vec{z}}$. The subscript $x \to y$ of $\mathtt{X}$ ensures (i) we preserve $|\vec{w}\rangle_{\vec{z}}$ (cp. Eqs. (A.2)–(A.3)) and (ii) we run $\mathtt{X}$ on $|b\rangle_{x}$ and name the output y (cp. Eq. (A.4)).

SEMANTICS OF SELECTED BUILT-IN FUNCTIONS Fig. A.13 shows the semantics of selected built-in functions in Silq-Core.

The semantics of $\texttt{forget}(\cdot = \cdot)$ is only defined if its two arguments evaluate to the same value.

A.5.3 *Semantics Example*

We provide an example semantic derivation tree in Fig. A.15. It demonstrates weakening, contraction, and function evaluation.

$$
\begin{aligned}
[\![\texttt{H}]\!]\,|b\rangle &= \frac{1}{\sqrt{2}}\Big(\,|0\rangle + (-1)^b\,|1\rangle\,\Big)\\
[\![\texttt{phase}]\!]\,|r\rangle &= e^{\mathrm{i}r}\cdot|()\rangle\\
[\![\texttt{rotX}]\!]\,|r\rangle\,|b\rangle &= \Big(\cos\frac{r}{2}\,|b\rangle - \mathrm{i}\sin\frac{r}{2}\,[\![\texttt{X}]\!]\,|b\rangle\Big)\\
[\![\texttt{X}]\!]\,|b\rangle &= |1-b\rangle\\
[\![\texttt{rotY}]\!]\,|r\rangle\,|b\rangle &= \Big(\cos\frac{r}{2}\,|b\rangle - \mathrm{i}\sin\frac{r}{2}\,[\![\texttt{Y}]\!]\,|b\rangle\Big)\\
[\![\texttt{Y}]\!]\,|b\rangle &= \mathrm{i}\cdot(-1)^b\,|1-b\rangle\\
[\![\texttt{rotZ}]\!]\,|r\rangle\,|b\rangle &= \Big(\cos\frac{r}{2}\,|b\rangle - \mathrm{i}\sin\frac{r}{2}\,[\![\texttt{Z}]\!]\,|b\rangle\Big)\\
[\![\texttt{Z}]\!]\,|b\rangle &= (-1)^b\,|b\rangle\\
[\![\texttt{dup}]\!]\,|v\rangle &= |v,v\rangle\\
[\![(\cdot,\ldots,\cdot)]\!]\,|v_1\rangle\cdots|v_n\rangle &= |v_1,\ldots,v_n\rangle\\
[\![\texttt{forget}(\cdot=\cdot)]\!]\,|v\rangle\,|w\rangle &= \begin{cases}|v\rangle & v=w\\ \text{undefined} & v\neq w\end{cases}\\
[\![\cdot\oplus\cdot]\!]\,|v_1,v_2\rangle &= |v_1,v_2,v_1\oplus v_2\rangle
\end{aligned}
$$

FIGURE A.14: Example semantics of built-in functions. Most definitions are taken from [47, §4.2]. All definitions can be linearly extended.

A.6 PROOFS

Here, we provide proofs for key results.

A.6.1 *Theorems*

We recall all theorems presented in §3.6 in the following.

Theorem 3.6.1 (Type Preservation). *If we have*

$$
\begin{aligned}
&\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}',\\
&\Big[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\Big|\; \psi\Big] \xrightarrow{run} \psi',\ \textit{and}\\
&\psi \in \iota\left([\![\Gamma,\Delta]\!]\right),
\end{aligned}
$$

then ψ' lies in $\iota\left([\![\textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta]\!]\right)$.

$$\triangledown_{1} = \dfrac{}{\left[\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x\colon \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 0\rangle_x \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y}\ \text{var}$$

$$\triangledown_{2} = \dfrac{}{\left[\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x\colon \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y}\ \text{var}$$

$$\triangledown_{3} = \dfrac{}{\left[\varnothing \overset{\alpha}{\vdash} \cdot \,||\, \cdot \colon \texttt{const}\ \mathbb{B} \times \texttt{const}\ \mathbb{B}\ !\!\xrightarrow{\alpha} \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y \otimes \lvert \cdot \,||\, \cdot, \varnothing\rangle_{\underline{\cdot ||\cdot}}}\ \text{const}$$

(a) Subtrees of full semantic derivation tree (provided separately due to space constraints).

$$\dfrac{\dfrac{\dfrac{\triangledown_{1} \quad \triangledown_{2} \quad \triangledown_{3} \quad \dfrac{\text{Assuming: } [\![\cdot \,||\, \cdot]\!] \lvert a\rangle \lvert b\rangle = \lvert a\rangle \lvert b\rangle \lvert a \,||\, b\rangle}{\left[(\cdot \,||\, \cdot)(x, x)\colon \texttt{const}\ \mathbb{B} \times \texttt{const}\ \mathbb{B}\ !\!\xrightarrow{\alpha} \mathbb{B} \;\middle|\; \varnothing\colon \varnothing \;\middle|\; \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y\right] \xrightarrow{\text{eval}} \lvert 0\rangle_x \lvert 0\rangle_x \lvert 0\rangle_{\underline{x}} \lvert 0\rangle_{\underline{x}} \lvert 1\rangle_y \lvert 0\rangle_{\underline{\text{ret}}}}\ \text{built-in-eval}}{\left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 0\rangle_x \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 0\rangle_x \lvert 1\rangle_y \lvert 0\rangle_{\underline{x||x}}}\ \text{func-eval}}{\left[\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 1\rangle_y \lvert 0\rangle_{\underline{x||x}}}\ \text{C}}{\left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B} \;\middle|\; \lvert 0\rangle_x \lvert 1\rangle_y\right] \xrightarrow{\text{run}} \lvert 0\rangle_x \lvert 1\rangle_y \lvert 0\rangle_{\underline{x||x}}}\ \text{W}$$

(b) Full semantic derivation tree for **const** $x\colon \mathbb{B}$, **const** $y\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x\colon \mathbb{B}}\ \text{var} \qquad \dfrac{}{\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x\colon \mathbb{B}}\ \text{var} \qquad \dfrac{\text{(assuming } || \text{ is built-in)}}{\varnothing \overset{\alpha}{\vdash} \cdot \,||\, \cdot\colon \texttt{const}\ \mathbb{B} \times \texttt{const}\ \mathbb{B}\ !\!\xrightarrow{\alpha} \mathbb{B}}}{\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}}\ \text{func-eval}}{\texttt{const}\ x\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}}\ \text{C}}{\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}}\ \text{W}$$

(c) Type derivation tree for **const** $x\colon \mathbb{B}$, **const** $y\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}$.

FIGURE A.15: Semantics of **const** $x\colon \mathbb{B}$, **const** $y\colon \mathbb{B} \overset{\alpha}{\vdash} x \,||\, x\colon \mathbb{B}$ on input state $\lvert 0\rangle_x \lvert 1\rangle_y$. Here, α = **qfree**, **mfree** and gray parts of states correspond to the additional context Δ.

Theorem 3.6.2 (Const Semantics). *If we have*

$$\begin{aligned}&\Gamma = \textbf{\texttt{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}',\\ &\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{run} \psi', \text{ and}\\ &\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},\end{aligned}$$

then $\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$ *for some* $\chi_{\vec{v},\vec{w}}$.

Theorem 3.6.3 (Mfree Semantics). *If* $\textbf{\texttt{mfree}} \in \alpha$, $\sigma \in [\![\Gamma, \Delta]\!]^c$,

$$\begin{aligned}&\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_1)\right] \xrightarrow{run} \psi_1' \quad \text{for} \quad \psi_1 \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right), \text{and}\\ &\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_2)\right] \xrightarrow{run} \psi_2' \quad \text{for} \quad \psi_2 \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right),\end{aligned}$$

then $\langle\psi_1|\,|\psi_2\rangle = \langle\psi_1'|\,|\psi_2'\rangle$.

Theorem 3.6.4 (Qfree Semantics). *If* $\Gamma \overset{\alpha}{\vdash} e\colon \tau''$ *for* $\textbf{\texttt{qfree}} \in \alpha$ *and context* $\Gamma = \textbf{\texttt{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$*, then there exists a function* $\bar{f}\colon [\![\Gamma]\!]^s \to [\![\textbf{\texttt{const}}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'']\!]^s$ *on ground sets such that*

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \left|\sigma\right\rangle \otimes \tilde{\psi}_\sigma\right] \xrightarrow{run} \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \left|\bar{f}(\sigma)\right\rangle \otimes \tilde{\psi}_\sigma,$$

where $[\![\Gamma]\!]^s$ *is a shorthand for the ground set* $[\![\Gamma]\!]^c \times [\![\Gamma]\!]^q$ *on which the Hilbert space* $[\![\Gamma]\!]^+ = \mathcal{H}\left([\![\Gamma]\!]^s\right)$ *is defined.*

We will prove a different formulation of this theorem to improve presentation. Because of Thm. 3.6.2, we know that the constant part of Γ is preserved, hence it suffices to prove that there exists a function $\bar{f}\colon [\![\vec{\tau}, \vec{\tau}']\!]^s \to [\![\tau'']\!]^s$ such that

$$\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$$

gets mapped to

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\bar{f}(\vec{v}, \vec{w})\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

Theorem 3.6.5 (Physicality). *The semantics of well-typed Silq programs is physically realizable on a QRAM.*

We use the following helper lemma to prove Thm. 3.6.5.

Lemma 1. *Any well-typed* **`mfree`** *expression e can be implemented on a QRAM which maps $\psi \in \iota\left(\llbracket \Gamma, \Delta \rrbracket\right)$ to ψ' if $\left[\Gamma \overset{\alpha,\texttt{mfree}}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{run} \psi'$.*

Proof. Let $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$ and $\sigma \in \llbracket \Gamma, \Delta \rrbracket^{\mathsf{c}}$. From Thm. 3.6.3, we know that there exists a linear isometry M_σ

$$M_\sigma\colon \mathcal{A} \to \iota\left(\left\llbracket \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau, \Delta \right\rrbracket\right),$$

where $\mathcal{A} := \{\iota(\sigma, \tilde{\psi}) \mid \tilde{\psi} \in \mathcal{H}\left(\llbracket \Gamma, \Delta \rrbracket^{\mathsf{q}}\right)\}$. Hence, given ψ, a QRAM can (i) extract the classical components of ψ, (ii) determine M_σ based on those classical components σ, and (iii) run M_σ on ψ, yielding ψ'. $\square$

A.6.2 *Proofs for Run*

To improve presentation, we prove all theorems simultaneously in one large inductive proof. In the following, we discuss each semantic rule, e.g., the rules in Fig. A.7. For each rule, we will mark the part for type-preservation (Thm. 3.6.1) by [T], the part for preserving constants (Thm. 3.6.2) by [C], the part for **`mfree`** expressions (Thm. 3.6.3) by [M], the part for **`qfree`** expressions (Thm. 3.6.4) by [Q], and the part for physicality (Thm. 3.6.5) by [P].

A.6.2.1 *[const]*

The rule

$$\frac{}{\left[\varnothing \overset{\alpha}{\vdash} c\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi \otimes |c\rangle_{\underline{c}}}$$

maps ψ to $\psi \otimes |c\rangle_{\underline{c}}$.

[T] Since $\Gamma = \varnothing$ we have that $\psi \in \iota\left(\llbracket \Delta \rrbracket\right)$. Hence we have immediately $\psi' = \psi \otimes |c\rangle_{\underline{c}} \in \iota\left(\left\llbracket \underline{c} : \tau, \Delta \right\rrbracket\right)$.

[C] Since $\Gamma = \varnothing$ we have that $\psi = \tilde{\psi}$. Hence we have immediately $\psi' = \tilde{\psi} \otimes \chi = \psi \otimes |c\rangle_{\underline{c}}$, where $\chi = |c\rangle_{\underline{c}}$.

[M] We have

$$(\psi_1^\dagger \otimes \langle c|_{\underline{c}})(\psi_2 \otimes |c\rangle_{\underline{c}}) = \psi_1^\dagger \psi_2,$$

where $\psi_1^\dagger \psi_2$ denotes the inner product $\langle \psi_1| \, |\psi_2\rangle$.

[Q] Function $\bar{f}(\cdot) = c$ has the correct behavior.

[P] A QRAM can prepare prepare state c in variable $\underline{c}$.

A.6.2.2 *[var]*

The rule

$$\frac{}{\left[x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \mathbb{I}_{x \to \underline{x}}(\psi)} \text{var}$$

maps $\psi = \sum_w \gamma_w \left|w\right\rangle_x \otimes \tilde{\psi}_w$ to $\sum_w \gamma_w \left|w\right\rangle_{\underline{x}} \otimes \tilde{\psi}_w$.

