ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SECOND CYCLE DEGREE IN

ARTIFICIAL INTELLIGENCE

MASTER THESIS IN

Quantum Computing

EVALUATING QUANTUM ALGORITHM RESOURCE REQUIREMENTS THROUGH THE QuRA TOOLSET

SUPERVISOR

Prof. Ugo Dal Lago

DEFENDED BY

Davide Sonno

CO-SUPERVISOR

Andrea Colledan, PhD

Graduation Session October 2025

Academic year 2024/2025

Abstract

Quantum computing promises significant advantages over classical computing for specific problems, yet current quantum devices remain highly resource-constrained, limiting the size and complexity of executable programs. Accurate estimation of quantum resources, such as qubit count, circuit depth, and gate usage, is therefore essential for program feasibility analysis and optimization.

This thesis extends the `QuRA` tool, which builds on *Proto-Quipper*, a functional quantum programming language with a strong type system for static resource analysis. Beyond type-checking and symbolic estimation of circuit size, we formalize an evaluation semantics that translates well-typed *Proto-Quipper* programs into a Circuit Representation Language (CRL), enabling explicit circuit construction and validation of inferred resource bounds.

Finally, we develop a translation pipeline from CRL to `OpenQASM 3.0`, bridging high-level program analysis with hardware-executable code. The resulting framework provides both formal guarantees of correctness and practical outputs, supporting early feasibility analysis, validation of parametric bounds, and integration with existing compilation toolchains. This work lays the foundation for further extensions, including tighter resource estimates, dynamic analysis, and broader backend compatibility.

Acknowledgements

I would like to sincerely thank my supervisor, Ugo Dal Lago, for giving me the opportunity to work on this thesis in the fascinating field of quantum computing, and Andrea Colledan for his immense patience, guidance, and valuable advice throughout the process.

I am deeply grateful to my family and friends for their constant support. Thank you for always being there for me — and for putting up with me.

Finally, a special thanks goes to my cats, who kept me company by peacefully sleeping beside me while I worked on this thesis.

Contents

List of Figures

List of Tables

List of Programs

Chapter 1

Introduction

1.1 Quantum Computing; an Overview

Quantum computing is an emerging paradigm of computation that takes advantage of the principles of quantum mechanics to process information in fundamentally novel ways. While classical computers represent and manipulate information using *bits*, which can take values of either 0 or 1, quantum computers operate on *qubits*. Unlike classical bits, qubits can exist in *superpositions* of states, allowing them to represent and process multiple possibilities simultaneously [29].

Beyond superposition, two additional quantum phenomena are central to quantum computation. The first is *entanglement*, a uniquely quantum correlation between qubits in which the state of one qubit cannot be described independently of the other, even when they are spatially separated [29, 5]. Entanglement enables powerful forms of information processing and communication that do not have a classical analog. The second is *interference*, which allows quantum amplitudes to combine constructively or destructively. Quantum algorithms exploit this effect to amplify desirable computational paths while canceling out incorrect ones, thereby enabling certain problems to be solved more efficiently than in classical settings [19].

The conceptual foundations of quantum computing have evolved over several decades. As early as 1976, Roman Ingarden [26] extended Shannon's classical information theory [37] into the quantum domain, laying the groundwork for quantum information science. In 1980, Paul Benioff [4] demonstrated how a Turing machine [44] could be modeled within the framework of quantum mechanics, thereby proving that computation could be performed reversibly on quantum systems. Two years later, Richard Feynman [19] highlighted the fundamental

limitations of classical computers in simulating quantum systems and proposed the idea of using *quantum computers* to overcome these challenges. Building on these insights, David Deutsch formalized the notion of a *universal quantum computer* [17] in 1985, generalizing the classical Turing machine model and showing that a single quantum device could simulate any physical process. These foundational contributions set the stage for the development of quantum algorithms and error-correction techniques, ultimately transforming quantum computing from a theoretical curiosity into a rapidly advancing field of research.

1.2 Motivation

Quantum computing holds the promise of achieving a form of computational power that surpasses the capabilities of classical systems. By exploiting superposition, entanglement, and interference, quantum computers can, in principle, tackle problems that are currently intractable. This potential is often referred to as *quantum advantage*, the point at which a quantum algorithm outperforms the best known classical alternative for a specific task.

One of the earliest demonstrations of this potential came with Shor's algorithm [38], introduced in 1994, which provides an exponential speedup for factoring large integers. This result has profound implications for modern cryptography, as many widely used encryption schemes, such as RSA [35], rely on the hardness of integer factorization. Another landmark result is Grover's algorithm [23], which achieves a quadratic speedup for unstructured search problems. In addition, quantum simulation, originally proposed by Feynman [19], stands out as one of the most promising applications: simulating quantum systems efficiently on classical hardware is prohibitively expensive, but quantum computers are naturally suited for this task.

Despite this profound potential, the current state of quantum hardware presents significant limitations. Today's devices face challenges such as high error rates, short coherence times, and a restricted number of logical qubits. These constraints make it crucial to accurately estimate the resources (specifically, the number of qubits and operations) required by quantum algorithms. Such estimates are essential to determine the feasibility of running these algorithms on existing and near-future quantum hardware.

Historically, understanding the resource requirements of quantum algorithms has often relied on manual estimations or classical simulations of the generated circuits [48, 24].

However, these methods are slow, prone to errors, and difficult to scale for complex programs. They also primarily focus on individual circuits after they have been generated, failing to address the parametric nature of quantum algorithms, which produce families of circuits whose size depends on classical input parameters. This gap highlights a critical need for static analysis techniques [9] that can rigorously analyze resource consumption at the program level, yielding parametric and concrete upper bounds.

1.3 Related Work

The growing interest in quantum computation has led to the development of several tools and languages aimed at constructing, simulating, and optimizing quantum programs. Among the earliest and most influential is `Quipper` [22], a scalable, functional programming language for describing quantum circuits. `Quipper` provides powerful abstractions for circuit construction and supports resource estimation through circuit generation and analysis. Similarly, `ProjectQ` [42] offers a modular `Python` framework for quantum programming, featuring compiler toolchains and backend integration.

More recently, `t|ket⟩` [40], developed by Cambridge Quantum Computing, focuses on circuit optimization and hardware-aware compilation, while Microsoft's `Q#` [43] integrates resource estimation libraries directly into its ecosystem. IBM's `Qiskit` [32] provides one of the most widely used imperative frameworks for quantum programming, supporting simulation, compilation, and execution on real devices. Google's `Cirq` [18] emphasizes low-level, hardware-oriented circuit construction, targeting near-term quantum processors. Finally, `PennyLane` [6] introduces a differentiable programming approach, bridging quantum computation with machine learning frameworks such as TensorFlow and PyTorch.

Most existing toolchains still adopt a common design philosophy: they prioritize qubits and quantum states as the primary objects of computation, while circuit structure itself only emerges as a byproduct after program execution or compilation. This makes it difficult to reason about circuit properties *a priori*, especially in the case of parametric programs that generate entire families of circuits depending on the input size.

To compute resource metrics, many existing tools rely on *dynamic resource analysis*, where circuits are explicitly generated and subsequently analyzed. Although effective for small-scale or fixed-size circuits, this methodology becomes prohibitively expensive for

parametric algorithms, as a new circuit must be constructed and analyzed for each possible input size. By contrast, state-of-the-art *static resource analysis* tools operate directly on the program's source code, deriving symbolic expressions for resource usage that are valid across all input sizes. Static analysis thus offers an attractive alternative for large-scale or parameterized quantum algorithms.

Within this landscape, `QuRA` represents the state of the art in type-based static resource analysis. It extends the functional quantum programming language *Proto-Quipper* with refinement types and effect annotations, enabling the derivation of parametric and sound upper bounds on resources such as width, depth, and gate count, without ever generating a concrete circuit.

Our contribution builds directly on `QuRA`: we extend the framework with an operational semantics that evaluates well-typed programs into explicit circuit representations, formalized in the Circuit Representation Language (CRL). This extension allows circuits to be treated as first-class entities, enabling not only static resource estimation but also circuit validation, visualization, and translation into hardware-agnostic formats such as `OpenQASM 3.0`. In this way, our work bridges the gap between symbolic type-driven analysis and backend execution.

1.4 Contributions of this Thesis

This thesis contributes to the state of the art in three ways:

1. **Formal Evaluation Engine for *Proto-Quipper* Programs.** We formalize the translation of programs written in *Proto-Quipper*, `QuRA`'s programming language, into a structured intermediate representation, the *Circuit Representation Language (CRL)*. We design and implement an interpreter for CRL expressions that reduces programs to an *operations buffer*, a sequential, low-level description of the quantum operations performed. This extension transforms `QuRA` from a purely static analysis tool into a system capable of explicit circuit construction, enabling big-step evaluation that simultaneously builds circuits while executing programs.

2. **Validation of Resource Estimates.** We rigorously validate `QuRA`'s inferred resource bounds by comparing them with the concrete resource metrics on generated circuits. This evaluation spans both non-parametric circuits, such as teleportation, and parametric families, such as the quantum Fourier transform, Grover's algorithm, and Shor's

algorithm. The analysis highlights both the correctness and the tightness of QuRA's estimates, demonstrating their reliability across a diverse range of algorithms.

3. **Conversion to Executable OpenQASM 3.0.** We construct a translation pipeline from QuRA's circuit representation into OpenQASM 3.0, a widely adopted, hardware-agnostic standard for quantum circuits. This step enables circuit visualization with Qiskit, structural correctness checks, and preparation for execution on simulators or hardware backends. In the process, we introduce QASM-specific resource metrics to better capture QASM's semantics regarding initialization and measurement.

The interpreter is implemented in Haskell as an open-source project, currently available at davidesonno/qura on GitHub, pending integration into the official QuRA repository.

1.5 Thesis Structure

Chapter 2 reviews the fundamental concepts of quantum computing and quantum programming languages, as well as a brief overview of the λ-calculus and Haskell.

The remainder of the thesis is structured into two parts:

Part I: Evaluation and Validation of PQ Programs

- **Chapter 3** introduces Quipper and the *Proto-Quipper* family of languages, motivating the choice of *Proto-Quipper-RA* as the basis for the QuRA tool.
- **Chapter 4** extends *Proto-Quipper-RA* with an evaluation process that produces *configurations*, each pairing a program expression with its corresponding circuit. This enables programs to be reduced with a big-step semantics while simultaneously constructing their circuit representation.
- **Chapter 5** validates QuRA's inferred resource bounds against the measured metrics of generated circuits, assessing both correctness and precision across a range of case studies.

Part II: Conversion of Produced Circuits to QASM

- **Chapter 6** presents the translation of QuRA circuits into OpenQASM 3.0, introducing QASM-specific metrics to account for initialization and measurement costs.