[T] Since $\Gamma = x\colon \tau$ and $\psi \in \iota\left(\llbracket x : \tau, \Delta \rrbracket\right)$, we have that $\psi' \in \iota\left(\left\llbracket \underline{x} : \tau, \Delta \right\rrbracket\right)$.

[C] We have that $\psi' = \sum_w \gamma_w \left|w\right\rangle_{\underline{x}} \otimes \tilde{\psi}_w$, hence the claim.

[M] This is straightforward as renaming does not change the inner product.

[Q] Function $\bar{f}(v) = v$ has the correct behavior.

[P] A QRAM can simply rename variable x to $\underline{x}$.

A.6.2.3 *[var-const]*

The rule is

$$\frac{}{\left[\texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \;\middle|\; \psi\right] \xrightarrow{\text{run}} \texttt{dup}_{x \to x, \underline{x}}(\psi)} \text{var-const}$$

mapping $\psi = \sum_v \gamma_v \left|v\right\rangle_x \otimes \tilde{\psi}_v$ to $\psi' = \sum_v \gamma_v \left|v\right\rangle_x \left|v\right\rangle_{\underline{x}} \otimes \tilde{\psi}_v$

[T] Since $\psi \in \iota\left(\llbracket \texttt{const}\ x\colon , \Delta \rrbracket\right)$, we have that the state $\psi' \in \iota\left(\left\llbracket \texttt{const}\ x\colon \tau, \underline{x}\colon \tau, \Delta \right\rrbracket\right)$.

[C] The claim follows immediately.

[M] We have

$$\begin{aligned}
\psi_1'^{\dagger} \psi_2' &= \sum_v \gamma_v^{1*} \left\langle v\right|_x \left\langle v\right|_{\underline{x}} \otimes \tilde{\psi}_v^{1\dagger} \sum_w \gamma_w^2 \left|w\right\rangle_x \left|w\right\rangle_{\underline{x}} \otimes \tilde{\psi}_w^2 \\
&= \sum_v \gamma_v^{1*} \gamma_v^2 \tilde{\psi}_v^{1\dagger} \tilde{\psi}_v^2 \\
&= \psi_1^{\dagger} \psi_2.
\end{aligned}$$

[Q] Function $\bar{f}(v) = v$ has the correct behavior.

[P] A QRAM can run the linear isometry **dup**.

A.6.2.4 *[!W]*

The rule is

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \operatorname{drop}^{(x)}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'} \text{ !W.}$$

The general state for $\psi \in \iota\left(\llbracket \Gamma, x\colon !\tau, \Delta \rrbracket\right)$ is

$$\psi = |v\rangle_x \otimes \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \, |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

where $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$. Note that

$$\operatorname{drop}^{(x)}(\psi) = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \, |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

[T] As $\operatorname{drop}^{(x)}(\psi) \in \iota\left(\llbracket \Gamma, \Delta \rrbracket\right)$, the claim follows from the induction hypothesis.

[C] The claim follows from the induction hypothesis.

[M] By the induction hypothesis, we know that

$$\operatorname{drop}^{(x)}(\psi_1)^\dagger \operatorname{drop}^{(x)}(\psi_2) = \psi_1'^\dagger \psi_2'.$$

Further, because $x\colon !\tau$, $\psi_1 = |v\rangle_x \otimes \operatorname{drop}^{(x)}(\psi_1)$ and similarly $\psi_2 = |v\rangle_x \otimes \operatorname{drop}^{(x)}(\psi_2)$. Thus the claim

$$\psi_1^\dagger \psi_2 = \operatorname{drop}^{(x)}(\psi_1)^\dagger \operatorname{drop}^{(x)}(\psi_2) = \psi_1'^\dagger \psi_2'.$$

[Q] By the induction hypothesis, we know that there is an $\bar{f}'$ satisfying

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \, |\vec{v}\rangle_{\vec{x}} \otimes |\bar{f}'(\vec{v},\vec{w})\rangle_{\underline{e}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

hence $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(\vec{v},\vec{w})$ suffices.

[P] A QRAM can remove x from consideration (which has the semantics of $\mathrm{drop}^{(x)}(\cdot)$ for classical values), and then compute ψ' by the induction hypothesis.

A.6.2.5 *[W]*

The rule is

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{W}.$$

The general form for $\psi \in \iota\left(\llbracket \Gamma, \texttt{const}\ x\colon \tau, \Delta \rrbracket\right)$ is

$$\psi = \sum_{v,\vec{v},\vec{w}} \gamma_{v,\vec{v},\vec{w}} \left|v\right\rangle_x \otimes \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{v,\vec{v},\vec{w}}$$

[T] We can apply the induction hypothesis by considering $\texttt{const}\ x\colon \tau$ as part of the remainder $\Delta' = \texttt{const}\ x\colon \tau, \Delta$, yielding $\psi' \in \iota\left(\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta'\right)$, hence the claim.

[C] Similarly, this claim follows immediately by applying the induction hypothesis after grouping $\left|v\right\rangle_x$ with $\tilde{\psi}_{v,\vec{v},\vec{w}}$.

[M] The induction hypothesis immediately yields the claim.

[Q] Here $\bar{f}(v, \vec{v}, \vec{w}) := \bar{f}'(\vec{v}, \vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.

[P] A QRAM can compute ψ' by the induction hypothesis.

A.6.2.6 *[!C]*

The rule is

$$\frac{\left[\Gamma, x\colon !\tau, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \textbf{dup}_{x\to x,x}\left(\psi\right)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ !\text{C}.$$

The general form for $\psi \in \iota\left(\llbracket \Gamma, x\colon !\tau, \Delta \rrbracket\right)$ is

$$\psi = \left|v\right\rangle_x \otimes \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

[T] It is clear that $\texttt{dup}_{x\to x,x}(\psi) \in \iota\left(\llbracket \Gamma, x\colon !\tau, x\colon !\tau, \Delta \rrbracket\right)$. Applying the induction hypothesis to $\texttt{dup}_{x\to x,x}(\psi)$, yields that $\psi' \in \iota\left(\llbracket \texttt{const}\ \vec{x}\colon \vec{\tau}, e\colon \tau'', \Delta \rrbracket\right)$, hence the claim.

[C] The claim follows from the induction hypothesis.

[M] The induction hypothesis yields the claim.

[Q] Here $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(v,v,\vec{v},\vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.

[P] A QRAM can duplicate x (which has the semantics of **dup** for classical values) and then compute ψ' by the induction hypothesis.

A.6.2.7 *[C]*

The rule is

$$\frac{\left[\Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \texttt{dup}_{x\to x,x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \text{drop}^{(x)}\left(\psi'\right)}\ \text{C}.$$

The general form for $\psi \in \iota\left(\llbracket \Gamma, \texttt{const}\ x\colon \tau, \Delta \rrbracket\right)$ is

$$\psi = \sum_{v,\vec{v},\vec{w}} \gamma_{v,\vec{v},\vec{w}} \left|v\right\rangle_x \otimes \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \tilde{\psi}_{v,\vec{v},\vec{w}}.$$

[T] It is clear that

$$\texttt{dup}_{x\to x,x}(\psi) \in \iota\left(\llbracket \Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau, \Delta \rrbracket\right).$$

Thus, the induction hypothesis yields

$$\psi' \in \iota\left(\left\llbracket \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau, \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta \right\rrbracket\right).$$

The claim follows by applying $\text{drop}^{(x)}\left(\cdot\right)$ to ψ'.

[C] The induction hypothesis yields the claim.

[M] A straightforward calculation and the induction hypothesis yield the claim.

[Q] Similar to before, $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(v,v,\vec{v},\vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.

[P] A QRAM can duplicate x using the linear isometry $[\![\texttt{dup}]\!]$, yielding a state of the form $\sum_v \gamma_v \left|v\right\rangle_x \left|v\right\rangle_x \otimes \tilde{\psi}_v$. By the induction hypothesis, the QRAM can then compute ψ' of the form $\psi' = \sum_v \gamma_v \left|v\right\rangle_x \left|v\right\rangle_x \otimes \chi_v$ (Thm. 3.6.2). Hence, reversing **dup** yields

$$[\![\texttt{dup}]\!]^{-1}_{x,x\to x}(\psi') = \sum_v \gamma_v \left|v\right\rangle_x \otimes \chi_v = \mathrm{drop}^{(x)}(\psi') .$$

A.6.2.8 *[ite]*

The rule is depicted in Fig. A.11.

[T] We first consider quantum control flow ($e_c \colon \mathbb{B}$).

Using the induction hypothesis, we know that the state after evaluating the condition is

$$\psi' \in \iota\left(\left[\!\!\left[\Gamma_c, \underline{e_c}\colon \mathbb{B}, \Gamma, \Delta\right]\!\!\right]\right),$$

hence we can write

$$\psi' = \psi_t \otimes \left|1\right\rangle_{\underline{e_c}} + \psi_f \otimes \left|0\right\rangle_{\underline{e_c}} .$$

Next we show that $\mathbb{I}_{\underline{e_t}\to\underline{e}}(\psi'_t) + \mathbb{I}_{\underline{e_f}\to\underline{e}}\left(\psi'_f\right)$ is in

$$\iota\left(\left[\!\!\left[\texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau, \Delta\right]\!\!\right]\right).$$

The induction hypothesis yields that

$$\psi'_t \in \iota\left(\left[\!\!\left[\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e_t}\colon \tau, \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \Delta\right]\!\!\right]\right),$$
$$\psi'_f \in \iota\left(\left[\!\!\left[\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e_f}\colon \tau, \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \Delta\right]\!\!\right]\right).$$

Because τ does not have any classical components, the classical components of ψ'_t and ψ'_f coincide, hence they can be added. This yields, after renaming, the claim.

Next we consider classical control flow ($e_c \colon !\mathbb{B}$). It is clear that ψ' originating from

$$\left[\Gamma_c \overset{\alpha_c}{\vdash} e_c\colon !\mathbb{B} \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi',$$

where $\psi \in \iota\left(\llbracket \Gamma_c, \Gamma, \Delta \rrbracket\right)$, can be written as

$$\begin{aligned}\psi' &= \psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}} \\ &\in \iota\left(\left\llbracket \texttt{const } \vec{x}_c \colon \vec{\tau}_c, \underline{e_c} \colon !\mathbb{B}, \Gamma, \Delta \right\rrbracket\right).\end{aligned}$$

We assume w.l.o.g. $\underline{e_c}$ evaluates to true, thus the ψ_f part has amplitude 0.

The induction hypothesis yields that

$$\psi'_t \in \iota\left(\left\llbracket \texttt{const } \vec{x} \colon \vec{\tau}, \underline{e_t} \colon \tau, \texttt{const } \vec{x}_c \colon \vec{\tau}_c, \Delta \right\rrbracket\right),$$

which is exactly what we would like to have after renaming $\underline{e_t}$ to $\underline{e}$. The second summand can be neglected due to having 0 amplitude.

[C] From the induction hypothesis for e_c, we know that $\psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}}$ are of the correct form. Thus, due to the induction hypotheses for e_t and e_f, $\psi'_t + \psi'_f$ is of the correct form. This proves the claim, up to renaming of variables.

[M] First, we consider quantum control flow. The induction hypothesis on e_c yields that

$$\begin{aligned}\psi_1^\dagger \psi_2 &= \left(\psi_t^{1\dagger} \otimes \langle 1|_{\underline{e_c}} + \psi_f^{1\dagger} \otimes \langle 0|_{\underline{e_c}}\right)\left(\psi_t^2 \otimes |1\rangle_{\underline{e_c}} + \psi_f^2 \otimes |0\rangle_{\underline{e_c}}\right) \\ &= \psi_t^{1\dagger}\psi_t^2 + \psi_f^{1\dagger}\psi_f^2\end{aligned}$$

The induction hypothesis on e_t and e_f yields that

$$\begin{aligned}\psi_t^{1\dagger}\psi_t^2 &= \psi_t^{1'\dagger}\psi_t^{2'} \\ \psi_f^{1\dagger}\psi_f^2 &= \psi_f^{1'\dagger}\psi_f^{2'},\end{aligned}$$

thus

$$\psi_1^\dagger\psi_2 = \psi_t^{1\dagger}\psi_t^2 + \psi_f^{1\dagger}\psi_f^2 = \psi_t^{1'\dagger}\psi_t^{2'} + \psi_f^{1'\dagger}\psi_f^{2'}.$$

This proves the claim after renaming.

Next, we consider classical control flow. If the classical components of ψ_1 and ψ_2 coincide, then also $\underline{e_c}^1 = \underline{e_c}^2$. Using the induction hypothesis, we see that the term for e_c preserves the inner product between ψ_1 and ψ_2. Furthermore, because $\underline{e_c}\colon !\mathbb{B}$, we get $\psi_1^\dagger\psi_2 = \psi_t^{1\dagger}\psi_t^2$, w.l.o.g. assuming $\underline{e_c} = 1$. Thus with the induction hypothesis on the term for e_t, we get that $\psi_1^\dagger\psi_2 = \psi_t^{1\prime\dagger}\psi_t^{2\prime}$. Renaming does not change the inner product, hence the claim.

[Q] First, we consider quantum control flow. If **qfree** $\in \alpha$ then by the typing rule **qfree** $\in \alpha_c \cap \alpha_t \cap \alpha_f$.

Let $\Gamma_c = \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c$ and $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$. The general form of ψ is

$$\psi = \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle_{\vec{x}_c} \otimes \left|\vec{v}\right\rangle_{\vec{x}} \otimes \left|\vec{w}\right\rangle_{\vec{y}} \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}$$

Using the induction hypothesis, we get the functions $\bar{f}_{e_c}$, $\bar{f}_{e_t}$ and $\bar{f}_{e_f}$. For the next step we only suppress variable names that are not immediately clear to lighten the notation. Thus evaluating e_c yields

$$\begin{aligned}\psi' &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|\bar{f}_{e_c}(\vec{v}_c)\right\rangle_{\underline{e_c}} \otimes \left|\vec{v}\right\rangle \otimes \left|\vec{w}\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}} \\ &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|0\right\rangle_{\underline{e_c}} \otimes \left|\vec{v}\right\rangle \otimes \left|\vec{w}\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}} \\ &+ \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|1\right\rangle_{\underline{e_c}} \otimes \left|\vec{v}\right\rangle \otimes \left|\vec{w}\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}\end{aligned}$$

Further, evaluating e_t and e_f yields

$$\begin{aligned}\psi'' &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|0\right\rangle_{\underline{e_c}} \otimes \left|\vec{v}\right\rangle \otimes \left|f_{e_f}(\vec{v},\vec{w})\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}} \\ &+ \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|1\right\rangle_{\underline{e_c}} \otimes \left|\vec{v}\right\rangle \otimes \left|f_{e_t}(\vec{v},\vec{w})\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}.\end{aligned}$$

Thus the final state can be described by

$$\psi''' = \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}} \left|\vec{v}_c\right\rangle \otimes \left|\vec{v}\right\rangle \otimes \left|\bar{f}(\vec{v}_c,\vec{v},\vec{w})\right\rangle_{\underline{e}} \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}},$$

where $\bar{f}$ is defined by

$$\bar{f}(\vec{v}_c,\vec{v},\vec{w}) := \begin{cases}\bar{f}_{e_t}(\vec{v},\vec{w}) & \text{if } \bar{f}_{e_c}(\vec{v}_c) = 1 \\ \bar{f}_{e_f}(\vec{v},\vec{w}) & \text{otherwise.}\end{cases}$$

The proof works analogously for the case where $e_c\colon !\mathbb{B}$.

[P] For quantum control flow ($e_c\colon \mathbb{B}$), expression

$$\texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_f$$

is **mfree** (ensured by our type system), hence Lemma 1 applies.

For classical control flow ($e_c\colon !\mathbb{B}$), a QRAM can first evaluate e_c (by the induction hypothesis). Then, as $e_c\colon !\mathbb{B}$, by Thm. 3.6.1, its value is classical, meaning the QRAM can classically determine what this value is, and run the appropriate branch (by induction hypothesis). This yields the correct state up to renaming of variables.

A.6.3 *Proofs for Eval*

In order to prove our theorems for rules involving $\xrightarrow{\text{eval}}$, we strengthen our theorems to also cover the following:

For $\vec{e} = \vec{e}^{c}, \vec{e}^{\not c}$, split into constant and non-constant arguments, assume

$$\left[e'(\vec{e})\colon \bigtimes_{i=1}^{n} \alpha_i \tau_i ! \xrightarrow{\alpha} \tau' \;\middle|\; \sigma\colon \Gamma \;\middle|\; \psi \right] \xrightarrow{\text{eval}} \psi',$$

where the general form of ψ is

$$\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}^{c}}} \otimes |\vec{w}\rangle_{\underline{\vec{e}^{\not c}}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

Then, we have the following:

[T] If $\psi \in \iota\left(\left[\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta \right]\!\right]\right)$, then

$$\psi' \in \iota\left(\left[\!\left[\underline{\vec{e}^{c}}\colon \vec{\tau}^{c}, \underline{\text{ret}}\colon \tau', \Delta \right]\!\right]\right).$$

[C]

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}^{c}}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

[M] If **mfree** $\in \alpha$, $\rho \in \left[\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta \right]\!\right]^{c}$,

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \sigma\colon \Gamma \;\middle|\; \iota(\rho, \psi_1) \right] \xrightarrow{\text{eval}} \psi_1'$$

for $\psi_1 \in \mathcal{H}\left(\llbracket \underline{\vec{e}} \colon \vec{\tau}, \Delta \rrbracket^{\mathsf{q}}\right)$ and

$$\left[\Gamma \overset{\alpha}{\vdash} e \colon \tau'' \;\middle|\; \sigma \colon \Gamma \;\middle|\; \iota(\rho, \psi_2)\right] \xrightarrow{\text{eval}} \psi_2'$$

for $\psi_2 \in \mathcal{H}\left(\llbracket \underline{\vec{e}} \colon \vec{\tau}, \Delta \rrbracket^{\mathsf{q}}\right)$, then it holds that

$$\psi_1^\dagger \psi_2 = \psi_1'^\dagger \psi_2'.$$

[Q] If **`qfree`** $\in \alpha$, then there exists $\bar{f} \colon \llbracket \vec{\tau} \rrbracket^{\mathsf{s}} \to \llbracket \tau' \rrbracket^{\mathsf{s}}$, such that

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}}} \otimes |\bar{f}(\vec{v},\vec{w})\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

where $\bar{f}$ can depend on σ.

[P] Then there exists a QRAM implementing this, i.e., maps input $\psi \otimes |e'', \sigma\rangle_{\underline{e'}}$ to the correct output ψ'.

A.6.3.1 *[built-in-eval]*

We require that all built-ins behave correctly, thus no further reasoning is needed.

A.6.3.2 *[measure]*

The rule is provided in Fig. A.9. The general form of ψ is

$$\psi = \sum_{w} \gamma_w \, |w\rangle_{\underline{e}} \otimes \tilde{\psi}_w$$

[T] Let $w' \in \llbracket \tau \rrbracket^{\mathsf{c}} \times \llbracket \tau \rrbracket^{\mathsf{q}}$. Then immediately

$$\psi' = \gamma_{w'} \, |w'\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{w'} \in \iota\left(\llbracket \underline{\text{ret}} \colon !\tau, \Delta \rrbracket\right).$$

[C] The claim follows immediately from the semantics of **`measure`**.

[M] Nothing to prove as **`measure`** is not **`mfree`**.

[Q] Nothing to prove as **`measure`** is not **`qfree`**.

[P] Measuring the appropriate value yields the correct semantics.

A.6.3.3 *[rev]*

[T] We see that

$$\texttt{reverse}(e_{\text{func}})\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau_k''! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{j=1}^{m} \tau_j',$$

hence the claim.

[C] The claim follows immediately from the semantics of **reverse**.

[M] The classical components of ψ_1 and ψ_2 coincide, hence in particular the expression e and the captured values σ coincide. The non-classical part of ψ_1 and ψ_2 does not get modified, thus the inner product is preserved.

[Q] The appropriate $\bar{f}$ is

$$\bar{f}(e_{\text{func}}, \sigma) = (\texttt{reverse}(e_{\text{func}}), \sigma).$$

[P] A QRAM can prepare the correct state by purely classical operations, replacing e_{func} by $\texttt{reverse}(e_{\text{func}})$.

A.6.4 *[call-rev]*

[T] Using the induction hypothesis, we know that

$$\psi' \in \left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \vec{t}\colon \vec{t}', \Delta\right]\!\right]^{+}.$$

We need to show $\psi' \in \iota\left(\left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \vec{t}\colon \vec{\tau}', \Delta\right]\!\right]\right)$, then the claim follows immediately after renaming.

By contradiction: Let $\psi' \notin \iota\left(\left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \vec{t}\colon \vec{\tau}', \Delta\right]\!\right]\right)$, then there exists a classical component of ψ' which is in superposition. The typing rule of **reverse** enforces that the arguments and the return value of reversed function are not classical, hence the classical component in superposition needs to lie in context Δ. By the induction hypothesis (specifically [C]), we know that evaluating $e_{\text{func}(\vec{e}^{\text{c}}, \vec{t})}$ leaves Δ unchanged, hence the classical component in superposition is also a classical component in superposition of ψ, which is a contradiction to $\psi \in \iota\left(\left[\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\right]\right)$.

[C] We know that $\psi' \in \llbracket \underline{\vec{e}^{\mathrm{c}}} \colon \vec{\tau}^{\mathrm{c}}, \vec{t} \colon \vec{\tau}', \Delta \rrbracket^{+}$. Further, the linear map sending ψ' to ψ is

$$\sum_{\vec{v}} \left|\vec{v}\right\rangle \left\langle\vec{v}\right| \otimes M_{\vec{v};\vec{t}\to\underline{\mathrm{ret}}} \otimes \mathbb{I}_{\Delta},$$

where $M_v \colon \llbracket \vec{\tau}' \rrbracket \to \llbracket \vec{\tau}'' \rrbracket$ is an isometry. The map becomes unitary by restricting the codomain to its image, which can be inverted resulting in

$$\sum_{\vec{v}} \left|\vec{v}\right\rangle \left\langle\vec{v}\right| \otimes M^{-1}_{\vec{v};\vec{t}\to\underline{\mathrm{ret}}}\big|_{M_{v;t\to\underline{\mathrm{ret}}}(\llbracket \vec{\tau}' \rrbracket)} \otimes \mathbb{I}_{\Delta},$$

which preserves $\vec{e}^{\mathrm{c}} \colon \vec{\tau}^{\mathrm{c}}$ and Δ, hence the claim.

[M] As renaming does not change the inner product, this claim follows from the induction hypothesis.

[Q] Let **`qfree`** $\in \alpha$. By the induction hypothesis, we get that

$$\psi' = \sum_{\vec{v},\vec{w}'} \gamma_{\vec{v},\vec{w}'} \left|\vec{v}\right\rangle_{\vec{e}^{\mathrm{c}}} \otimes \left|\vec{w}'\right\rangle_{\vec{t}} \otimes \tilde{\psi}_{\vec{v},\vec{w}'}$$

and that there exists an $\bar{f}'$ such that after renaming $\underline{\mathrm{ret}}$ to $\vec{e}^{\not c}$ we have

$$\psi = \sum_{\vec{v},\vec{w}'} \gamma_{\vec{v},\vec{w}'} \left|\vec{v}\right\rangle_{\vec{e}^{\mathrm{c}}} \otimes \left|\bar{f}'(\vec{v},\vec{w}')\right\rangle_{\vec{e}^{\not c}} \otimes \tilde{\psi}_{\vec{v},\vec{w}'}.$$

We note that $\bar{f}$ is injective by Thm. 3.6.3, since non-injectivity would violate the semantics of the **`mfree`** e_{func} being a linear isometry. Thus, there exists a function $\bar{f} = \bar{f}'^{-1}$ satisfying $\bar{f}'^{-1}(\vec{v}, \bar{f}'(\vec{v},\vec{w}')) = \vec{w}'$, hence the claim.

[P] As e_{func} is **`mfree`**, a QRAM can implement it, according to Lemma 1. As e_{func} has no classical components in its type, the implementation depends only on e_{func}, not on classical components of the input state. Then, applying the reverse of the implementation to ψ yields the correct result (up to renaming).