- The resulting circuits are visualized and verified using `Qiskit`, with representative examples such as the quantum Fourier transform and Grover's algorithm.
- **Chapter 7** concludes the thesis, summarizing contributions and outlining possible directions for future work.

Chapter 2

Preliminaries

Quantum computing promises computational advantages over classical approaches by exploiting quantum-mechanical phenomena such as superposition and entanglement. To analyze the cost and efficiency of quantum algorithms, it is crucial to understand the mathematical model of quantum computation, how quantum states are manipulated via circuits, and how resource metrics such as qubit usage, circuit depth, and gate counts are derived. This chapter introduces the foundational concepts underpinning both quantum computation and the formal tools used in this thesis.

2.1 Basics of Quantum Computing

Unlike classical computation, which operates on deterministic binary states, quantum computation is fundamentally probabilistic and uses *qubits* as its basic units of information. Quantum programs are described using sequences of unitary transformations (quantum gates) and measurements, which form quantum circuits [29].

2.1.1 Qubits

A *qubit* is the quantum analogue of a classical bit, but unlike a bit, it can exist in a *superposition* of states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1.$$

The coefficients α and β are called *probability amplitudes*. Upon measurement in the computational basis, the state collapses probabilistically to $|0\rangle$ with probability $|\alpha|^2$ or to $|1\rangle$

with probability $|\beta|^2$.

A useful geometric representation of a qubit is the *Bloch sphere* [7] (see Fig. 2.1). Any pure qubit state can be parametrized as:

$$|\psi\rangle = \cos\left(\tfrac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\tfrac{\theta}{2}\right)|1\rangle,$$

where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$ define polar and azimuthal angles.

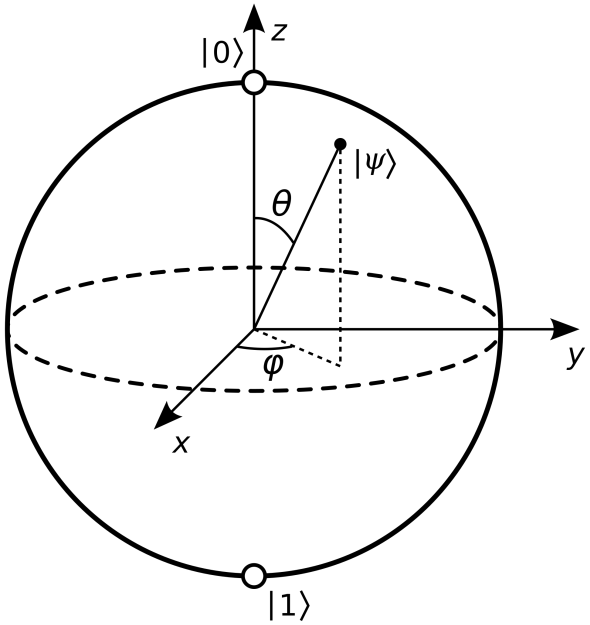


Figure 2.1: Representation of a qubit on the Bloch sphere. Source: Wikimedia Commons [41].

2.1.2 Measurement and No-Cloning

Measurement in quantum mechanics is inherently destructive: once a qubit is measured, it collapses to a classical bit and loses its superposition. This has two key implications for quantum programming:

1. Intermediate measurements must be carefully planned as they irreversibly affect the computation.

2. Quantum data cannot generally be reused once measured.

Another central result is the *no-cloning theorem* [47]: there exists no unitary transformation that can perfectly duplicate an unknown quantum state. This property sharply

distinguishes quantum data from classical information and motivates the linearity constraints in quantum programming languages.

2.1.3 Quantum Registers and Entanglement

For an n-qubit system, the overall state lives in a 2^n-dimensional Hilbert space [29]. Superposition naturally generalizes to registers, allowing a single n-qubit state to encode 2^n classical configurations simultaneously.

A distinctive quantum phenomenon is *entanglement*, in which the joint state of multiple qubits cannot be expressed as a tensor product of individual single-qubit states. For example, consider the Bell state:

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

This state cannot be factored into $|\psi_1\rangle \otimes |\psi_2\rangle$, which means that the qubits do not possess well-defined independent states of their own. Measurement outcomes on one qubit are perfectly correlated with those of the other: if the first qubit is measured as $|0\rangle$, the second is guaranteed to also be $|0\rangle$, and also for $|1\rangle$. Entanglement thus encodes non-classical correlations that have no analogue in classical probability theory, and it is a fundamental resource exploited by quantum algorithms and communication protocols such as teleportation and quantum key distribution.

2.1.4 Quantum Gates and Circuits

Quantum computations are performed by applying *quantum gates*, which are represented by *unitary matrices* acting on qubit states [29]. Since unitary transformations preserve norms, they ensure that probabilities remain valid throughout the computation. Gates are usually categorized into *single-qubit* and *multi-qubit* operations.

Single-Qubit Gates

Single-qubit gates act on states of the form $\alpha\,|0\rangle + \beta\,|1\rangle$ and can create superpositions, introduce phase shifts, or flip states. Common examples include:

- **Pauli Gates** (X, Y, Z): These gates correspond to rotations by π around the x, y, and z axes of the Bloch sphere.

– *Pauli-X (bit flip)*:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle.$$

– *Pauli-Z (phase flip)*:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle.$$

– *Pauli-Y (bit and phase flip)*:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad Y|0\rangle = i|1\rangle, \quad Y|1\rangle = -i|0\rangle.$$

- **Hadamard Gate** (H): Creates uniform superpositions by mapping basis states as:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad H|0\rangle = \tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle, \quad H|1\rangle = \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle.$$

  Thus, applying H to $|0\rangle$ produces $|+\rangle$, an equal superposition of $|0\rangle$ and $|1\rangle$.

- **Phase Gates** (S and T): These gates apply controlled phase shifts without altering the amplitude of $|0\rangle$.

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad S|1\rangle = i|1\rangle,$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}, \qquad T|1\rangle = e^{i\pi/4}|1\rangle.$$

  The T gate, also called the $\pi/8$ gate, is crucial for achieving universal quantum computation when combined with Clifford gates.

Multi-Qubit Gates

Multi-qubit gates operate on two or more qubits, introducing entanglement or conditional behavior:

- **Controlled-NOT (CNot):** Flips the target qubit if the control qubit is $|1\rangle$:

$$\text{CNot} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

  so that $|10\rangle \mapsto |11\rangle$ and $|11\rangle \mapsto |10\rangle$, while states with control $|0\rangle$ remain unchanged.

- **Toffoli Gate (CCNot):** A controlled-controlled-NOT gate: the target qubit flips only if both control qubits are $|1\rangle$. This gate is universal for classical reversible logic.

- **Controlled Phase Gates** (CZ, CR_k): These gates introduce conditional phase shifts. For example, the controlled-Z gate applies a Z gate to the target if the control is $|1\rangle$:

$$CZ\,|11\rangle = -\,|11\rangle\,, \quad CZ\,|10\rangle = |10\rangle\,.$$

  Such gates are essential for creating entanglement.

2.1.5 Universality and Gate Decomposition

Quantum circuits are built by composing quantum gates, which correspond to unitary transformations. Composition operates in two ways:

- **Sequential composition** corresponds to matrix multiplication: applying gate U followed by V results in the unitary VU.

- **Parallel composition** corresponds to the tensor product: applying U to one qubit and V to another yields the combined transformation $U \otimes V$.

A finite set of gates is said to be *universal* if any n-qubit unitary operation can be approximated to arbitrary precision using only gates from that set. One widely used universal set is:

$$\{H, T, \text{CNot}\},$$

where:

- H creates superpositions,

- T introduces a non-Clifford phase shift,
- CNot entangles qubits.

In practice, quantum algorithms are first expressed in terms of high-level operations (e.g., the Quantum Fourier Transform [29]) but must eventually be *decomposed* into gates supported by the target hardware. For example, many fault-tolerant architectures natively support only the so-called *Clifford+T* gate set. A Toffoli gate, which acts on three qubits, is not native in this set but can be decomposed into a sequence of Clifford and T gates:

$$\text{Toffoli} = \ldots \quad (7\ T \text{ gates} + 8 \text{ CNots} + 6\ H \text{ gates})$$

This decomposition is not unique: different compiler strategies can produce circuits with lower T-depth, fewer qubits, or reduced overall gate counts.

The choice of decomposition has a direct impact on *resource metrics*:

- **Circuit depth:** the number of sequential layers of gates.
- **Gate count:** especially T-count, which dominates cost in fault-tolerant settings.
- **Qubit usage:** some decompositions introduce ancillary qubits to reduce depth.

Thus, understanding universality and decomposition is essential not only for compilation but also for *resource analysis*, as different decompositions can lead to vastly different performance and hardware requirements.

2.2 The Quantum Circuit Model

Quantum circuits provide both a visual and an algebraic framework for modeling quantum computations. In this model, a computation is represented by a set of *wires*, one for each bit or qubit, that runs from left to right, where the horizontal position corresponds to the temporal order of operations. Quantum gates are represented as boxes placed on the wires: single-qubit gates act on a single wire, while multi-qubit gates (such as controlled-NOT) connect multiple wires. Measurement operations are typically depicted as meter symbols and collapse the quantum state onto a classical outcome. As a first example, consider the following circuit that implements the quantum teleportation protocol [5] :

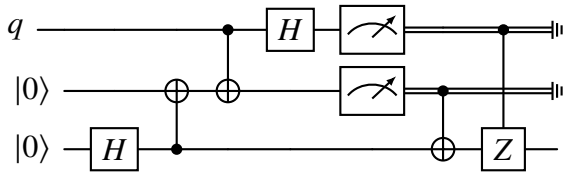


Figure 2.2: The Quantum teleportation circuit. The value of the first wire $|q\rangle$ is teleport to the last wire.

Unlike classical circuits, quantum circuits are inherently probabilistic due to the role of measurements. The evolution of the quantum state up to measurement is unitary and reversible, but the act of measuring introduces randomness, returning classical bits sampled from the underlying quantum state. Importantly, this allows quantum circuits to combine quantum and classical information flow in a single computational model.

It is also important to distinguish between *quantum algorithm* and *quantum circuit*. A quantum circuit represents a fixed, concrete computation for a given input size, while a quantum algorithm describes a *family of circuits* parameterized by the size of the input or other problem-specific parameters. For example, Grover's search algorithm can be seen as a collection of circuits whose depth and gate count scale with the size of the search space. This distinction becomes particularly relevant when analyzing resource requirements, since algorithms often produce symbolic expressions for qubit counts, gate counts, and depth as functions of problem size, while concrete circuits give numerical resource counts for fixed inputs.

Quantum circuits often incorporate elements of classical control flow. After certain measurement operations, the resulting classical bits can influence which subsequent gates are applied. This hybrid model—where classical and quantum instructions are interleaved—is essential for a number of practical protocols and algorithms. Examples include:

- **Quantum Teleportation:** measurement outcomes determine classically controlled Pauli corrections.
- **Quantum Error Correction:** errors on qubits are detected via syndrome measurements, which indicate the type of error and guide the appropriate recovery operations to restore the quantum state.
- **Variational Quantum Algorithms (VQAs):** classical optimizers iteratively update the parameters of quantum subroutines [30].

2.3 Quantum Programming Languages

Quantum programming languages provide high-level abstractions for specifying quantum computations, effectively bridging the gap between theoretical models and executable circuits on real hardware. Due to significant differences in their design principles, abstraction mechanisms, and underlying programming paradigms, these languages can be systematically classified into distinct categories.

High-level, functional, or circuit-description languages These languages emphasize symbolic circuit construction rather than runtime qubit manipulation. Programs describe entire families of circuits parametrically, which is especially useful for static analysis and resource estimation. Examples include `Quipper` [21] and *Proto-Quipper* [36]: they embed a functional programming paradigm into quantum computation, enabling recursion, higher-order functions, and parametric circuit definitions.

Low-level, hardware-agnostic languages These languages act as intermediate representations, focusing on explicit gate sequences rather than high-level abstractions.
`OpenQASM 3.0` [14] is a prime example: it provides a textual format with qubit and classical register declarations, sequential gate application, and explicit classical control flow. Such languages serve as the common denominator for compilation, transpilation, and execution on hardware backends.

Imperative or hardware-oriented frameworks Languages in this category, such as `Qiskit` [32], embed an imperative programming model within a host language (e.g., `Python` [45]). They allow developers to build circuits programmatically, execute or simulate them, and leverage backend-specific optimizations. They are particularly convenient for prototyping and experimenting, but generally do not support symbolic parametric circuit generation for static analysis.

2.4 Resource Analysis of Quantum Circuit Families

Resource analysis provides a framework for quantifying the cost of quantum computations. Since current and near-term quantum devices are resource-constrained, efficient use of qubits and gates is critical. The most common metrics include:

- **Width:** The number of qubits (wires) required by the circuit. This corresponds to the memory footprint of the quantum computation.
- **Depth:** The length of the longest path of dependent gates, that is, the minimum number of time steps required under parallel execution. Variants such as T-depth are also used, focusing on costly non-Clifford gates.
- **Gate Count:** The total number of gates in the circuit. Particular emphasis is placed on the T-count, as T gates are resource-intensive in fault-tolerant computation.

Resource metrics can be categorized into two types:

- **Local metrics:** These metrics are defined with respect to individual wires. They describe properties such as the depth of a specific wire, which is determined by the number of operation that the wire itself depends on.
- **Global metrics:** These metrics quantify the properties of the circuit as a whole. They aggregate information across all wires and gates, for example, the overall gate count, the total width (number of qubits or wires used), or the global circuit depth.

The distinction between local and global metrics is essential: local metrics enable reasoning about the placement and ordering of operations along specific wires, whereas global metrics provide a comprehensive assessment of the circuit's structural complexity. Together, these metrics support both the theoretical analysis and the practical optimization of quantum circuits.

Example (Quantum Teleportation). To illustrate the resource analysis of a circuit, consider again the quantum teleportation protocol from Figure 2.2. Given the circuit representation of the program, the teleportation protocol can be quickly analyzed: it uses three qubits, two Hadamard gates, two two-qubit gates (CNots), two measurements, two classically controlled gates (a NOT and a Pauli-Z) and has a depth of six. Although the maximum number of subsequent operations that act on a wire is four, some of them are applied between wires at different depth. If that is the case, both outputs will be at the depth of the deeper wire plus the depth of the gate, usually one. For example, the last operation, the CZ gate, can only be applied after the last CNot; therefore, the wire q increases its depth by 2 when applying the controlled Z gate.

2.4.1 The Resource Estimation Problem

The current state of quantum hardware, often referred to as the NISQ (Noisy Intermediate-Scale Quantum) era, means that devices struggle with high error rates, short coherence times, and a limited number of logical qubits. These constraints make accurate resource estimation critical for determining the feasibility and practical requirements of quantum algorithms on current and near-future hardware.

Traditional approaches to resource estimation, such as manual calculations or classical simulations, are often slow, error-prone and do not scale well for complex quantum programs. Furthermore, they typically analyze circuits *after* they have been generated (recall the previous example from Figure 2.2), meaning they inspect a concrete instance rather than providing parametric guarantees that hold for an entire family of circuits. This makes it difficult to assess how resource requirements scale with the size of the problem.

This challenge highlights the need for static analysis techniques, particularly type inference, to verify resource requirements at the program level. By integrating resource analysis directly into the type system of a programming language, it becomes possible to infer parametric upper bounds on circuit size measures (width, depth, gate count) directly from the program's source code, ensuring that these bounds are mathematically guaranteed correct. This shifts the focus from costly post-generation testing and optimization to *correct by design* program development.

2.5 The Lambda Calculus

The *lambda calculus* is a formal system for defining and applying functions [31]. It forms the theoretical foundation of functional programming languages, including Haskell [13], and is therefore central to understanding the semantics of the code and analyses presented in this work.

2.5.1 Syntax and Semantics

The syntax of the untyped lambda calculus consists of:

- **Variables** $x, y, z, \ldots$
- **Abstractions** $\lambda x.\, t$ (functions with parameter x and body t)

- **Applications** $t\ u$ (applying a function t to an argument u)

For example, the term:

$$(\lambda x.\, x + 1)\ 2$$

represents a function that increments its argument by 1, applied to the value 2.

2.5.2 Free occurrences and free variables

An occurrence of a variable x in a term t is said to be *bound* if it lies within the scope of a λx abstraction; otherwise, it is *free*. For example, in the term $\lambda x.\ (x\ y)$, the first occurrence of x is bound, whereas the occurrence of y is free.

The set of free variables of a term t, written $\mathrm{FV}(t)$, is defined inductively as follows:

$$\mathrm{FV}(x) = \{x\}$$

$$\mathrm{FV}(\lambda x.\, t) = \mathrm{FV}(t) \setminus \{x\}$$

$$\mathrm{FV}(t_1\ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$$

A term is called *closed* if it has no free variables, i.e., if $\mathrm{FV}(t) = \emptyset$.

2.5.3 Substitution

A key operation in lambda calculus is *substitution*, denoted $t[x/u]$, meaning "the term obtained by replacing all free occurrences of x in t with u".

Formally:

$$(x)[x/u] = u$$

$$(y)[x/u] = y \quad \text{if } y \neq x$$

$$(\lambda y.\, t)[x/u] = \begin{cases} \lambda y.\, t & \text{if } y = x \\ \lambda y.\, t[x/u] & \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(u) \end{cases}$$

Care must be taken to avoid *variable capture*, typically resolved by renaming bound variables as needed.

2.5.4 Operational Semantics and Inference Rules

Reduction rules describe how terms are evaluated. The central rule is *beta-reduction*:

$$(\lambda x.\, t)\; u \longrightarrow t[x/u]$$

This models the application of a function to an argument by substituting the argument into the body.

We distinguish between:

- **Small-step semantics:** evaluation proceeds by repeatedly applying a single reduction rule until no further rules apply.

- **Big-step semantics:** evaluation directly relates a term to its final result.

Example: Addition For example, suppose we extend lambda calculus with integer literals and an addition operator +. We might define an inference rule for addition in big-step semantics as:

$$\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 + t_2 \Downarrow n_1 + n_2}$$

where $t \Downarrow n$ denotes that term t evaluates to the integer n. The evaluation of $t_1 + t_2$ corresponds to evaluating the single terms and obtaining two integers, and the result of the expression is given by the sum of these two integers.

These inference rules will be particularly relevant in later chapters when we define operational semantics for resource-aware quantum languages.

2.6 Haskell

`Haskell` is a purely functional programming language [13] widely used in research for implementing interpreters, type systems, and domain-specific languages. Its strong type system, concise syntax, and emphasis on immutability make it well-suited for modeling and reasoning about quantum programming languages.

2.6.1 Functional Paradigm

In `Haskell`, functions are first-class citizens: they can be passed as arguments, returned from other functions, and composed. Unlike imperative languages, `Haskell` avoids mutable state and side effects by default, which aligns well with the mathematical nature of lambda calculus.

2.6.2 Folds

A *fold* is a higher-order function that processes data structures recursively and accumulates a result. For lists, there are two main folds:

Definition 2.1 (Right fold)**.** Let $f : A \times B \to B$ be a binary function and $z \in B$. For a list $[x_1, x_2, \ldots, x_n] \in A^n$, the right fold is defined as:

$$\texttt{foldr}\ f\ z\ [\,] = z, \quad \texttt{foldr}\ f\ z\ [x_1, x_2, \ldots, x_n] = f(x_1, f(x_2, \ldots f(x_n, z)\ldots)).$$

Equivalently, recursively:

$$\texttt{foldr}\ f\ z\ (x : xs) = f(x, \texttt{foldr}\ f\ z\ xs).$$

where the notation $x : xs$ denotes a list with head x and tail xs. For example, $[x_1, x_2, \ldots, x_n]$ could be written as $x_1 : [x_2, \ldots, x_n]$.

Definition 2.2 (Left fold)**.** Let $f : B \times A \to B$ be a binary function and $z \in B$. For a list $[x_1, x_2, \ldots, x_n] \in A^n$, the left fold is defined as:

$$\texttt{foldl}\ f\ z\ [\,] = z, \quad \texttt{foldl}\ f\ z\ [x_1, x_2, \ldots, x_n] = f(\ldots f(f(z, x_1), x_2)\ldots, x_n).$$

Equivalently, recursively:

$$\texttt{foldl}\ f\ z\ (x : xs) = \texttt{foldl}\ f\ (f(z, x))\ xs.$$

Folds are fundamental in `Haskell` because they encapsulate recursion patterns over lists, simplify code, and allow reasoning about program behavior in a compositional way. They will also become relevant when analyzing circuit generation in later chapters, in which right folds are used to traverse and combine structures.

Part I

Evaluation and Validation of PQ Programs

In this part, we develop the foundations for the execution and analysis of PQ programs. We begin by extending the theoretical framework of *Proto-Quipper-RA* with an evaluation process that produces *configurations*, each consisting of a program expression and its corresponding circuit. This allows us to reduce the expression of PQ programs with a big-step semantics while simultaneously constructing its circuit representation.

Once the evaluation step has constructed the circuit, we demonstrate how its metrics can be computed, such as gate counts, depth, and width. These concrete metrics, are compared against the static upper bounds inferred by QuRA on the circuit-generating functions. Although soundness guarantees that the inferred bounds are never violated, our analysis investigates how *tight* these bounds are in practice. This provides insight into both the correctness and the practical precision of QuRA's resource analysis.