A.6.4.1 *[eval-λ-abs]*

[T] We know that $\psi \in \iota\left(\llbracket \underline{\vec{e}} \colon \vec{\tau}, \Delta \rrbracket\right)$, thus

$$\psi \otimes \left|\sigma\right\rangle \in \iota\left(\llbracket \underline{\vec{e}} \colon \vec{\tau}, \Gamma, \Delta \rrbracket\right),$$

which leads to $\mathbb{I}_{\underline{\vec{e}}\to\vec{x}}(\psi \otimes |\sigma\rangle) \in \iota\left(\llbracket \vec{x}\colon \vec{\tau}, \Gamma, \Delta \rrbracket\right)$. The induction hypothesis yields now that

$$\psi' \in \iota\left(\llbracket \vec{\alpha}^c \vec{x}^c : \vec{\tau}^c, \underline{e''} : \tau', \Delta) \rrbracket \right),$$

thus after renaming, $\psi' \in \iota\left(\left\llbracket \vec{e}^c : \vec{\tau}^c, \underline{\mathrm{ret}} : \tau', \Delta \right\rrbracket\right)$.

[C] The claim follows from the induction hypothesis.

[M] It is immediate that $\psi_1^\dagger \psi_2 = \psi_1^\dagger \otimes \langle\sigma|\, \psi_2 \otimes |\sigma\rangle$. Further, renaming does not change the inner product, hence by the induction hypothesis, we get that $\psi_1^\dagger \psi_2 = \psi_1'^\dagger \psi_2'$. Renaming again leaves the inner product invariant, hence the claim.

[Q] The $\bar{f}$ obtained from the induction hypothesis behaves correctly, up to adding σ to the state and renaming variables.

[P] Given input $\psi \otimes |e'', \sigma\rangle_{\underline{e'}}$, a QRAM can rename variables ($\underline{\vec{e}} \to \vec{x}$), run e'' (by induction hypothesis), and rename variables in the result again.

A.6.4.2 *[func-eval]*

[T] After applying the induction hypothesis from left to right on all terms on top, we get

$$\psi_{n+2} \in \iota\left(\llbracket \underline{\vec{e}^c} : \vec{\tau}^c, \underline{\mathrm{ret}} : \tau', \Delta_{n+2} \rrbracket\right),$$

where Δ_{n+2} accumulated additionally to the Δ of ψ_0 all constant parts of $\vec{\Gamma}$ and Γ'. Thus

$$\mathrm{drop}^{(\underline{\vec{e}^c})}\left(\psi_{n+2}\right) \in \iota\left(\llbracket \underline{\mathrm{ret}} : \tau', \Delta_{n+2} \rrbracket\right),$$

which leads to

$$\mathbb{I}_{\underline{\mathrm{ret}}\to\underline{e'(\vec{e})}} \circ \mathrm{drop}^{(\underline{\vec{e}^c})}\left(\psi_{n+2}\right) \in \iota\left(\left\llbracket \underline{e'(\vec{e})} : \tau', \vec{\Gamma}^c, \Gamma'^c, \Delta \right\rrbracket\right).$$

[C] The claim follows from the induction hypotheses.

[M] Using the induction hypothesis iteratively, we get that $\psi_{0,1}^\dagger \psi_{0,2} = \psi_{i,1}^\dagger \psi_{i,2}$ for all $1 \le i \le n$. Using the induction hypothesis on the other parts yields $\psi_{0,1}^\dagger \psi_{0,2} = \psi_{n+2,1}^\dagger \psi_{n+2,2}$. The type system guarantees that sub-expressions which are not consumed, that is **`const`** $\in \alpha_i'$ are **`qfree`**,

	Silq			Q# reference solution			Q# average of top 10		
	S18	W19	Both	S18	W19	Both	S18	W19	Both
Lines of code	**99**	**168**	**267**	251	242	493	282.9	372.7	655.6
Quantum primitives	**8**	**10**	**10**	12	19	22	8.1	12.0	-
Annotations	2	3	3	3	6	6	**1.0**	4.0	-
Low-level quantum gates	**14**	**23**	**37**	33	54	87	38.2	102.9	141.1

Table A.1: Silq compared to Q#. Two entries in the last column are missing, because the top 10 contestants are not the same for both competitions and the number of used annotation and built-in and library functions where calculated per contestant.

and thus they can be uncomputed similarly to the case for ite where we uncomputed the expression e_c. Thus $\text{drop}^{(\underline{\vec{e}^c})}(\psi_{n+2})$ preserves the inner product, and hence the claim.

[Q] An appropriate composition of all $\bar{f}$ from the induction hypotheses, $\text{drop}^{(\underline{e_i})}$, and variable renamings yields the claim.

[P] We can evaluate all arguments, and determine the function itself by the induction hypothesis, yielding $\psi_{n+1} \otimes |e'', \sigma\rangle_{\underline{e'}}$. By the strengthened induction hypothesis, we have

$$\psi_{n+2} = \sum_{\vec{v},\vec{w}_1,\dots,\vec{w}_n} \gamma_{\vec{v}} |\vec{v}\rangle_{\vec{x}} \otimes \bigotimes_{\{i|\texttt{const}\in\alpha'_i\}} |\bar{f}_i(\vec{v},\vec{w}_i)\rangle_{\underline{e_i}} \otimes \tilde{\psi}_{\vec{v}},$$

where $\vec{x}$ consists of all constant variables in $\vec{\Gamma}, \Gamma'$. This is due to Thm. 3.6.4 (which ensures this holds after evaluating all arguments) and Thm. 3.6.2 (which ensures this form is preserved).

Hence, a QRAM can reverse $\bar{f}_i$, to implement $\text{drop}^{(\underline{e_i})}(\cdot)$ for each i with $\texttt{const} \in \alpha'_i$. This yields the correct result up to renaming of variables.

A.7 EVALUATION

A.7.1 *Evaluation against Q#*

In this section, we provide a detailed evaluation of the Q# Summer 2018 [63] and Winter 2019 [64] coding contests.

Table A.1 summarizes the comparison of our solutions written in Silq against the solutions written by (i) the Q# language designers and (ii) the respective top 10 contest participants. Our results demonstrate that Silq requires significantly fewer lines of code and only requires roughly half the built-in features and library functions. The remaining tables contain more detailed results.

LINES OF CODE When counting lines of code, we did not count empty lines, lines that only consist of comments, contain import or namespace statements or code that is unreachable for the solving operation.

QUANTUM PRIMITIVES AND ANNOTATIONS We counted both the number of quantum primitives and annotations. Note that annotations are called functors in Q#. The summary in Table A.1 shows how many quantum primitives and annotations were used *at least once*, measuring how many concepts a programmer needs to know.

LOW-LEVEL QUANTUM GATES We also counted low-level quantum gates, which are marked as ♣ in the detailed results. The summary in Table A.1 shows how many low-level quantum gates were used in total, measuring how often the programmer has to resort to low-level operations.

For Q#, we did not include the counts of operations like `ControlledOnInt`, as they are more high-level. For the same reason, for Silq, we did not include `phase`, `if`, or `forget`.

Further, we did not add the counts for `M` or `Measure` (Q#) or `measure` (Silq), because `measure` can be applied to any data structure, and is thus more high-level, but gets often used similarly to `M` in Q#.

TOP 10 CONTESTANTS In order to compare the Silq solutions against the solutions of the top 10 contestants of the Q# Summer 2018 and Winter 2019 coding contest, we evaluate the submissions of the top 10 contestants using the same methods as before. We provide detailed results in Table A.6, Table A.7, Table A.8, and Table A.9.

	Summer 2018															Winter 2019													
	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	D1	D2	D3	E1	E2	A1	A2	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5	D6	Sum
Quantum primitives																													
`ApplyToEach`								3						2			1								2				8
`ApplyToEachC`																										1			1
`ApplyToEachCA`																								1					1
`CCNOT`♠													3														1		4
`CNOT`♠		1	1	1							1	2							3		2				2		3	2	18
`ControlledOnBitString`																	2					2							4
`ControlledOnInt`																	1	1	1		2	1		1					7
`H`♠	1	1	1	1			2	1		1				3			1		2				1	1	1	1	2	1	21
`M`					1	1	2	2	1	2				1	1				1										12
`MResetZ`																			1										1
`MeasureInteger`																		1											1
`PrepareUniformSuperposition`																1													1
`R1`♠																		2											2
`ResetAll`														1	1														2
`ResultAsInt`							1	1																					2
`Ry`♠									1									1	1										3
`S`♠																			1										1
`SWAP`♠				1				1																				1	3
`With`								1																					1
`WithA`																						1							1
`X`♠			2	1				2				2		1	1		3	1	1	2	3	3				5	2	4	33
`Z`♠								1											1										2
Annotations																													
`Adjoint`																		1		1	1							1	4
`Controlled`				1				1											2	1				1		3	1	2	12
`adjoint self`								1													1								2
`adjoint auto`				1														2		2	1	2		2				1	11
`controlled auto`																		1						1		1			3
`controlled adjoint auto`																		1						1					2
Low-level quantum gates (marked by ♠)	1	2	4	4			2	5	1	1	1	4	3	4	1		4	4	9	2	5	3	1	1	3	6	8	8	87
Lines of code	9	12	32	24	12	16	9	19	11	28	11	15	9	23	21	3	20	21	30	18	27	19	3	12	5	21	10	53	493

Table A.2: Evaluation of the solutions provided by the Q# language designers for the Summer 2018 and Winter 2019 coding contest.

	Summer 2018															Winter 2019													
	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	D1	D2	D3	E1	E2	A1	A2	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5	D6	Sum
Quantum primitives																													
`dup`																												1	1
`forget`		1	1	1													1							2	1		2	1	10
`H`♠	1	1	1	1			2	2		2				2		2	1		2				1	1	1	1	2	1	24
`if`		1	1	1				1						1			3	3	5					1	2	2	2	4	27
`measure`					1	1	1	1	1	3				1	1	2		1	1										14
`phase`								1						1				2	2										6
`reverse`																		1											1
`rotY`♠									1									1	1										3
`X`♠				1														1	1						2	3	2		10
`[]`				1													1												2
Annotations																													
`mfree`																		1											1
`lifted`				1							1	1	1	1	1					1	1	1						1	10
`!` (classical)	1	2	2	1	1	1					2	2		3	3		2	2		1	1	1	1	1	1	1		16	45
Low-level quantum gates (marked by ♠)	1	1	1	2			2	2	1	2				2		2	1	2	4				1	1	3	4	4	1	37
Lines of code	5	6	12	12	3	9	4	5	3	7	7	7	7	7	5	10	10	17	15	7	11	7	4	15	18	17	15	22	267

Table A.3: Evaluation of the Silq solutions for Q# Summer 2018 and Winter 2019 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
Quantum primitives											
`ApplyToEach`			4				3	5	5	5	2.2
`BoolArrFromResultArr`									3		0.3
`BoolFromResult`									1		0.1
`CCNOT`♠	6	3		3		3		12	3	3	3.3
`CNOT`♠	7	8	9	7	7	9	8	9	6	6	7.6
`H`♠	14	11	12	12	13	13	12	14	13	12	12.6
`IsResultZero`									1		0.1
`M`	12	11	12	16	12	18	10	12	3	12	11.8
`MultiM`									6		0.6
`R`♠										1	0.1
`Reset`								3			0.3
`ResetAll`	2		2		5		3	2	2	2	1.8
`ResultAsInt`			5					1	2		0.8
`Ry`♠	1	1	1	2	3	3	1	1	2		1.5
`SWAP`♠			1							2	0.3
`X`♠	7	12	15	5	42	11	9	10	6	10	12.7
`Z`♠			1							1	0.2
Annotations											
`Controlled`		2	5	1	8		3		2	2	2.3
`controlled auto`		1							1	1	0.3
Quantum primitives (number of non-zero rows)	7	6	10	6	6	6	7	10	13	10	8.1
Annotations (number of non-zero rows)		2	1	1	1		1		2	2	1.0
Low-level quantum gates (marked by ♠)	35	35	39	29	65	39	30	46	30	34	38.2
Lines of code	181	312	259	313	313	387	228	271	280	285	282.9

Table A.4: Summer 2018: Overview of the evaluation of the Q# solution provided by the top 10 contestants.

A.7.2 *Evaluation against Quipper*

In order to compare the amount of features, we counted the definitions provided in Quipper's core library [1] and list them by rubric and type in Table A.10.