Through a series of case studies, from small, non-parametric circuits such as teleportation to scalable parametric families like the quantum Fourier transform and Grover's algorithm, we validate the evaluation mechanism and quantify the accuracy of resource estimates.

Chapter 3

QuRA: A Tool for Resource Analysis of Quantum Programs

Quantum programming languages are essential for expressing algorithms that operate on qubits, but reasoning about their resource requirements, such as qubit usage, gate counts, or circuit depth, is often challenging. This chapter introduces QuRA [9] and the underlying theoretical environment, a state-of-the-art framework designed to facilitate the analysis of quantum programs with a focus on resource estimation and correctness.

To better understand QuRA's foundations, we begin by reviewing the Quipper language [21], highlighting its capabilities to describe quantum circuits and how it serves as the foundation for *Proto-Quipper* [36, 12, 11] languages. We then introduce the *Proto-Quipper* family of languages and a concrete circuit model, which provides the basis for formal reasoning about programs and their resource consumption.

Building on this concepts, we present *Proto-Quipper-RA* [9], a theoreticla programming language for resource-aware quantum programming. We describe its syntax, tpe-and-effect system, and operational semantics, which allow precise modeling of program behavior and enable tracking of global and local resource metrics throughout circuit construction.

The second part of the chapter focuses on QuRA [9] itself. We describe the role and functionalities of the tool, emphasizing its key differences with *Proto-Quipper-RA*, including the way it integrates type checking, symbolic indices, and resource analysis into a unified workflow. We then detail the inference algorithm that underpins resource estimation and illustrate its use with a concrete example, showing how QuRA can automatically compute metrics such as qubit counts, gate counts, and other circuit characteristics for a given quantum

program.

3.1 The Quipper Language

Quipper [21] is a functional programming language designed for quantum computing that serves as a *circuit description language*. This means that it can be used to construct quantum circuits in a structured manner, applying gates one at a time. A distinguishing feature is its ability to treat completed circuits as data, allowing them to be stored in variables, passed to subroutines, and subjected to meta-operations like transformations, gate counts, inversion, and error correction. This mechanism is known as *boxing*: subcircuits can be encapsulated into reusable modules (or "boxes") and subsequently invoked as single gates within larger circuits. This two-level description (gate operations and meta-operations on entire circuits) aligns well with how quantum algorithms are often specified in the literature, providing a useful high-level programming paradigm. Quipper is practical and has been used to implement large-scale quantum algorithms, generating circuits with trillions of gates. It is particularly adept at describing parameterized families of quantum circuits, such as a different circuit for each integer to be factored in Shor's algorithm [38].

Quipper's implementation as an embedded domain-specific language within Haskell [13], allowing it to inherit its advanced features, including a strong type system and higher-order functional programming capabilities, which are valuable for expressing abstract circuit construction. It provides a rich set of primitives for manipulating qubits, gates, and circuits, supporting common quantum gates, controlled operations, ancilla management, and measurement. Higher-level features such as boxed subcircuits (which promote modularity and scalability) and functions to estimate circuit resources (like gate count, qubit usage, and circuit depth) are also included. The language maintains a crucial conceptual distinction between circuit generation time (when classical parameters define the circuit's shape and structure) and circuit execution time (when quantum states are instantiated on a quantum device). This separation allows Quipper to define general circuit families that are parameterized by input size or structure. Quipper programs appear to imperatively manipulate qubits, but they actually describe circuits behind the scenes.

3.2 From Quipper to a *Proto-Quipper* Family of Languages

Despite its strengths, Quipper has several drawbacks. Due to mismatches between its type system and Haskell's, the original Quipper language is not type-safe, meaning that some well-typed programs can lead to run-time errors. A critical issue is Haskell's inability to enforce linear quantum types, which is the fundamental requirement that a quantum state (qubit) cannot be used more than once, consistent with the no-cloning property of quantum information. Furthermore, as an embedded language, Quipper lacks formal semantics and dependent types [31], which could improve correctness guarantees by allowing types to depend on values, such as encoding circuit sizes directly into the type system.

These shortcomings motivated the development of *Proto-Quipper*, a family of research-oriented, formal calculi that serve as stand-alone programming languages with their own custom type systems and semantics. Each *Proto-Quipper* variant is designed to address specific aspects and problems of the original Quipper language, such as type safety, linearity, and formal semantics.

Essentially, the *Proto-Quipper* languages amount to effectful linear lambda calculi specifically crafted for constructing and handling circuits [36]. Some of them introduce a categorical model for circuits and programs, such as *Proto-Quipper-M* [34], *Proto-Quipper-D* [20], and *Proto-Quipper-L* [20]. Some other languages, among them *Proto-Quipper-RA* [10], instead introduce a concrete categorical model of parameters and states and use label contexts and wire bundles to define the *Circuit Representation Language* (CRL).

3.2.1 Circuit Representation Language

The CRL allows for a low-level description of circuits, which are represented by a chain of operations, applied to a bundle of named wires, which are called labels. Each wire is designated by a label ℓ, and collections of multiple qubits are denoted as $\bar{k}$. Wire bundles serve to determine the input and output parameters in the quantum operations that characterize the circuit. The empty circuit is represented with the *identity* notion, written as id_Q. Together with the circuits, a *label context* is paired, a mapping that links wire names to their respective types, which may be Qubit or Bit, to facilitate identification of circuit components. An empty label context is denoted by the symbol $\bullet$.

The CRL grammar is described formally as follows:

$$CRL \quad C, \mathcal{D} ::= id_Q \mid C; g(\bar{\ell}) \to \bar{k}, \tag{3.1}$$

where Q is the label context and g represents a quantum operation, such as qubit initializations, single-qubit gates, and multiple-qubit gates, coming from a fixed set $\mathscr{G}$.

CRL expressions serve as the target model for circuit building in *Proto-Quipper*. For instance, the teleportation circuit [5] can be represented in CRL as:

$$id_{q:\mathsf{Qubit}}; \mathsf{qinit}_0(*) \to d; \mathsf{qinit}_0(*) \to a; \mathsf{hadamard}(d) \to d' \tag{3.2}$$

$$\mathsf{CNot}(\langle d', a\rangle) \to \langle d'', a'\rangle; \mathsf{CNot}(\langle q, a'\rangle) \to \langle q', a''\rangle; \tag{3.3}$$

$$\mathsf{hadamard}(q') \to q''; \mathsf{measure}(a'') \to x; \mathsf{measure}(q'') \to y; \tag{3.4}$$

$$\mathsf{cCNot}(\langle x, d''\rangle) \to \langle x', d'''\rangle; \mathsf{ccz}(\langle y, d'''\rangle) \to \langle y', d''''\rangle; \tag{3.5}$$

$$\mathsf{cdiscard}(x) \to *; \mathsf{cdiscard}(y) \to *. \tag{3.6}$$

where $\mathsf{qinit}_0, \mathsf{hadamard}, \mathsf{CNot}, \mathsf{measure}, \mathsf{cCNot}, \mathsf{ccz}, \mathsf{cdiscard} \in \mathscr{G}$.

Circuits described in this way present a structure composed of sequential operations, and therefore it is useful to be able to describe the *concatenation* of them as follows:

Definition 3.1 (concatenation)**.**

$$C :: id_Q = C, \tag{3.7}$$

$$C :: (\mathcal{D}; g(\bar{\ell}) \to \bar{k}) = (C :: \mathcal{D}); g(\bar{\ell}) \to \bar{k}. \tag{3.8}$$

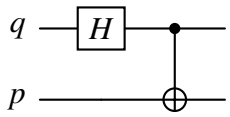
A key aspect of these languages is their effectful nature, which captures the fact that evaluating a program has a direct impact on the underlying circuit structure. Instead of producing only a pure value, the programs extend the current circuit by appending new gates and operations as a side effect. We can notice in the previous example that a label does not exist until a gate produces it as an output and extends the label context; circuit-building effects are explicitly tracked and encapsulated by the type system.

For example, consider the Bell circuit from earlier, shown again in Figure 3.1, which applies a Hadamard gate to the first wire and then performs a Controlled-NOT using the other wire as the control. Although the function can be written in a purely functional style,

evaluating it implicitly modifies the global circuit by inserting these two gates in sequence. The program's return value consists of updated *references* to the affected qubits in the form of labels, while the accumulated gates are recorded in the circuit representation. This tight integration between program evaluation and circuit generation plays a central role in analyzing quantum programs *symbolically*.

```
bell :: Qubit ->
        Qubit ->
        Circ (Qubit, Qubit)
bell q p = do
    q <- hadamard q
    p <- qnot p `controlled` q
    return (q, p)
```

(a) `Quipper` code of the Bell circuit



$$id_\bullet;\ \mathsf{qinit}_0(*) \to q; \tag{3.9}$$
$$\mathsf{qinit}_0(*) \to p; \tag{3.10}$$
$$\mathsf{hadamard}(q) \to q' \tag{3.11}$$
$$\mathsf{CNot}(\langle q', p\rangle) \to \langle q'', p'\rangle. \tag{3.12}$$

(b) Bell circuit representation and its CRL notation

Figure 3.1: Bell state preparation in `Quipper`: the code on the left, the circuit representation on the top right, and its CRL notation below.

3.3 *Proto-Quipper-RA*: a Framework for Resource Analysis

Proto-Quipper-RA [10] (PQRA for short) is an extension of the *Proto-Quipper* family, a type-safe, functional quantum programming language specifically designed to support flexible and compositional resource estimation of quantum programs. It addresses the need to accurately track resource consumption, such as gate count, qubit usage (width), and circuit depth, which are critical to determining the feasibility of quantum algorithms on current hardware.

The power of *Proto-Quipper-RA* lies in its type system, which integrates refinement types, effects, and closures into a unified framework for reasoning about quantum resources. Resource bounds are embedded directly into types using *index terms* (I, J), arithmetic expressions over natural numbers, and symbolic parameters. Depending on the construct, these annotations represent upper bounds on different aspects of the circuit.

- **Refinement types:** These types enrich the type system by attaching quantitative information directly to types, enabling precise specification of constraints on quantum resources, such as the number of operations carried by a wire or the expected size of

a subcircuit. The associated indices can be parametric, depending on classical input parameters, which allows for accurate and scalable analysis across entire families of circuits.

- **Effect typing:** This enhancement of the type system enables tracking the global resource consumption of functions that generate circuits. Each function is annotated with symbolic expressions that summarize its side effects—such as the number of gates introduced or the amount of qubits allocated, which are combined according to the program's structure to approximate the overall cost of the circuit.

- **Closure types:** To maintain soundness under abstraction, closure types track the resources captured by higher-order functions that generate subcircuits. They ensure that the size of these captured wires is accurately represented in the analysis, allowing safe handling of such functions.