1 https://www.mathstat.dal.ca/~selinger/quipper/doc/Quipper.html

	1	2	3	4	5	6	7	8	9	10	average
Quantum primitives											
`ApplyPauliFromBitString`				4							0.4
`ApplyToEach`				2				7	6		1.5
`ApplyToEachA`								8	3		1.1
`ApplyToEachC`				1							0.1
`ApplyToEachCA`	1			5					7	6	1.9
`CCNOT`♠	6	10		1	2	9	31		1	4	6.4
`CNOT`♠	5	15	12	14	20	17	13	23	18	15	15.2
`ControlledOnBitString`	2			6						2	1.0
`ControlledOnInt`	4			6							1.0
`H`♠	13	11	13	14	13	10	13	12	13	15	12.7
`IntegerIncrementLE`				2						4	0.6
`M`		3	3		12	3	5	4		8	3.8
`Measure`								1			0.1
`MeasureInteger`				1					2		0.3
`MultiM`	2			1							0.3
`MultiX`	9			2							1.1
`QFT`										1	0.1
`R1`♠			2	2	2				5		1.1
`Reset`	1			1	2			1			0.5
`ResetAll`			1							3	0.4
`ResultAsInt`	2			1							0.3
`Rx`♠				1			1				0.2
`Ry`♠	2	3	3	2	3	4	4	8	6	2	3.7
`Rz`♠		4		1		5	1	2		2	1.5
`S`♠	2		1		1			1			0.5
`SWAP`♠		1	1	1		1		5	1	1	1.1
`StatePreparationComplexCoefficients`	2									1	0.3
`StatePreparationPositiveCoefficients`										1	0.1
`WithA`									1		0.1
`X`♠	18	62	50	36	98	65	119	64	58	27	59.7
`Z`♠	1	1	2		4						0.8
Annotations											
`Adjoint`	2	1	8	2		1		2		3	1.9
`Controlled`	14	26	27	25	37	27	30	57	36	12	29.1
`adjoint self`	1			3				1			0.5
`adjoint auto`	2	5	23	15	5	8	3	4	9	8	8.2
`controlled auto`		2	10	28		2		4	6		5.2
`controlled adjoint auto`				2		2			1	4	0.9
Quantum primitives (number of non-zero rows)	15	9	10	21	10	8	8	12	12	15	12.0
Annotations (number of non-zero rows)	4	4	4	6	2	5	2	5	4	4	4.0
Low-level quantum gates (marked by ♠)	47	107	84	72	143	111	182	115	102	66	102.9
Lines of code	163	322	461	298	367	358	543	610	323	282	372.7

TABLE A.5: Winter 2019: Overview of the evaluation of the Q# solution provided by the top 10 contestants.

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`							1	1			0.2
`H`	1	1	1	1	1	1	1	1	1	1	1.0
`ResetAll`					1						0.1
Lines of code	7	11	9	10	10	10	5	7	12	10	9.1

(a) Summer 18: A1

Rank	1	2	3	4	5	6	7	8	9	10	average
`CNOT`	1	1	1	1	1	1	1	1	1	1	1.0
`H`	1	1	1	1	1	1	1	2	1	1	1.1
`M`				1							0.1
`X`				1							0.1
Lines of code	11	17	12	24	12	23	11	16	24	20	17.0

(b) Summer 18: A2

Rank	1	2	3	4	5	6	7	8	9	10	average
`CNOT`	1	2	1	1	2	1	1	1	1	2	1.3
`H`	1	1	1	1	1	1	1	1	1	2	1.1
`M`				1							0.1
`X`	2	3	3	2	5	1	1	3	2	2	2.4
Lines of code	24	46	38	35	39	30	22	33	40	42	34.9

(c) Summer 18: A3

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`										1	0.1
`CCNOT`	3							2			0.5
`CNOT`	1	1	1	1		3	2	1		1	1.1
`H`	1	1	1	1			1	1		1	0.7
`M`				1							0.1
`Ry`					2	2			1		0.5
`SWAP`										2	0.2
`X`	1	3	1	1	2	1	3	1	1	3	1.7
`Controlled`		2	1	1	1		2		1	1	0.9
`controlled auto`		1							1	1	0.3
Lines of code	24	25	22	46	17	19	25	24	13	47	26.2

(d) Summer 18: A4

Rank	1	2	3	4	5	6	7	8	9	10	average
`BoolFromResult`									1		0.1
`M`	1	1	1	1	1	1	1	1		1	0.9
`MultiM`									1		0.1
`ResetAll`							1				0.1
`X`							1				0.1
Lines of code	8	16	12	12	13	16	25	13	9	13	13.7

(e) Summer 18: B1

Rank	1	2	3	4	5	6	7	8	9	10	average
`BoolArrFromResultArr`									1		0.1
`CNOT`								1			0.1
`H`								1			0.1
`M`	2	1	1	1	1	1	1	1		2	1.1
`MultiM`									1		0.1
Lines of code	9	22	18	17	16	24	15	17	19	14	17.1

(f) Summer 18: B2

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`			1								0.1
`H`	2	2	1	2	2	2	2	2	2	2	1.9
`M`	2	2	2	3	2	2	2	2		2	1.9
`MultiM`									1		0.1
`ResultAsInt`			2					1	1		0.4
Lines of code	10	20	10	20	16	34	10	9	15	15	15.9

(g) Summer 18: B3

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`										2	0.2
`CNOT`	1	1		1		1	1	1	1		0.7
`H`	4	2	2	3	3	2	2	2	2	2	2.4
`M`	2	2	2	3	2	2	2	2		2	1.9
`MultiM`									1		0.1
`ResetAll`					1						0.1
`ResultAsInt`			1						1		0.2
`SWAP`			1								0.1
`X`					6					2	0.8
`Z`			1							1	0.2
`Controlled`			1		1					1	0.3
Lines of code	13	21	12	24	26	35	14	17	13	18	19.3

(h) Summer 18: B4

Table A.6: Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.

Rank	1	2	3	4	5	6	7	8	9	10	average
IsResultZero									1		0.1
M	1	1	1	1	1	1	1	1	1	1	1.0
R										1	0.1
ResultAsInt			1								0.1
Ry	1	1	1	1	1	1	1	1	1		0.9
Lines of code	5	15	9	12	14	14	10	11	13	11	11.4

(a) Summer 18: C1

Rank	1	2	3	4	5	6	7	8	9	10	average
H	1	1	2		2	2	1	2	1	1	1.3
M	2	2	2	2	3	5	1	3	2	2	2.4
Reset								2			0.2
ResetAll			1		1						0.2
ResultAsInt			1								0.1
Ry				1							0.1
X						2					0.2
Controlled			1								0.1
Lines of code	10	26	22	20	29	40	13	23	24	18	22.5

(b) Summer 18: C2

Rank	1	2	3	4	5	6	7	8	9	10	average
CNOT	1	1	1	1	1	1	1	1	1	1	1.0
Lines of code	7	13	12	11	11	13	10	12	14	12	11.5

(c) Summer 18: D1

Rank	1	2	3	4	5	6	7	8	9	10	average
ApplyToEach							2				0.2
CNOT	2	2	2	2	2	2	2	2	2	1	1.9
X	2	2	2	1	2	2	2	2	1	2	1.8
Lines of code	13	18	17	14	16	19	17	23	19	16	17.2

(d) Summer 18: D2

Rank	1	2	3	4	5	6	7	8	9	10	average
ApplyToEach			2					2			0.4
CCNOT	3	3		3		3		10	3	3	2.8
CNOT			3		1			1			0.5
X			5		26		1	3			3.5
Controlled			2		6		1				0.9
Lines of code	7	9	14	9	33	9	9	22	10	9	13.1

(e) Summer 18: D3

Rank	1	2	3	4	5	6	7	8	9	10	average
ApplyToEach								2	2	2	0.6
BoolArrFromResultArr									1		0.1
H	3	2	3	3	3	4	3	2	2	2	2.7
M	1	1	2	1	1	3	1	1		1	1.2
MultiM									1		0.1
Reset								1			0.1
ResetAll	1				1		1		1	1	0.5
X	1	3	3		1	3	1	1	1	1	1.5
Lines of code	19	30	33	32	30	45	20	25	27	21	28.2

(f) Summer 18: E1

Rank	1	2	3	4	5	6	7	8	9	10	average
ApplyToEach			1						3		0.4
BoolArrFromResultArr									1		0.1
H									3		0.3
M	1	1	1	1	1	3	1	1		1	1.1
MultiM									1		0.1
ResetAll	1		1		1		1	2	1	1	0.8
X	1	1	1			2			1		0.6
Controlled									1		0.1
Lines of code	14	23	19	27	31	56	22	19	28	19	25.8

(g) Summer 18: E2

Table A.7: Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`				1							0.1
`ApplyToEachC`				1							0.1
`CNOT`									1		0.1
`H`	1	1		2	2	1	1	1	1		1.0
`M`					2						0.2
`Reset`				1							0.1
`Ry`	1	1	1			1	1	1	1		0.7
`StatePreparationPositiveCoefficients`										1	0.1
`X`	1	2	8	1	2	1	2	2			1.9
`Controlled`	1	1	2	2	1	1	1	1	1		1.1
Lines of code	5	6	27	10	21	5	6	8	6	6	10.0

(a) Winter 19: A1

	1	2	3	4	5	6	7	8	9	10	average
`ApplyPauliFromBitString`				4							0.4
`ApplyToEach`				1					6		0.7
`CCNOT`		4				4	4			1	1.3
`ControlledOnBitString`	2			4							0.6
`ControlledOnInt`	1										0.1
`H`	2	2	2	1	2	2	2		2	4	1.9
`M`					1					3	0.4
`ResetAll`										2	0.2
`Ry`								2			0.2
`X`	3	18	3	8	23	18	18	5	20	5	12.1
`Controlled`		4	1	4	8	4	4	2	8		3.5
Lines of code	22	66	78	19	72	65	89	63	56	33	56.3

(b) Winter 19: A2

	1	2	3	4	5	6	7	8	9	10	average
`CCNOT`		1		1		1					0.3
`CNOT`				1	2		3	4	2		1.2
`H`				1	1		1		1		0.4
`M`		1	1		2	1	3	3		3	1.4
`MeasureInteger`									1		0.1
`MultiM`	1			1							0.2
`MultiX`	4										0.4
`R1`			2	2	2				2		0.8
`ResultAsInt`	1			1							0.2
`Ry`		1	1	1	1	1	1	2	1		0.9
`Rz`		2				2	1	2			0.7
`S`	1										0.1
`StatePreparationComplexCoefficients`	2									1	0.3
`X`		4	6	1	2	4		1			1.8
`Z`	1										0.1
`Adjoint`	2		1	1				1		1	0.6
`Controlled`	2	4	2	1	1	4	2	1	1		1.8
`adjoint auto`			1	1		2					0.4
`controlled auto`		1				1					0.2
`controlled adjoint auto`						1					0.1
Lines of code	16	24	86	18	11	33	13	28	11	18	25.8

(c) Winter 19: B1

	1	2	3	4	5	6	7	8	9	10	average
`CNOT`			4	5	4			3	2		1.8
`H`	2		2	2	2		1	2	2	4	1.7
`M`		2	2		6	2	2	1		2	1.7
`Measure`								1			0.1
`MeasureInteger`				1					1		0.2
`MultiM`	1										0.1
`QFT`										1	0.1
`R1`									3		0.3
`Reset`	1				2			1			0.4
`ResetAll`			1							1	0.2
`ResultAsInt`	1										0.1
`Rx`				1			1				0.2
`Ry`	1	1	1	1	1	1	2	1	1	1	1.1
`Rz`		2		1		3				2	0.8
`S`	1		1		1			1			0.4
`SWAP`									1		0.1
`X`		1	3	1	5	2	1	2	6		2.1
`Z`		1	2		4						0.7
`Controlled`	1	4	3	2	6	4	1	1	3	3	2.8
`adjoint auto`			4	2		2				1	0.9
`controlled auto`		1	3	1		1					0.6
`controlled adjoint auto`				1		1				1	0.3
Lines of code	12	34	44	28	50	48	27	31	30	40	34.4

(d) Winter 19: B2

	1	2	3	4	5	6	7	8	9	10	average
`CNOT`		2	2			4			2		1.0
`ControlledOnBitString`				2						1	0.3
`ControlledOnInt`	1										0.1
`X`	4	1	2	2	6	2	5	5	1	4	3.2
`Adjoint`			1								0.1
`Controlled`	1	1	1		2	1	2	2	1		1.1
`adjoint self`								1			0.1
`adjoint auto`	1	1	2	1	1	1	1		1	1	1.0
`controlled auto`				2							0.2
Lines of code	13	13	25	23	28	26	25	23	13	19	20.8

(e) Winter 19: C1

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachA`								6	3		0.9
`CCNOT`							24				2.4
`CNOT`		4	2	2	2	4		4	2	2	2.2
`ControlledOnBitString`										1	0.1
`ControlledOnInt`	1			2							0.3
`WithA`									1		0.1
`X`	1	10	6	3	7	11	60	10	6	1	11.5
`Adjoint`			1								0.1
`Controlled`		3	2		2	3	5	3	2		2.0
`adjoint self`	1			2							0.3
`adjoint auto`		1	2		2	1	1	1	3	1	1.2
`controlled auto`				4							0.4
Lines of code	19	46	22	33	31	48	116	46	34	17	41.2