3.3.1 The Syntax of *Proto-Quipper-RA*

Figure 3.2 illustrates the formal grammar that defines the core components of *Proto-Quipper-RA*.

Types	*TYPE*	A, B	$::= \mathbb{1} \mid w^I \mid !^I A \mid A \otimes B \mid A \multimap_T^I B$
			$\mid \mathsf{List}_{i<I}\, A \mid \mathsf{Circ}_\Theta^I(T, U)$
Param. types	*PTYPE*	P, R	$::= \mathbb{1} \mid !^I A \mid P \otimes R \mid \mathsf{List}_{i<I}\, P \mid \mathsf{Circ}_\Theta^I(T, U)$
Bundle types	*BTYPE*	T, U	$::= \mathbb{1} \mid w^I \mid T \otimes U \mid \mathsf{List}_{i<I}\, T$
Terms	*TERM*	M, N	$::= V\, W \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\, V$
			$\mid \mathsf{box}_{\Theta,T}\, V \mid \mathsf{apply}(V, W) \mid \mathsf{return}\, V$
			$\mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \mathsf{fold}_i\ V\ W\ X$
Values	*VAL*	V, W	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\bar{\ell}, C, \bar{k})$
			$\mid \langle V, W\rangle \mid \mathsf{nil} \mid \mathsf{rcons}\ V\ W$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{rcons}\ \bar{\ell}\ \bar{k}$
Indices	*INDEX*	I, J	$::= n \mid i \mid I + J \mid I - J \mid I \cdot J \mid \mathsf{max}\,(I, J)$
			$\mid \mathsf{size}_W \mid \mathsf{append}_g(I, J, E, F) \mid \mathsf{out}_{g,n}(I_1, \ldots, I_m)$
			$\mid \mathsf{e} \mid \mathsf{wire}_w \mid I \circledast J \mid I \oplus J \mid \circledast_{i<I} J \mid \bigoplus_{i<I} J$

Figure 3.2: The syntax and types of *Proto-Quipper-RA*.

Types

Types in *Proto-Quipper-RA* classify both data and computations, incorporating resource information directly into their structure. The **unit type** ($\mathbb{1}$) represents a trivial value, while **wire types** (w^I), such as `Qubit`, or `Bit`, represent individual wires in the circuit of size I; for example, $\texttt{Qubit}^0$ could represent a qubit at depth zero, if depth is the tracked metric. The **bang type** $!^I A$ denotes a suspended computation of type A, with the index I tracking the resources required by the computation. Linear combinations of types are expressed using the **tensor type** ($A \otimes B$), whereas the **arrow type** ($A \multimap^I_T B$) is annotated with two indices to capture both effect I and closure information T. The arrow type is a function from A to B that produces a circuit of size at most I: this allows for a static verification of the resource needed by the function. The **list type** ($\text{List}_{i<I} A$) is refined with an annotation indicating that the list contains exactly I elements, where the type of the i-th element may depend on its position i. Finally, the **circuit type** ($\text{Circ}^I_\Theta(T, U)$) is quite useful, as it represents boxed circuits in which I provides an upper bound on the circuit size, and Θ is a set of index variables encoding local metrics within the input (T) and output (U) types of the circuit.

To give some context, if we were to track the width resource metric, a function whose type signature is Qubit $\multimap^3_0$ Qubit takes as input a qubit and returns a qubit in output, while producing a circuit of width three.

Parameter Types

Parameter Types are a subset of Types specifically denoting classical parameters. Values of these types can be freely copied or ignored, which is a key distinction from linear quantum types, adhering to the linear-nonlinear typing discipline of *Proto-Quipper-RA*. This category includes **unit types**, **bang types**, **tensor types** of parameters, **dependent lists** of parameters, and boxed **circuit types**, already detailed in the previous paragraph.

Bundle Types

Bundle types are another subset of Types, used to denote collections of wires. They are useful for inferring the input and output type of quantum operations and boxed circuits, and for formalizing the structure of the wire bundles.

Terms

Terms are the basic building blocks of the program expression and represent most of the elements of the language, as in the usual lambda calculus. Terms can be evaluated by applying certain evaluation rules and reduced to other terms or wire bundles, to represent that a quantum operation occurred and produced a certain set of labels, or other side effects.

Terms vary a lot in their syntax, allowing for many different operations. **function application** $(V\ W)$ applies the term W to V, where the term W is usually an **abstraction** (see 3.3.1). The **let** expression (let $x = M$ in N) binds one or more variables x to an expression M of compatible shape and substitutes it within the term N. The force V operator executes a suspended computation of bang type $!^I A$, yielding the results and circuit it generates. The **boxing** construct $(\text{box}_{\Theta,T} V)$ is a powerful element of the language, enabling the creation of boxed circuits that can be freely duplicated and reused, just like regular data; similarly to the `circ` type, Θ tracks index variables for local metrics, and T is the input type of the circuit. $\text{apply}(V, W)$ is used to apply a boxed circuit V to a wire bundle W, effectively appending the results of the application of the boxed circuit to the underlying circuit. **return** is used to create a trivial computation from a value V that does not explicitly modify the underlying circuit.

The last and perhaps most complex term used to provide a simple, yet powerful form of recursion on lists is the **fold** construct. $\text{fold}_i\ V\ W\ X$ is annotated with an iteration variable that enables different behaviors based on the current step of the fold evaluation. The fold syntax is almost identical to the Haskell implementations, in which the step function V is applied to the term X, using W as the accumulator. Note that this operator corresponds to a right-fold, since that PQRA lists grows from left to right, as we will detail in a moment.

Values

Values are defined as terms that cannot be reduced any further, possibly because they are variables/labels or expressions that cannot be reduced by themselves.

Proto-Quipper-RA defines the following terms as irreducible:

- **Unit value** $(*)$: utilized as a undefined value or as an empty wire. It can be used applied to operations that do not need any input or output label, for example to initialize or discard a qubit or a bit.

- **Variables** (x, y): the atomic symbols used to represent placeholders or unevaluated terms within expressions. They are the simplest kind of terms and form the foundation upon which abstractions and applications are built, employing them to substitute and assign terms.
- **Labels** (ℓ, k): used to represent the wires of the underlying circuit.
- **Abstractions** ($\lambda x_A.M$): a function that is waiting for an argument x of type A to perform the evaluation of M. It cannot be reduced because it needs an argument passed through an application.
- **Lifted expressions** ($\mathsf{lift}\, M$): suspends the evaluation of the term, *lifting* it to a value that can be easily passed around the expression, copied, or discarded.
- **Boxed Circuits** ($\bar{\ell}, C, \bar{k}$): explicitly represents a concrete quantum circuit C as data within the language, along with its input ($\bar{\ell}$) and output ($\bar{k}$) interfaces, expressed as wire bundles.
- **Tensor** ($\langle V, W \rangle$): a pair of values V and W.
- **Empty list** (nil).
- **Lists** ($\mathsf{rcons}\ V\ W$): A list V with element W appended to it. Lists grow to the right with the right-cons operator and represent ordered collections of terms.

Wire Bundles

Wire Bundles' grammar is used to reason over structured collections of labels, matching the shape of the input and output types of circuit-building functions. Empty (non existing) wires can be represented using the **unit value** $*$, while single wires are named after their **label**. Wire bundles can be composed of multiple labels, grouped using **tuples** or **lists**, built using the nil and rcons operators. Such constructors can be used to create lists of lists, tuples of empty wires, or more complex structures.

Indices

Indices are arithmetic expressions that play a crucial role in describing resource dependencies and quantifying various properties of quantum circuits within the language's type-and-effect

system. Indices are built from natural **numbers** (n), index **variables** (i, j, e), and natural arithmetic operations such as addition, natural subtraction, multiplication and maximum. A key feature of *Proto-Quipper-RA* is the inclusion of abstract resource operators within the indices. These operators do not have a standard arithmetic interpretation by themselves but act as placeholders that are interpreted differently depending on the specific resource metric being analyzed. These include: the null size e, the size of a single wire of type w wire_w, the size of sequential and parallel circuit composition, $I \circledast J$ and $I \oplus J$, along with their generalization to bounded compositions, $\circledast_{i<I} J$ and $\bigoplus_{i<I} J$.

3.3.2 The Operational Semantics of *Proto-Quipper-RA*

Up to this point, we managed to introduce the potentialities of the language and its grammar, as we should now be able to represent a quantum program using the constructs of *Proto-Quipper-RA*. What we are missing are the inference rules according to which we will be able to reduce the terms and store their effects, building up the CRL expression of the circuit. The symbol $\Downarrow$ represents the evaluation relation of a configuration (C, M) of a circuit and a term, and is used in $(C, M) \Downarrow (\mathcal{D}, V)$ to signify that the term M is evaluated to V and, in doing so, modifies the circuit C to $\mathcal{D}$. Recall the substitution syntax $M[V/x]$ from Section 2.5.3 to represent the capture-avoiding substitution of x with V in M. Note that such substitution is not an operation with side-effects, and the circuit will not be affected by it.

$$\textsc{app}\ \frac{(C, M[V/x]) \Downarrow (\mathcal{D}, W)}{(C, (\lambda x_A.M)\, V) \Downarrow (\mathcal{D}, W)}$$

The APP rule specifies that to evaluate the application of a term V to an abstraction, we substitute the parameter of the abstraction with the argument of the application and then evaluate the resulting term.

$$\textsc{let-pair}\ \frac{(C, M[V/x][W/y]) \Downarrow (\mathcal{D}, X)}{(C, \mathsf{let}\ \langle x, y\rangle = \langle V, W\rangle\ \mathsf{in}\ M) \Downarrow (\mathcal{D}, X)} \qquad \textsc{let}\ \frac{(C, M) \Downarrow (\mathcal{D}, V) \qquad (\mathcal{D}, N[V/x]) \Downarrow (\mathcal{E}, W)}{(C, \mathsf{let}\ x = M\ \mathsf{in}\ N) \Downarrow (\mathcal{E}, W)}$$

The evaluation of a let expression depends on the structure of the variables that are being

defined. If the bound variable is a tuple, its elements are substituted individually and the resulting term is then evaluated. Otherwise, the term (C, M) is evaluated first, producing a new configuration $(\mathcal{D}, N)$; the substitution is then performed, and the resulting term is evaluated. This rule provides the first example of an evaluation that requires multiple "steps": a term is evaluated in the context of an initial circuit, producing an updated circuit-term configuration, which is subsequently used in further evaluations.

$$\textsc{return}\quad (C, \mathsf{return}\ V) \Downarrow (C, V) \qquad \textsc{force}\quad \frac{(C, M) \Downarrow (\mathcal{D}, V)}{(C, \mathsf{force}(\mathsf{lift}\ M)) \Downarrow (\mathcal{D}, V)}$$

The rules for return and force are quite simple. Whenever we encounter a term that has to be returned, we simply obtain the configuration described by the starting circuit and the returned term. Instead, we evaluate a forced lifted term M by only evaluating the term itself in the context of the circuit.

$$\textsc{apply}\quad \frac{(\mathcal{E}, \bar{q}) = \mathit{append}(C, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))}{(C, \mathsf{apply}((\bar{\ell}, \mathcal{D}, \bar{k}), \bar{t})) \Downarrow (\mathcal{E}, \bar{q})}$$

In order to apply a boxed circuit to a wire bundle and evaluate the obtained configuration, the *append* function is used to match the input labels of the boxed operation and the actual labels appearing in the circuit and in the input labels. In particular, the *append* function is defined as follows:

Definition 3.2 (*append*)**.** We define *appending* $(\bar{\ell}, \mathcal{D}, \bar{k})$ *to* C *on* $\bar{t}$, which we write $\mathit{append}(C, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))$, as the pair of a circuit and a wire bundle computed as follows:

1. Find $(\bar{t}, \mathcal{D}', \bar{q}) \cong (\bar{\ell}, \mathcal{D}, \bar{k})$ such that the labels shared by C and $\mathcal{D}'$ are exactly those in $\bar{t}$, where the $\cong$ symbol represents *equivalence*, meaning that two circuits exhibit the same structure and only differ by a renaming of labels.