(f) Winter 19: C2

TABLE A.8: Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachA`								2			0.2
`ApplyToEachCA`										2	0.2
`CCNOT`	5	5				4				1	1.5
`CNOT`							2		4		0.6
`ControlledOnInt`				4							0.4
`IntegerIncrementLE`										1	0.1
`M`					1						0.1
`SWAP`								4		1	0.5
`X`	4	12	6	5	10	13	4	6	7	4	7.1
`Adjoint`			1	1				1		1	0.4
`Controlled`			5	3	3	1	2	3	3	1	2.1
`adjoint self`				1							0.1
`adjoint auto`	1	1	3	2	2	1	1	2	3	3	1.9
`controlled auto`			2	3				2	4		1.1
`controlled adjoint auto`				1						1	0.2
Lines of code	20	27	40	32	35	31	33	36	39	31	32.4

(a) Winter 19: C3

	1	2	3	4	5	6	7	8	9	10	average
`H`	1	1	1	1	1		1	1	1	1	0.9
`Ry`						1					0.1
`adjoint auto`			1	1							0.2
`controlled auto`				2							0.2
Lines of code	3	3	7	8	3	3	3	4	3	3	4.0

(b) Winter 19: D1

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`	1			1					1	2	0.5
`ControlledOnInt`	1										0.1
`H`	1	1	1	1	1	1	1		1	1	0.9
`Ry`								1			0.1
`X`		2	2	2	2	2	2		2	2	1.6
`Controlled`		1	2	2	1	1	1	1	1	1	1.1
`adjoint auto`			1	1							0.2
`controlled auto`			1	2							0.3
Lines of code	5	14	16	13	14	17	14	9	10	9	12.1

(c) Winter 19: D2

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`				2					2		0.4
`CNOT`		2	2		2	2	2	2		2	1.4
`H`	1	1	1	1	1	1	1	1	1	1	1.0
`MultiX`	2										0.2
`X`				2					2		0.4
`Controlled`	2			2					2		0.6
`adjoint auto`			1	1							0.2
`controlled auto`				2							0.2
Lines of code	5	10	14	12	10	11	11	11	6	9	9.9

(d) Winter 19: D3

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`								6			0.6
`ApplyToEachCA`				2					4		0.6
`CNOT`		2	1		3	2	2	8		4	2.2
`H`	1	1	1	1	1	1	1	4	1	1	1.3
`IntegerIncrementLE`				2						2	0.4
`MultiX`	3			2							0.5
`X`	1	7	5	4	10	7	7	15	8	3	6.7
`Adjoint`			2								0.2
`Controlled`	3	2	3	4	4	2	4	7	7	2	3.8
`adjoint auto`			4	3							0.7
`controlled auto`			2	6					1		0.9
Lines of code	10	29	44	38	35	28	51	153	25	22	43.5

(e) Winter 19: D4

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`								1			0.1
`CCNOT`	1				2		3		1	2	0.9
`CNOT`	3	5	1	4	6	5	2	2	4	4	3.6
`H`	3	3	3	3	2	3	2	3	3	3	2.8
`IntegerIncrementLE`										1	0.1
`SWAP`		1	1	1		1		1			0.5
`X`	2	2	7	5	22	2	5	5	2	2	5.4
`Adjoint`			1								0.1
`Controlled`	3	4	3	4	5	4	1	3	3	4	3.4
`adjoint auto`			2	3							0.5
`controlled auto`				6							0.6
Lines of code	11	13	25	38	17	13	14	19	13	15	17.8

(f) Winter 19: D5

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`										2	0.2
`CNOT`	2			2	1		2		1	3	1.1
`H`	1	1	2	1		1	2				0.8
`Ry`					1			1	3	1	0.6
`X`	2	3	2	2	9	3	15	13	4	6	5.9
`Adjoint`		1	1			1				1	0.4
`Controlled`	1	2	3	1	4	2	7	33	4	1	5.8
`adjoint auto`		2	2			1		1	2	2	1.0
`controlled auto`			2					2	1		0.5
`controlled adjoint auto`									1	2	0.3
Lines of code	22	37	33	26	40	30	141	179	77	60	64.5

(g) Winter 19: D6

Table A.9: Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.

	funcs	data	type	class
The Circ monad		1		
Basic types		2	2	
Basic gates	76		1	
Other circuit-building functions	17			
Notation for controls	4	1		1
Signed items	2	1		
Comments and labelling	4			1
Hierarchical circuits	4			
Block structure	17			
Operations on circuits	17	2		
Circuit transformers	3	2	5	
Circuit generation from classical code	2			
Extended quantum data types	8			8
Sum	154	9	8	10

TABLE A.10: The number of functions, data types, types and classes provided by Quippers core library in the respective category.

A.7.3 *Further Algorithms*

In the following, we provide further algorithms implemented in Silq.

```
// Wiesner’s quantum money: Conjugate coding, Stephen Wiesner, https://dl.acm.org/
     citation.cfm?id=1008920

def create_bill[n:!ℕ](){
    // generate new bill and verifier
    secret:=uniform[4,n]();
    bill:=encode(secret)(0:uint[n]);
    verifier:=λ(b:uint[n]). verify(b,secret);
    return (bill,verifier);
}

def verify[n:!N](bill:uint[n],secret:!ℕ^n):uint[n]×!𝔹{
    // verify a given bill
    check:=reverse(encode(secret))(bill);
    if measure(check)==0{ // ok, give money back
        return (encode(secret)(0:uint[n]),true);
    }else{ // forged!
        return (0:uint[n],false);
    }
}

// ENCODING FUNCTIONS

def encode[n:!ℕ](secret:!ℕ^n)(bill:uint[n])mfree{
    for k in [0..n){
        bill[k]:=encode_𝔹[secret[k]](bill[k]);
    }
    return bill;
}
def encode_𝔹[state:!ℕ](b:𝔹)mfree{
    // 0↦0, 1↦1, 2↦+, 3↦-
    if state%2==1{ b:=X(b); }
    if state>=2  { b:=H(b); } // switch basis to +/-
    return b;
}

// SIMPLE TEST

def verify_new_test[n:!ℕ](){
    // verify a new bill twice

    // create new bill
    (bill,verifier):=create_bill[n]();
    // verify twice it is genuine
    (bill,ok1):=verifier(bill);
    assert(ok1);
    (bill,ok2):=verifier(bill);
    assert(ok2);
    // discard the bill
    measure(bill);
}
def main(){
    verify_new_test[3]();
}

// HELPER FUNCTIONS

def uniform[range:!ℕ,length:!ℕ](){
    // returns (x1,...,x_length) with xi~{0,...range-1}
    n:=round(log(range)/log(2)) coerce !ℕ;
    assert(2^n==range);
    r:=vector(length,0:!ℕ);
    for l in [0..length){
        for k in [0..n){
            r[l]+=2^k*rand();
        }
    }
    return r;
}
def rand(){
    // quantum number generator
    return measure(H(false));
}
```

```
import quantum_money;

// PRIMITIVE FORGE ATTEMPT
// The attempt does not work due to the no-cloning theorem

def forge_primitive[n:!ℕ](bill:uint[n]){
    forged:=dup(bill);
    return (bill,forged);
}

// SIMPLE TEST

def forge_primitive_test[n:!ℕ](){
    // create new bill
    (bill,verifier):=create_bill[n]();
    // try to duplicate it
    (bill,forged):=forge_primitive(bill);
    // verify both
    (bill,ok_original):=verifier(bill);
    (forged,ok_forged):=verifier(forged);
    assert(!ok_original || !ok_forged);
    // discard bills
    measure(bill);
    measure(forged);
}
def main(){
    forge_primitive_test[4]();
}
```

```
// An adaptive attack on Wiesner’s quantum money, https://arxiv.org/abs/1404.1507

import quantum_money;

def forge_nagaj[n:!ℕ](bill:uint[n],verifier:uint[n]! → uint[n]×!𝔹){
    secret:=vector(n,0);
    for k in [0..n){
        (bill,is_plus):=determine(bill,verifier,k,true);
        if is_plus{
            secret[k]=2;
        }else{
            (bill,is_minus):=determine(bill,verifier,k,false);
            if is_minus{
                secret[k]=3;
            }else{
                secret[k]=measure(bill[k]);
            }
        }
    }
    return (bill, encode(secret)(0:uint[n]));
}

def determine[n:!ℕ](bill:uint[n],verifier:uint[n]! → uint[n]×!𝔹,k:!ℕ,check_plus:!𝔹):
  uint[n]×!𝔹{
    // determine the value of the k-th bit of the quantum bill
    // - check_plus=true: return 1 iff bit is plus
    // - check_plus=false: return 1 iff bit is minus
    fail_prob:=0.01;
    N:=ceil(π^2*n/(2*fail_prob)); // choose N
    if N%2==1{ N+=1; } // ensure N is even
    // choose δ
    δ:=π/(2*N);

    probe:=0:𝔹;
    repeat N{
        probe:=rotY(δ*2,probe); // rotate slightly towards 1
        if probe{ // entangle
            bill[k]:=X(bill[k]);
            if !check_plus{ phase(π); }
        }
        (bill,ok):=verifier(bill); // project back by verification
        assert(ok==1); // we should not be caught
    }
    return (bill, measure(probe));
}

// SIMPLE TEST
def forge_nagaj_test[n:!ℕ](){
    // create a new bill
    (bill,verifier):=create_bill[n]();
    // forge
    (bill,forged):=forge_nagaj(bill,verifier);
    // verify both bills
    (bill,ok_original):=verifier(bill);
    assert(ok_original);
    (forged,ok_forged):=verifier(forged);
    assert(ok_forged);
    // discard both bills
    measure(bill);
    measure(forged);
}

def main(){
    forge_nagaj_test[2]();
}
```

B

APPENDIX: UNQOMP

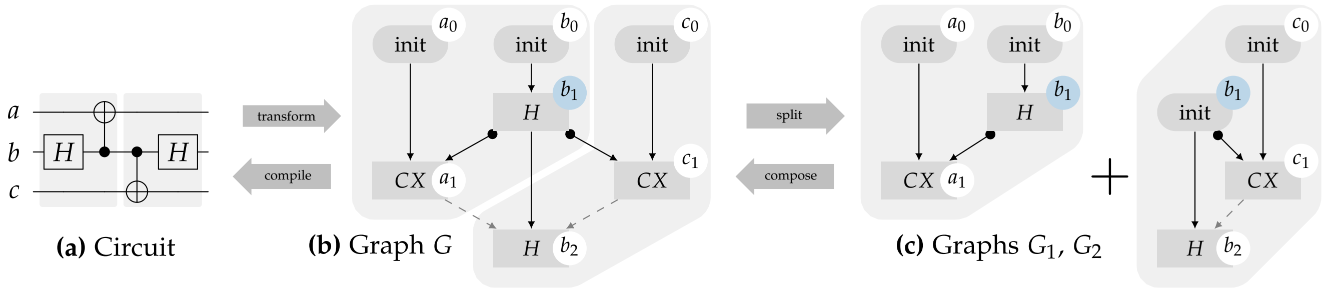


Figure B.1: Operations on circuit graphs.

B.1 OPERATIONS ON CIRCUIT GRAPHS

In this section, we introduce the operations of circuit graph splitting and composition. These are key operations in our proof of Thm. 4.2.1 of Unqomp (see App. B.2).

B.1.1 *Splitting*

At a high level, the split operation divides a circuit graph G into two graphs G_1 and G_2 such that all gates in G_2 can be evaluated after G_1. The graph G_2 is extended by missing init nodes for all qubits involved in G_2.

SPLITTING EXAMPLE Fig. B.1c shows the graphs G_1 and G_2 obtained by splitting G along the indicated cut. The graph G_1 is simply the subgraph induced by $\{a_0, b_0, a_1, b_1\}$. The graph G_2 includes the node b_1 because of the edge $b_1 \rightarrow b_2$ in G. While b_1 is a gate node in G_1, it is an init node in G_2. The anti-dependency edge $a_1 \dashrightarrow b_2$ is not present in neither graph.

SPLITTING FORMALIZATION More formally, consider an arbitrary graph cut (V', V'') in the circuit graph $G = (V, E)$, such that there are no edges in E from a node in V'' to a node in V'. That is, we partition the nodes V into V' and V'' such that all nodes in V'' come after all nodes in V' according to the partial order induced by E. Fig. B.1b illustrates such a cut for $V' = \{a_0, b_0, a_1, b_1\}$ and $V'' = \{c_0, c_1, b_2\}$.