2. Compute circuit concatenation $\mathcal{E} = C :: \mathcal{D}'$.

3. Finally, return $(\mathcal{E}, \bar{q})$.

The evaluation of a boxed circuit $\mathsf{box}_T\ (\mathsf{lift}\, M)$ produces a circuit $\mathcal{E}$ and its output labels $\bar{k}$ applying the following BOX rule:

$$\text{BOX}\quad \frac{(Q, \bar{\ell}) = \mathit{freshlabels}(\Theta, T) \qquad (id_Q, M) \Downarrow (\mathcal{D}, V) \qquad (\mathcal{D}, V\ \bar{\ell}) \Downarrow (\mathcal{E}, \bar{k})}{(C, \mathsf{box}_{\Theta,T}\ (\mathsf{lift}\, M)) \Downarrow (C, (\bar{\ell}, \mathcal{E}, \bar{k}))}$$

The rule tells us that we first evaluate the term of the boxed circuit using a blank configuration with the identity circuit. With the term obtained, we evaluate its application with a bundle $\bar{\ell}$, generated with the *freshlabels* function. This function is used to create a fresh label context Q and a wire bundle, given the input type and the index variables context of the box. We can treat this whole operation as the independent evaluation of the boxed term, reducing the evaluation of the configuration to the initial circuit and the evaluated circuit $\mathcal{E}$ along with its input and output bundles.

We conclude this exposition with the rules to evaluate folds, namely FOLD-END and FOLD-STEP:

$$\text{FOLD-END}\quad (C, \mathsf{fold}\ V\ W\ \mathsf{nil}) \Downarrow (C, W)$$

$$\text{FOLD-STEP}\quad \frac{(C, M\{0/i\}) \Downarrow (\mathcal{D}, Y) \qquad (\mathcal{D}, Y\ \langle V, W\rangle) \Downarrow (\mathcal{E}, Z) \qquad (\mathcal{E}, \mathsf{fold}_i\ (\mathsf{lift}\, M\{i+1/i\})\ Z\ W') \Downarrow (\mathcal{F}, X)}{(C, \mathsf{fold}_i\ (\mathsf{lift}\, M)\ V\ (\mathsf{rcons}\ W'\ W)) \Downarrow (\mathcal{F}, X)}$$

The first rule is straightforward, telling us that the evaluation of a fold with nil input results is the accumulator term, without further evaluations, defining the base case of the fold evaluation. The step rule to evaluate a lifted expression M on a list $\mathsf{rcons}\ W'\ W$, using V as accumulator, can be described as follows:

1. Instantiate i with 0, through the capture-avoiding index substitution $M\{0/i\}$, in M and evaluate it. The resulting step function Y is specialized for the first iteration.

2. Apply Y to the tuple $\langle V, W\rangle$ and evaluate the term to obtain the new accumulator Z.

3. Recur the fold by incrementing the index i by 1, then evaluate the term to obtain the final configuration.

3.3.3 Soundness of *Proto-Quipper-RA*

The type-and-effect system underlying *Proto-Quipper-RA* is proven to be sound [10] in the sense that, under reasonable assumptions about how abstract resource indices are interpreted, the inferred resource bounds (on width, depth, gate count, etc.) never underestimate the true consumption of well-typed programs.

The soundness proof hinges on *coherence* constraints, which are mild algebraic conditions placed on how index terms and resource metrics behave. These include, for example:

- The index assigned to the "empty" circuit ("e") must over-approximate its actual size.
- The parallel composition operator ($\oplus$) must be associative and commutative (since the relative ordering of wires does not influence size).
- The parallel composition operator must satisfy a *lax distributivity* property with respect to sequential composition operator ($\circledast$): loosely, size estimates become more accurate when parallel circuit composition distributes over sequential composition.

Because these coherence rules are quite reasonable, the system can support a variety of resource metrics (width, depth, gate count, or restricted versions thereof) while placing minimal burden on the user to satisfy complicated invariants.

In concrete terms, if a program M is assigned a resource bound I by *Proto-Quipper-RA*, then for any circuit C compatible with M, we have that $(C, M) \Downarrow (C :: D, V)$ and that the actual size of the generated circuit D is guaranteed to be at most I. This result establishes that the symbolic, compile-time resource bounds enforce an upper bound on the real, runtime consumption of circuits.

This soundness guarantee justifies the use of `QuRA` (based on *Proto-Quipper-RA*) for static resource estimation: one can trust that the inferred bounds are valid. At the same time, because the required coherence conditions are weak, the system remains expressive enough to capture varied metrics, without imposing heavy constraints on program structure.

3.4 The QuRA Tool

We now transition from the theoretical framework of *Proto-Quipper-RA* to its practical realization, introducing **`QuRA` (`Quipper` Resource Analysis)**, an open-source static analysis

tool. QuRA is implemented in Haskell and its source code is publicly available[1] .

3.4.1 Role and Functionality of QuRA

QuRA's primary role is to automate the estimation and verification of resource consumption for quantum programs described in Quipper-like languages. It takes programs written in PQ, a concrete variant of *Proto-Quipper-RA*, and outputs their inferred types along with parametric upper bounds on the size of the circuits they construct.

The tool is capable of analyzing a diverse set of resource metrics, encompassing both global circuit metrics (e.g., gate count and width) and local wire metrics (e.g., depth). This comprehensive analysis allows QuRA to automatically derive concrete, parametric upper bounds for real-world quantum algorithms, such as the quantum teleportation algorithm, the quantum Fourier transform (QFT), and Grover's algorithm [23], demonstrating its applicability in practical scenarios.

3.4.2 Key Differences and Architectural Changes from *Proto-Quipper-RA*

The shift from a minimalistic theoretical calculus such as *Proto-Quipper-RA* to an usable and practical language, PQ, necessitates several significant architectural and linguistic transformations, which are crucial to understanding QuRA's design and operation.

In *Proto-Quipper-RA*, indices (representing classical parameters) are treated as global, undeclared parameters, with their instantiation being a meta-theoretical property [10]. This approach, while convenient for formal proofs, is impractical for programming. To enhance usability and clarity, PQ introduces explicit index abstraction ($\Lambda I.M$) and application ($M\ @\ I$) constructs directly into the language. This change clarifies the scope of index variables and makes writing functions more intuitive. Consequently, operators such as `fold` and `box` are relieved of the responsibility of explicit index binding, and the `Circ` type no longer requires an explicit index context (Θ) annotation.

The theoretical *Proto-Quipper-RA* calculus maintains a syntactic distinction between terms and values. PQ, however, merges these categories by treating all expressions as effectful terms as in any regular language. Furthermore, when a constructor expecting values is applied to terms, QuRA interprets this as the sequential evaluation of the subterms, followed by the application of the constructor to their resulting values.

[1] Available on GitHub at andreacolledan/qura.

Proto-Quipper-RA relies on the explicit notion of boxed circuits of the form $(\bar{\ell}, C, \bar{k})$ to allow circuit objects (C) to enter the higher-level language. While useful for defining operational semantics, these explicit circuit objects are not typically written by programmers. QuRA takes a more abstract approach by completely removing the underlying circuit model from the surface language. Instead, basic circuit operations (e.g., the Hadamard gate) are introduced as atomic primitives (e.g., a `Hadamard` constant) and are given appropriate circuit types. This abstraction simplifies the language's surface syntax and provides flexibility, as it defers the commitment to a specific circuit implementation. These QuRA primitives are called *constants* and are shown in Figure 3.3.

Quantum Op.	*QOP*	op	::= $\mathsf{QInit}_{\{0,1\}}$ | $\mathsf{QDiscard}_I$ | Meas_I | $\mathsf{CInit}_{\{0,1\}}$ | $\mathsf{CDiscard}_I$ | H_I | X_I | Y_I | Z_I | T_I | $\mathsf{R}_I\, n$ | $\mathsf{Rinv}_I\, n$ | $\mathsf{CNot}_{I,J}$ | $\mathsf{CZ}_{I,J}$ | $\mathsf{CR}_{I,J}\, n$ | $\mathsf{CRinv}_{I,J}\, n$ | $\mathsf{CCNot}_{I,J}$ | $\mathsf{CCZ}_{I,J}$ | $\mathsf{Toffoli}_{I,J,E}$
Constants	*CONST*	c	::= Boxed op | $\mathsf{MakeRGate}_{n,I}$ | $\mathsf{MakeRinvGate}_{n,I}$ | $\mathsf{MakeCRGate}_{n,I,J}$ | $\mathsf{MakeCRinvGate}_{n,I,J}$ | MakeUnitList n

Figure 3.3: The QuRA constants.

When typing constants and quantum operations, indexes such as I and J are used to describe information about the indexes and parameters such as H_3 may denote that the Hadamard gate is applied on a wire at depth 3. The natural numbers n are used as arguments for parametric constants. As an example, $\mathsf{MakeRGate}_{3,0}$ would create the quantum operation $\mathsf{R}_0\ 3$: a gate that performs rotations of $\pi/2^{3-1}$ degrees around the Z axis on a wire at depth 0.

Although cumbersome at first, constants are really important as they allow us to generate the correct quantum operations according to specific indexes, which can depend on the iteration number or the size of the registers. The reader should also notice that the parameters on constants are indexes, but whenever they represent a parameter of the gate, as in the rotation gate R, and not their size, they are not indexes anymore, but simple integer parameters.

In the Table 3.1, the syntactic translations between PQ and *Proto-Quipper-RA* are shown. A new type introduced during the construction of the PQ language, is the *dependent function type*. They represent a function from a natural number i to a type A which can depend on

i. Similar to the arrow type, A is the size of the constructed circuit, and B is the size of the captured wires.