The operation $\textsc{Split}(G, V', V'')$ first checks whether the given cut satisfies the above constraints and returns an error if not. Otherwise, it returns two graphs G_1 and G_2, where G_1 is the subgraph induced by V', and G_2 is the graph induced by $P \cup V''$, where $P \subseteq V'$ are the control and target parents of nodes in V''. The nodes P are init nodes in G_2. Formally, the graph $G_2 = (V^{(2)}, E^{(2)})$ is built as follows.

$$\begin{aligned}
V^{(2)}_{\text{gates}} &= V_{\text{gates}} \cap V'' \\
V^{(2)}_{\text{init}} &= (V_{\text{init}} \cap V'') \cup P \\
P &:= \{u \in V' \mid \exists v \in V''.\ (u \rightarrow v) \in E \ \vee \ (u \bullet\!\!\rightarrow v) \in E\} \\
E^{(2)} &\text{ are the edges from } E \text{ induced by } V^{(2)}_{\text{init}} \cup V^{(2)}_{\text{gates}}
\end{aligned}$$

We note that for $u \in P$, u is an init node in G_2 but may be a gate node in G_1. In both graphs, $\text{qbit}(u)$ is the same qubit.

B.1.2 *Composing*

The composition operation $G_1 \bullet G_2$ can be thought of as the inverse of SPLIT. It composes two graphs $G_1 = (V^{(1)}, E^{(1)})$ and $G_2 = (V^{(2)}, E^{(2)})$ such that init nodes in G_2 are merged with the "output nodes" in G_1, and any missing anti-dependency edges are introduced.

COMPOSING EXAMPLE Fig. B.1b shows the graph G obtained by composing G_1 and G_2. Node $b_1 \in V^{(1)}_{\text{gate}}$ is merged with $b_1 \in V^{(2)}_{\text{init}}$ to the gate node $b_1 \in V_{\text{gate}}$ (see highlighted). This corresponds to evaluating the H gate in G_1 first, before using the result as an input to the H gate in G_2. The anti-dependency edge $a_1 \dashrightarrow b_2$ is not present in G_1 or G_2, but introduced during composition.

COMPOSING FORMALIZATION For any node u, we say that u is a *last* node if u is the last node along the target edge path including u. In other words, u is the last gate node targeting $\text{qbit}(u)$. Upon composing G_1 and G_2, we merge the last nodes in G_1 with the corresponding init node in G_2 (if any such node exists). More formally, we union the graphs and perform an edge contraction along the following set of edges E': [1]

$$E' = \left\{ (u, v_0) \in V^{(1)} \times V^{(2)}_{\text{init}} \;\middle|\; \begin{array}{l} \text{qbit}(u) = \text{qbit}(v_0), \\ u \text{ is a last node} \end{array} \right\}.$$

When contracting an edge (u, v_0), we retain the node u and discard v_0. In particular, if u is a gate node in G_1, it is a gate node in G. The resulting set of edges E is extended by any missing anti-dependency edges.

SEMANTICS OF COMPOSED GRAPHS A key property of the composition operation is the natural composition of semantics. More precisely, it is $[\![G_1 \bullet G_2]\!] = [\![G_2]\!] \circ [\![G_1]\!]$ for any circuit graphs G_1, G_2. This follows from the definition of circuit graph semantics, that allows any total order respecting the edges of $G_1 \bullet G_2$ when defining its semantics. As the definition of composition makes sure that there are no edges from nodes in $V^{(2)}$ to nodes in $V^{(1)}$, we can pick a total order on $G_1 \bullet G_2$ such that all nodes in G_1 come before all nodes in G_2. The composition of the gates semantics then yields $[\![G_1 \bullet G_2]\!] = [\![G_2]\!] \circ [\![G_1]\!]$.

1 Technically, we would need to temporarily rename nodes occurring in both graphs, such that the graph union leaves the graphs disconnected.

B.2 CORRECTNESS PROOF OF UNQOMP

In the following, we provide a proof for Thm. 4.2.1:

Theorem 4.2.1 (Correctness). *Let* $\textsc{Unqomp}(G, A) = \mathcal{G}$ *for circuit graph G with n qubits of which m are ancilla qubits. Without loss of generality, assume that those ancillae* $A = \left(a^{(1)}, \dots, a^{(m)}\right)$ *are the first m qubits of G. If*

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k\in\{0,1\}^m} \gamma_k \, |k\rangle_A \otimes \phi_k, \textit{ then} \tag{4.4}$$

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![\mathcal{G}]\!]} \sum_{k\in\{0,1\}^m} \gamma_k \, |0\cdots0\rangle_A \otimes \phi_k. \tag{4.5}$$

Proof. Our proof proceeds by induction on the number of gate nodes in G and relies on the operations from App. B.1.

BASE CASE In the base case, G and therefore also $\mathcal{G}$ consist of zero gate nodes, and Thm. 4.2.1 follows because the left-hand side and the right-hand side of Eq. (4.4) and Eq. (4.5) are all the same.

INDUCTION STEP For the induction step, we consider graph G with $n+1$ gate nodes. We then select the first gate node v according to the total order in Line 3, and split G according to $(G_v, G_{-v}) = \textsc{Split}(G, \{v\}, V\backslash\{v\})$. Note that the induction hypothesis holds for G_{-v} with $\textsc{Unqomp}(G_{-v}, A) = \mathcal{G}_{-v}$, as G_{-v} consists of n gate nodes.

CASE I If $\text{qbit}(v) \notin A$, then $[\![v]\!]$ preserves A, and hence

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![v]\!]} |0\cdots0\rangle_A \otimes \chi \xmapsto{[\![G_{-v}]\!]} \sum_{k\in\{0,1\}^n} \gamma_k \, |k\rangle_A \otimes \phi_k.$$

We then use the induction hypothesis to show Eq. (4.5) by

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![v]\!]} |0\cdots0\rangle_A \otimes \chi \xmapsto{[\![\mathcal{G}_{-v}]\!]} \sum_k \gamma_k \, |0\cdots0\rangle_A \otimes \phi_k.$$

CASE II If $\text{qbit}(v) \in A$, then $v = a_n$ and Unqomp inserted a gate node $a^\star_{n-1}$ into $\mathcal{G}$. In the following, we refer to $a^\star_{n-1}$ as $v^\dagger$, because $v^\dagger$ uncomputes v.

Then, consider a split of $\mathcal{G}$ into subgraphs G_v (containing gate node v), $\mathcal{G}_{\square v^\dagger}$ (containing gate nodes before $v^\dagger$), $\mathcal{G}_{v^\dagger}$ (containing gate node $v^\dagger$), and

$\mathcal{G}_{v^\dagger\square}$ (containing all other gate nodes). Omitting gate nodes inserted by Unqomp from $\mathcal{G}_{v^\dagger\square}$ and $\mathcal{G}_{v^\dagger\square}$ yields graphs $G_{v^\dagger\square}$ and $G_{v^\dagger\square}$, respectively. Overall, this yields two decompositions of G and $\mathcal{G}$, respectively:

$$G = G_v \bullet G_{-v} = G_v \bullet G_{\square v^\dagger} \qquad\quad \bullet\, G_{v^\dagger\square} \text{ and} \tag{B.1}$$

$$\mathcal{G} \qquad\qquad\quad = G_v \bullet \mathcal{G}_{\square v^\dagger} \bullet \mathcal{G}_{v^\dagger} \bullet \mathcal{G}_{v^\dagger\square}. \tag{B.2}$$

First, we show the semantics of G, according to the graph split of G in Eq. (B.1). Here, a is the qubit targeted by v, C are non-ancilla control qubits of v, A' are ancilla qubits controlling v, and A'' are other ancilla qubits. Further, to avoid notational clutter, we write $\mathbf{0}$ for $0 \cdots 0$ in the following.

$$\sum_i \gamma_i \qquad |i\rangle_C \ |\mathbf{0}\rangle_{A'}\, |0\rangle_a \qquad |\mathbf{0}\rangle_{A''} \otimes \varphi_i^{(1)} \tag{B.3}$$

$$\xmapsto{[\![G_v]\!]} \sum_i \gamma_i \qquad |i\rangle_C \ |\mathbf{0}\rangle_{A'}\, |f_{i,\mathbf{0}}(0)\rangle_a\, |\mathbf{0}\rangle_{A''} \otimes \varphi_i^{(1)} \tag{B.4}$$

$$\xmapsto{[\![G_{\square v^\dagger}]\!]} \sum_i \gamma_i \qquad \psi_i \tag{B.5}$$

$$\xmapsto{[\![G_{v^\dagger\square}]\!]} \sum_i \gamma_i \sum_{i'jkl} \gamma^{(3)}_{ii'jkl}\, |i'\rangle_C\, |j\rangle_{A'}\ |k\rangle_a \qquad |l\rangle_{A''} \otimes \varphi^{(3)}_{ii'jkl} \tag{B.6}$$

Eq. (B.4) follows by observing that v is qfree and controlled by C and A'. Eq. (B.5) and Eq. (B.6) describe arbitrary quantum states that can be generated from Eq. (B.4) by a linear transformations.

Now, we show the semantics of $G_v \bullet \mathcal{G}_{\square v^\dagger} \bullet \mathcal{G}_{v^\dagger\square}$:

$$\sum_i \gamma_i \qquad |i\rangle_C \ |\mathbf{0}\rangle_{A'}\, |0\rangle_a \qquad |\mathbf{0}\rangle_{A''} \otimes \varphi_i^{(1)} \tag{B.7}$$

$$\xmapsto{[\![G_v]\!]} \sum_i \gamma_i \qquad |i\rangle_C \ |\mathbf{0}\rangle_{A'}\, |f_{i,\mathbf{0}}(0)\rangle_a\, |\mathbf{0}\rangle_{A''} \otimes \varphi_i^{(1)} \tag{B.8}$$

$$\xmapsto{[\![\mathcal{G}_{\square v^\dagger}]\!]} \sum_i \gamma_i \qquad |i\rangle_C \otimes \chi_i \tag{B.9}$$

$$\xmapsto{[\![\mathcal{G}_{v^\dagger\square}]\!]} \sum_i \gamma_i \sum_{i'jkl} \gamma^{(3)}_{ii'jkl}\, |i'\rangle_C\, |\mathbf{0}\rangle_{A'}\, |f_{i,\mathbf{0}}(0)\rangle_a\, |\mathbf{0}\rangle_{A''} \otimes \varphi^{(3)}_{ii'jkl} \tag{B.10}$$

Eq. (B.8) follows analogously to Eq. (B.4). Eq. (B.9) is an arbitrary quantum state generated from Eq. (B.8) by preserving C. We note that C must be preserved, as $\mathcal{G}_{\square v^\dagger}$ only contains nodes occurring before $v^\dagger$, and no gate node targeting a control qubit C can come before $v^\dagger$. To get to Eq. (B.10), we use the induction hypothesis and Eq. (B.6). To apply the induction hypothesis here, we first need to show that $\mathcal{G}_{\square v^\dagger} \bullet \mathcal{G}_{v^\dagger\square} = \mathcal{G}_{-v}$. Using Lemma 2, we can pick a total order on G such that v is first. We can then assume that this total order was used to get $\mathcal{G}$, and the same total order minus v was used by Algorithm 1, Line 3 to get $\mathcal{G}_{-v}$ from G_{-v}. Each step of uncomputation on G and G_{-v} would then have been exactly the same up until the uncomputation of v that inserted $v^\dagger$ in $\mathcal{G}$, that is to say $\mathcal{G}_{-v}$ is exactly $\mathcal{G}$ without v and $v^\dagger$, and all edges pointing to and from those nodes i.e., $\mathcal{G}_{-v} = \mathcal{G}_{\square v^\dagger} \bullet \mathcal{G}_{v^\dagger \square}$.

In order to apply the induction hypothesis in the case where $f_{i,\mathbf{0}}(0) \neq 0$, we strengthen Thm. 4.2.1 to not only hold for ancillae initialized to $0 \cdots 0$, but for arbitrary bit strings $b_1 \cdots b_n \in \{0,1\}^n$ (our proof naturally generalizes to this case).

Next, we observe that $\mathcal{G}_{v^\dagger \square}$ neither targets a (as $v^\dagger$ is the last node operating on a) nor A'. The latter is because all ancilla controls of v must be init nodes a'_0 (as v is the first node). Now, if a'_0 is not the last node on qubit a', it must be targeted by a gate node a'_1, such that $v \dashrightarrow a'_1$. According to Line 3, a'_1 has then been uncomputed before v, and $v^\dagger$ is hence controlled by $a_0'^\star$, which is the last operation on qubit a'.