Language Construct	PQ	*Proto-Quipper-RA*
Wire type	`Qubit{I}`	Qubit^I
Tensor type	`(A, B)`	$A \otimes B$
Dependent list type	`List[i<I] A`	$\mathsf{List}_{i<I}\, A$
Circuit type	`Circ[I](T,U)`	$\mathsf{Circ}^I(T, U)$
Arrow type	`A -o[I,J] B`	$A \multimap_J^I B$
Dependent function type	`forall[I,J] i. A`	$\Pi_J^I i.A$
Empty list	`[]`	nil
Right-cons	`M : N`	rcons M N
Abstraction	`\x :: A . M`	$\lambda x_A.M$
Index abstraction	`forall i. M`	$\Lambda i.M$
Index application	`M @ I`	M @ I

Table 3.1: Translation table between PQ and *Proto-Quipper-RA*

3.4.3 Inference Algorithm and Semantic Judgments

QuRA's inference algorithm operates in two main passes over the program's *Abstract Syntax Tree* (AST):

1. The **first pass** performs a Hindley-Milner style inference [25, 28], focusing on basic type checking while ignoring refinements and effects. Its purpose is to identify fundamental type errors and annotate the AST with base type information to facilitate the subsequent pass.

2. The **second pass** focuses on resource estimation. With base types already established, this pass synthesizes and checks refinements and effects, computing parametric size upper bounds based on the program's structure.

The tool relies on SMT [3] solvers like CVC5 [2] to check the validity of inequalities between index terms by formulating them as satisfiability problems, thereby automating the verification of non-trivial parametric bounds.

3.4.4 Resource Estimation Example

The Quantum Fourier Transform (QFT) is a fundamental quantum algorithm, conceptually serving as the quantum analogue of the Discrete Fourier Transform (DFT) [29]. It is a key subroutine in many other quantum algorithms that depend on analyzing the periodicity of a state, including Shor's algorithm [38].

When implemented on a register of n qubits, the QFT circuit follows a distinct structure. Each iteration of the QFT typically involves performing a sequence of controlled rotation gates on a target qubit, with each rotation controlled by other qubits in the input register. Last, applying a Hadamard gate to the target qubit. The overall QFT circuit on an n-qubit register is schematically represented in Figure 3.4.

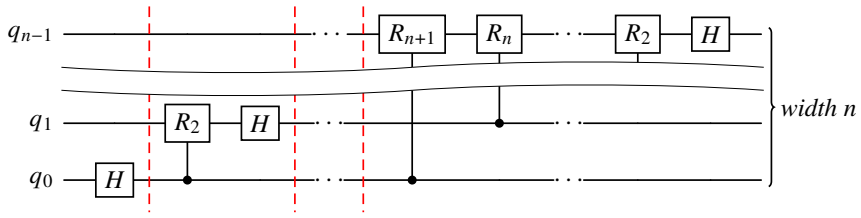


Figure 3.4: The generic QFT circuit on an n-qubit register.

Inspecting the circuit in the previous figure, we can intuitively estimate the resource requirements for a QFT circuit on n qubits as follows:

- **Width:** For an input list of length n qubits, the width of the QFT circuit is equal to n.
- **Gate Count:** The total gate count for an n-qubit QFT circuit, where the i-th iteration (starting from $i = 0$) consists of $i + 1$ gates, is given by the sum:

$$\sum_{i=0}^{n-1}(i+1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- **Depth:** The depth of the QFT circuit is a local metric, meaning it varies per wire. If all n input qubits begin at an initial depth d, the i-th output qubit is expected to have a depth of $d + n + i$.

The quantum Fourier program can be found in Program 3.1.

```
1 -- qft.pq annotated with width and depth analysis
2 --- HELPER FUNCTIONS ---
3 -- invert the list of intermediate qubits at iteration iter
```

```
qrev = forall iter. forall d.
  \reg :: List[i<iter] Qubit{d+iter+i}.
    let revStep = lift forall step.
      \(rev, q) :: (List[i<step] Qubit{d+2*iter-(i+1)}, Qubit{d+2*iter-(step+1)}).
        rev:q in
    fold(revStep, [], reg)
-- apply the controlled rotation gate to the target qubit trg at iteration iter
rotate = forall d. forall iter. lift forall step.
  \((ctrls, trg), ctrl)::((List[i<step] Qubit{d+iter+i+1}, Qubit{d+iter+step}), Qubit{d+
      iter+step}).
    let (ctrl, trg) = (force cr @((iter+1)-step) @(d+iter+step) @(d+iter+step)) ctrl trg in
    (ctrls:ctrl, trg) -- :: (List[i<step+1] Qubit{d+iter+i+1}, Qubit{d+iter+step+1})
--- QUANTUM FOURIER TRANSFORM ---
--- Parameters:
--- n     : size of the input to the QFT
--- d     : initial depth of the input qubits
--- iter  : current iteration of the QFT
-- apply the Quantum Fourier Transform to n qubits at depth d
qft :: ![0](forall[0,0] n. forall[0,0] d. List[i<n] Qubit{d} -o[n,0] List[i<n] Qubit{d+n+i
    })
qft n d reg =
    let qftIter = lift forall iter. -- define the iteration of the QFT
      \(ctrls, trg)::(List[i<iter] Qubit{d+iter+i}, Qubit{d}).
        let revctrls = (force qrev @iter @d) ctrls in -- List[i<iter] Qubit{d+2*iter-(i+1)}
        let (ctrls, trg) = fold(force rotate @d @iter, ([], trg), revctrls) in
        -- note (ctrls, trg) :: (List[i<iter] Qubit{d+iter+i+1}, Qubit{d+2*iter})
        let trg = (force hadamard @(d+2*iter)) trg in
        ctrls:trg -- List[i<iter+1] Qubit{d+iter+1+i}
    in fold(qftIter, [], reg) -- List[i<n] Qubit{d+n+i}
```

Program 3.1: PQ implementation of the Quantum Fourier Transform, annotated with width and depth signatures.

We can analyze the width and depth of the program by running QuRA on it:

```
$ qura qft.pq -g width -l depth
```

and obtain:

```
Analyzed file 'qft.pq'.
Checked type, width, depth.

qrev :: ![0](forall[0, 0] iter. forall[0, 0] d. List[i < iter] Qubit{d + iter
    + i} -o[iter, 0] List[i0 < iter] Qubit{d + 2 * iter - (i0 +

rotate :: ![0](forall[0, 0] d. forall[0, 0] iter. ![0](forall[0, 0] step.
    ((List[i < step] Qubit{d + iter + i + 1}, Qubit{d + iter + stepit{1 + d +
    iter + step})))
```

```
qft :: ![0](forall[0, 0] n. forall[0, 0] d. List[i < n] Qubit{d} -o[n, 0]
    List[i < n] Qubit{d + n + i})
```

Inspecting the signature of the `qft` confirms the quantities of the metrics that we postulated earlier by inspecting the circuit:

- `![0]`: the function is duplicable and does not build any circuit, so the *width* of the generated circuit is zero.
- `forall[0, 0] n. forall[0, 0] d.`: are the index abstractions of the function, acting as generic parameters. n is the size of the input register, d is the initial depth of the input qubits. They will be used to infer the values of the metrics of the circuit.
- `List[i < n] Qubit{d} -o[n, 0] List[i < n] Qubit{d + n + i}`: this expression represents the input and output type of the `qft`, and can be split in three parts:
  - `List[i < n] Qubit{d}`: the input is a list of n qubits at depth d.
  - `-o[n, 0]`: the function produces a circuit of width n as a side effect.
  - `List[i < n] Qubit{d + n + i}`: the output is a list of n qubits, where the i-th qubit sits at depth $d + n + 1$.

If we were to execute the program with the `-g gatecount` flag to compute the gatecount, we would end up with the following output:

```
* Expected expression 'qft'
   to have type
    '![0](forall[0, 0] n. forall[0, 0] d. List[i < n] Qubit -o[n, 0] List[i <
       n] Qubit)',
   got
    '![0](forall[0, 0] n. forall[0, 0] d. List[i < n] Qubit -o[sum[iter <
       n]iter + 1, 0] List[i < n] Qubit)'
   instead
```

QuRA is telling us that the inferred gatecount it expects is $\sum_{i=0}^{n-1}(i+1)$, and since the current program is annotated for another global metric, width, there is a discrepancy. We could proceed by changing the annotations inside the program, but the error confirms that the inferred `gatecounts` are, in fact, what we were expecting.

Chapter 4

Enabling Compilation and Interpretation Capabilities in QuRA

In the previous chapters, we introduced the syntax, type system, and operational semantics of *Proto-Quipper-RA*, together with the QuRA tool that builds upon this formal foundation [9]. At its core, QuRA is a framework designed for the *static resource analysis* of quantum programs: given a well-typed PQ program, it can estimate resource metrics such as circuit size, depth, and gate counts *without ever generating the corresponding circuits*.

This approach is representative of the current *state-of-the-art* in resource-aware quantum programming. By symbolically reasoning over program structure rather than manipulating explicit qubits and gates, QuRA achieves several important goals:

- It guarantees strong *type safety*, ensuring that well-typed programs are free from a wide class of runtime errors.
- It enables *parametric reasoning* on entire families of circuits: instead of compiling a single instance, the tool can derive expressions for resource usage as functions of input parameters.
- It supports *quantitative analyses* without requiring access to specific quantum hardware backends, making it suitable for both theoretical studies and early-stage algorithm design.

However, QuRA is just an analyzer: it does not execute programs and therefore does produce concrete circuits. In other words, PQ lacks an interpreter. While this state of affair suffices for algorithm analysis, it is unsatisfactory as soon as we plan to:

1. compute and export export the verified circuits to standardized formats such as QASM for execution on real quantum hardware, and

2. verify that the size of computed circuit does not exceed the predicted resource usage,

3. inspect intermediate configurations to better understand how program semantics translate into concrete gate sequences.

In this chapter, we present the first main contribution of this thesis: an *extension* of QuRA with a complete *interpreter*[1] implemented in Haskellthat enables the tool to *interpret* well-typed PQ programs. Concretely, our implementation introduces an operational layer that reduces expressions using the *big-step semantics* detailed in Section 4.1, following very closely the previously presented *Proto-Quipper-RA* operational semantics from Section 3.3.2. By doing so, *configurations* are produced, consisting of:

1. a residual program expression, and

2. the quantum circuit constructed so far.

This bridges the gap between symbolic analysis and runtime: instead of reasoning only about abstract resource formulas, we can now recover the exact circuit generated by evaluating a program.

To support this, we extend the operational semantics of *Proto-Quipper-RA* with additional reduction rules and error cases tailored to the implementation. These extensions ensure that all language constructs can be evaluated systematically and that potential errors are handled gracefully.