From this, we conclude that χ_i must be of the form

$$\chi_i = \sum_l \gamma_{il}^{(4)} |\mathbf{0}\rangle_{A'} |f_{i,\mathbf{0}}(0)\rangle_a |l\rangle_{A''} \otimes \varphi_{il}^{(4)}. \tag{B.11}$$

This allows us to derive the semantics of $\mathcal{G}$ by continuing from Eq. (B.9) as:

$$\xmapsto{[\![\mathcal{G}_{\square v^\dagger}]\!]} \sum_i \gamma_i \sum_l \gamma_{il}^{(4)} \quad |i\rangle_C \ |\mathbf{0}\rangle_{A'} \ |f_{i,\mathbf{0}}(0)\rangle_a \ |l\rangle_{A''} \ \otimes \varphi_{il}^{(4)} \tag{B.12}$$

$$\xmapsto{[\![\mathcal{G}_{v^\dagger}]\!]} \sum_i \gamma_i \sum_l \gamma_{il}^{(4)} \quad |i\rangle_C \ |\mathbf{0}\rangle_{A'} \ |0\rangle_a \qquad |l\rangle_{A''} \ \otimes \varphi_{il}^{(4)} \tag{B.13}$$

$$\xmapsto{[\![\mathcal{G}_{v^\dagger \square}]\!]} \sum_i \gamma_i \sum_{i'jkl} \gamma_{ii'jkl}^{(3)} \, |i'\rangle_C \ |\mathbf{0}\rangle_{A'} \ |0\rangle_a \qquad |\mathbf{0}\rangle_{A''} \ \otimes \varphi_{ii'jkl}^{(3)} \tag{B.14}$$

Here, Eq. (B.12) follows by plugging Eq. (B.11) into Eq. (B.9). Then, Eq. (B.13) follows because gate$(v^\dagger)$ = gate$(v)^\dagger$ inverts the actions of v. Finally, Eq. (B.14) follows analogously to Eq. (B.10), by observing that no gate node in $\mathcal{G}_{v^\dagger \square}$ relies on the value of a.

Observing that Eq. (B.14) demonstrates Eq. (4.5) concludes the proof. □

Lemma 2. *For a given circuit graph and set of ancillae, for any choice of total order in Algorithm 1, Line 3, the result of the Unqomp procedure will be the same.*

Proof. We first prove a restriction of this lemma: for a given circuit graph and set of ancillae, for two total orders in Algorithm 1 that only differ by a swap of two adjacent nodes, the result of the Unqomp procedure will be the same.

Take such a circuit graph G, set of ancillae A and two total orders $<_1$ and $<_2$. $<_1$ and $<_2$ are the same except for two nodes v and w: $v <_1 w$ but $w <_2 v$. We now consider the application of Unqomp using those two total orders on G and A. Up to the uncomputation of v and w, both applications of the procedure are exactly the same and fail iff the other fails. Besides, as both $<_1$ and $<_2$ respect the edges of G yet order v and w differently, we know that there is no path in G between v and w. Specifically, v and w target different qubits, and cannot be controls of one another. Hence in both applications of the procedure, $v^\star, w^\star$ and their updated *ctrl* sets are the same. After the uncomputation of both v and w, the resulting graphs hence have the same nodes and set of control and target edges. They are exactly the same, and have cycles iff the other has one as well. The rest of the uncomputation is then again the same, concluding the proof of the restricted lemma.

Using the restricted lemma, we can prove the more general one: for any two total orders on G, we can go from one to the other through swap of adjacent nodes, and applying the restricted lemma at each step.

$\square$

B.3 EVALUATION DETAILS

For the code comparison, no comments or blank lines were counted. All programs contain the initialization of quantum registers and circuits, and an extra line for uncomputation when necessary.

For circuit size comparison to Qiskit, the parameters for each of the examples are:

- Adder: 12 qubits for each operand, and 12 for the result as well
- Deutsch-Jozsa: 10 control qubits, MCX as an oracle, returning true iff the value is 1111111111;
- Grover's algorithm: 10 control qubits, MCX as an oracle, returning true iff the value is 1111111111;
- IntegerComparator: 12 control qubits, comparing to $i = 40$;
- MCRY: 12 control qubits, rotation angle $\theta = 2$;
- MCX: 12 control qubits;
- Multiplier: 12 qubits for each operand, and 12 for the result as well

- PiecewiseLinearR: 12 control qubits, function breakpoints are [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 14], both coefficients and offsets are [1, 2, 3, 4, 3, 4, 3, 4, 5, 6, 4];
- PolynomialPauliR: 8 control qubits, polynomial coefficients are [1, 2, 3, 4, 5, 4, 1, 2, 3, 4, 5];
- WeightedAdder: 12 control qubits, values for sum are [1, 2, 3, 2, 5, 6, 5, 3, 4, 5, 8, 2];

For circuit size comparison to Quipper, the parameters for each of the examples are:

- Adder: 4 qubits for each operand, and 4 for the result as well
- Deutsch-Jozsa: 10 control qubits, MCX as an oracle, returning true iff the value is 1111111111;
- Grover's algorithm: 10 control qubits, MCX as an oracle, returning true iff the value is 1111111111;
- IntegerComparator: 4 control qubits, comparing to $i = 4$;
- MCX: 10 control qubits;
- Multiplier: 4 qubits for each operand, and 4 for the result as well
- WeightedAdder: 4 control qubits, 6 for the output, values for sum are $[15, 15, 15, 15]$;

The complete gate and qubits counts used in Table 4.2 are shown in Table B.1 and Table B.2.

TABLE B.1: Number of gates and qubits in Qiskit and Unqomp circuits

Benchmark	**Qiskit library**			**Unqomp**		
	qubits	all gates	CX gates	qubits	all gates	CX gates
Adder	36	706	310	36	464	200
Deutsch-Jozsa	19	181	54	19	181	54
Grover	19	8562	2550	19	8562	2550
IntegerComparator	24	390	126	24	270	66
MCRY	23	49144	24568	24	202	68
MCRY *	23	392	132	24	202	68
MCX	23	195	66	23	195	66
Multiplier	60	10812	4656	59	6972	2868
PiecewiseLinearR	35	14082	4926	25	8328	2870
PolynomialPauliR	18	76361	37192	16	14863	5124
PolynomialPauliR *	18	26729	9244	16	14863	5124
WeightedAdder	24	5406	2394	27	3090	1086
WeightedAdder *	24	4446	1626	27	3090	1086
WeightedAdder alt. impl.	24	5406	2394	24	3738	1302
WeightedAdder alt. impl. *	24	4446	1626	24	3738	1302

TABLE B.2: Number of gates and qubits in Quipper and Unqomp circuits.

Benchmark	**Quipper**			**Unqomp**		
	qubits	all gates	CX gates	qubits	all gates	CX gates
Adder	23	388	150	19	171	57
Deutsch-Jozsa	20	293	109	19	181	54
Grover	20	14262	5150	19	8562	2550
IntegerComparator	8	111	37	8	66	18
MCX	20	271	109	19	159	54
Multiplier	29	552	226	19	692	284
WeightedAdder	87	2630	948	19	1300	448
WeightedAdder alt. impl.	87	2630	948	16	1372	472

C APPENDIX: ABSTRAQT

C.1 ABSTRACT TRANSFORMERS SOUNDNESS

Here, we prove the soundness of the trace transformer Eq. (5.38):

Theorem C.1.1 (Trace)**.** *For all $\boldsymbol{\rho} \in \mathbb{D}$ we have $\gamma \circ tr(\boldsymbol{\rho}) \supseteq tr \circ \gamma(\boldsymbol{\rho})$.*

Proof. The over-approximation $\mathfrak{F}^{\#}$ follows closely the form of $\mathfrak{F}$, where the first term $\mathfrak{f}(\boldsymbol{P})$ over-approximates the prefactors of $\boldsymbol{P}$ and second term over-approximates the prefactors originating from the solution space for y of $\mathfrak{b}(\boldsymbol{P}) = \mathfrak{b}(\prod_{j=1}^{n} Q_j^{y_j})$. Overall, we have:

$$
\begin{aligned}
&\operatorname{tr} \circ \gamma(\boldsymbol{\rho}) \\
=&\operatorname{tr}\left(\left\{\sum_{i=1}^{r} c_i P_i \prod_{j=1}^{n} \tfrac{1}{2}\left(\mathbb{I} + (-1)^{b_{ij}} Q_j\right) \,\middle|\, c_i \in \boldsymbol{c}, P_i \in \boldsymbol{P}, b_{ij} \in \boldsymbol{b}_j\right\}\right) \\
=&\left\{\operatorname{tr}\left(\sum_{i=1}^{r} c_i P_i \prod_{j=1}^{n} \tfrac{1}{2}\left(\mathbb{I} + (-1)^{b_{ij}} Q_j\right)\right) \,\middle|\, c_i \in \boldsymbol{c}, P_i \in \boldsymbol{P}, b_{ij} \in \boldsymbol{b}_j\right\} \\
=&\left\{\sum_{i=1}^{r} \Re\left(c_i \mathrm{i}^{\mathfrak{F}(P,Q,b_{ij})}\right) \,\middle|\, c_i \in \boldsymbol{c}, P_i \in \boldsymbol{P}, b_{ij} \in \boldsymbol{b}_j\right\} && \text{Concrete trace, §5.5.4} \\
=&\sum_{i=1}^{r} \Re\left(\{c_i \in \boldsymbol{c}\} \cdot \mathrm{i}^{\mathfrak{F}(\{P_i \in \boldsymbol{P}\},Q,\{b_{ij} \in \boldsymbol{b}_j\})}\right) \\
\subseteq&\gamma\left(\sum_{i=1}^{r} \Re\left(\boldsymbol{c} \cdot \mathrm{i}^{\mathfrak{F}^{\sharp}(\boldsymbol{P},Q,\boldsymbol{b}_j)}\right)\right) && \text{Soundness of transf.} \\
=&\gamma\left(r \cdot \Re\left(\boldsymbol{c} \cdot \mathrm{i}^{\mathfrak{F}^{\sharp}(\boldsymbol{P},Q,\boldsymbol{b}_j)}\right)\right) && \text{Property of intervals} \\
=&\gamma \circ \operatorname{tr}(\boldsymbol{\rho}).
\end{aligned}
$$

□

Table C.1: States stabilized by Pauli matrices P and also $-P$, where $X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $Y := \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix}$, $Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, and $\mathbb{I}_2 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Stab.	**State vec.**	**Dens. mat.**	**Stab.**	**State vec.**	**Dens. mat.**
X	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \mathrel{\widehat{=}} \lvert + \rangle$	$\frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$-X$	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} \mathrel{\widehat{=}} \lvert - \rangle$	$\frac{1}{2}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$
Y	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ \mathrm{i} \end{pmatrix}$	$\frac{1}{2}\begin{pmatrix} 1 & \mathrm{i} \\ -\mathrm{i} & 1 \end{pmatrix}$	$-Y$	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -\mathrm{i} \end{pmatrix}$	$\frac{1}{2}\begin{pmatrix} 1 & -\mathrm{i} \\ \mathrm{i} & 1 \end{pmatrix}$
Z	$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \mathrel{\widehat{=}} \lvert 0 \rangle$	$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$	$-Z$	$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \mathrel{\widehat{=}} \lvert 1 \rangle$	$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$
$\mathbb{I}_2$	(any vec.)	-	$-\mathbb{I}_2$	(no vec.)	-

Table C.2: Multiplication of Pauli matrices.

$\mathbb{I}\mathbb{I} = \mathbb{I}$	$\mathbb{I}X = X$	$\mathbb{I}Y = Y$	$\mathbb{I}Z = Z$
$X\mathbb{I} = X$	$XX = \mathbb{I}$	$XY = \mathrm{i}Z$	$XZ = -\mathrm{i}Y$
$Y\mathbb{I} = Y$	$YX = -\mathrm{i}Z$	$YY = \mathbb{I}$	$YZ = \mathrm{i}X$
$Z\mathbb{I} = Z$	$ZX = \mathrm{i}Y$	$ZY = -\mathrm{i}X$	$ZZ = \mathbb{I}$

C.2 STABILIZERS AND PAULI MATRICES

Table C.1 shows the states stabilized by each Pauli matrix, together with the density matrix of the stabilized state. Further, Table C.2 shows the multiplication table for the Pauli matrices.