A central aspect of the evaluation process is the treatment of function calls. Each call is replaced by the body of the corresponding definition, wrapped and abstracted over its variables according to the typing information provided by QuRA. This ensures that the evaluation respects the type system while faithfully representing the intended behavior of the program.

The evaluation process begins with the construction of a complete program expression starting from the required main function. The functions encountered are substituted in the body where they appear, wrapped using the typing information provided by QuRA, after which the initial configuration is created: an identity circuit with an empty label context paired with

[1] davidesonno/qura on GitHub

the constructed term. From there, evaluation proceeds according to the extended big-step semantics: each quantum operation updates the circuit and the label context, progressively building the final circuit until the expression is fully reduced.

To demonstrate the capabilities of our implementation, we revisit the quantum teleportation protocol introduced earlier in Figure 2.2 [5]. By examining the resulting configuration, we verify that the generated circuit exactly matches the CRL representation defined previously, thereby validating the correctness and consistency of our evaluation strategy.

In the sections that follow, we describe the extended reduction rules, the substitution and variable-wrapping mechanisms, and the evaluation procedure for configurations. Finally, we present examples that demonstrate the evaluation of non-trivial expressions and the resulting circuits, showing how `QuRA` will be able to compute concrete configurations from `PQ` programs.

4.1 Extending *Proto-Quipper-RA* Evaluation Rules

When we first introduced the evaluation rules for the *Proto-Quipper-RA* language, we did not have any strategy for the evaluation of the constructs introduced later by `QuRA`. To be able to create circuit representations, we first have to extend the existing rules to also account for new terms, namely the index abstraction and index application, to create error cases to signal irreducible expressions, and to modify the existing rules to also consider the constants representing the quantum operations (recall Figure 3.3).

The extended evaluation rules are shown in Figures 4.1 and 4.2. These are quite more complex, so let us proceed by steps, inspecting the old rules that got changed first.

4.1.1 Updated Rules

The rule for evaluating an application is now composed of three steps; earlier, we only evaluated this expression if the first term was an abstraction, whereas it is enough that the first term *evaluates to* an abstraction. Once the first term is reduced to an abstraction and the second term is reduced to a value, we can proceed with evaluating the result of the substitution of the reduced value inside the abstraction. Here, the reduction would encounter an error in the case of the first term not evaluating to an abstraction (rules APP-ABS and APP-ERR).

Starting with Figure 4.1, the rule for evaluating force follows a similar scheme, as it

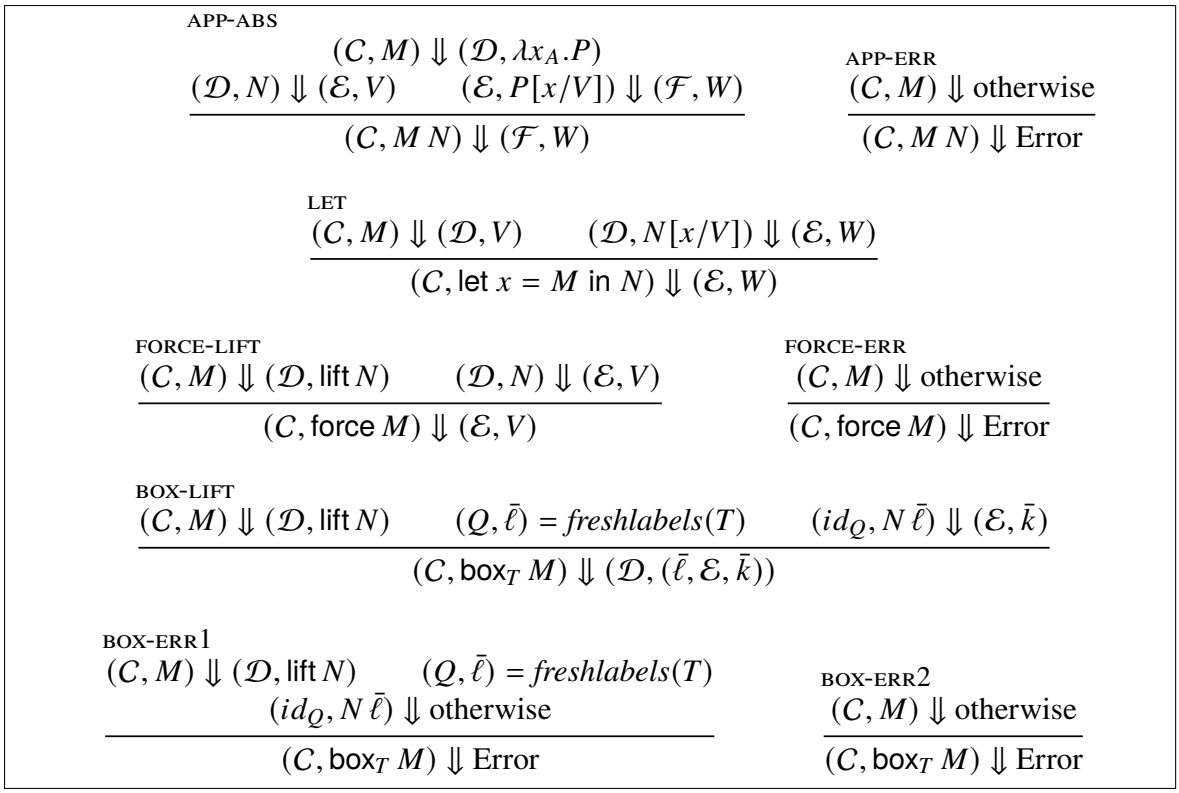


Figure 4.1: (i) The updated evaluation rules of QuRA expressions.

also evaluates the term in search of a lift, rule FORCE-LIFT rather than requiring it directly as its argument, an error is otherwise handled with the FORCE-ERR rule. This is even more clear in the BOX rule, where the reduction of the argument towards of a lifted term modifies the underlying circuit, which will appear in the resulting configuration, rule BOX-LIFT. The attentive reader might have noticed that the *freshlabels* function is missing a term. As we mentioned in Section 3.4.2, the introduction of explicit terms to operate on indices has simplified the box term by removing information about the index context which is no longer required. Evaluation of a boxed term can fail whenever a lifted term is not found (rule BOX-ERR2) or if the boxed expression N, applied to fresh labels in the identity context, does not output a wire bundle, rule BOX-ERR1. Boxed circuits are meant to represent independent circuit expressions that can be fully evaluated by themselves, producing the updated labels as a result of the computation.

Moving onto Figure 4.2, the APPLY rule has become more complex: we now allow terms to be reducible to boxed circuits and quantum operations. The first scenario is exactly the same of *Proto-Quipper-RA*, with the *append* function defined in the same way. In the case of a quantum operation, however, applying a wire bundle is quite easier, since the behavior of the operation is fixed by construction. The *appendqop* function simply creates a fresh set of

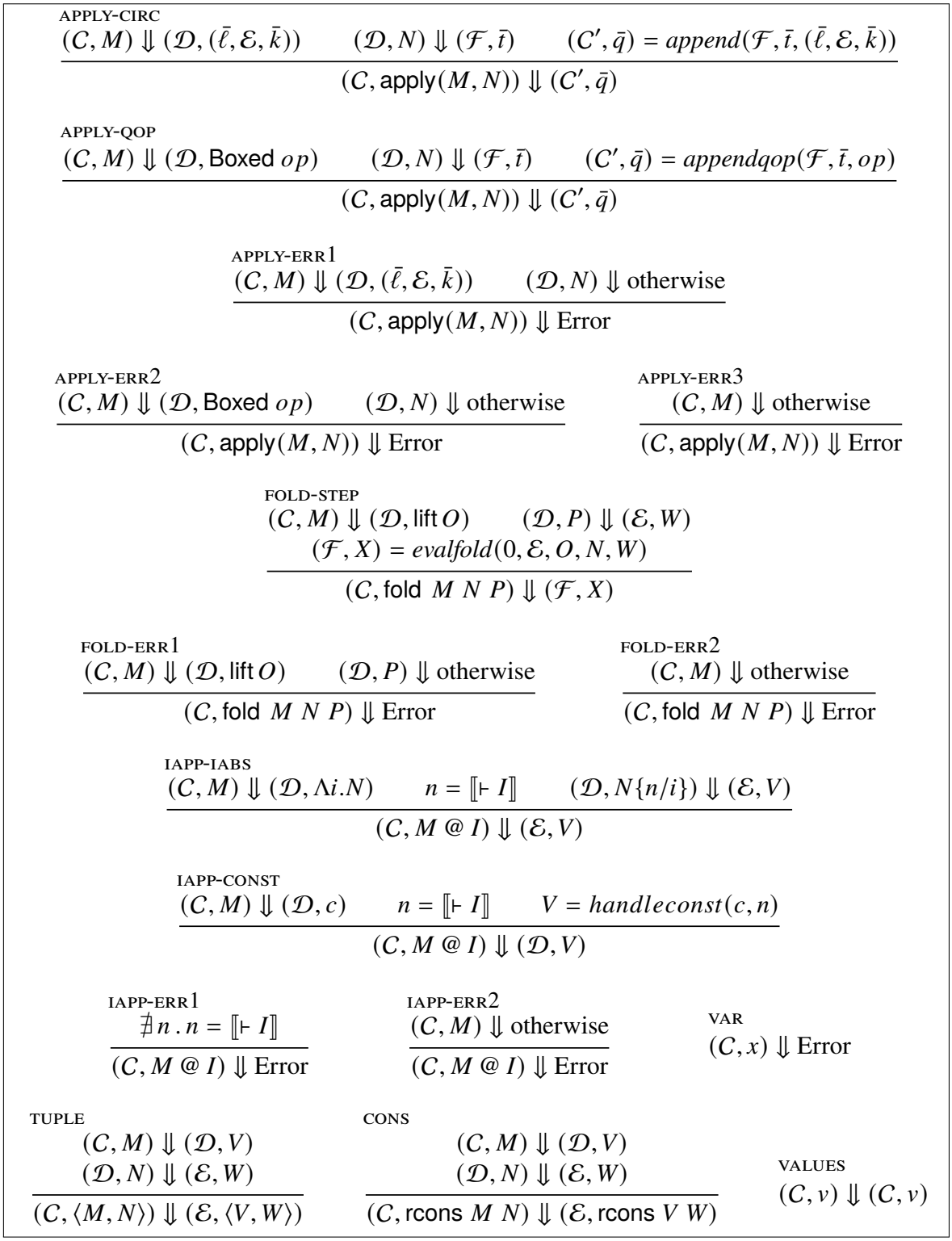


Figure 4.2: (ii) The updated evaluation rules of QuRA expressions.

output labels using the current label context of the configuration, also providing an updated version of it with the new labels such that there are no duplicates. The quantum operation is then appended to the CRL of the circuit using the result of the evaluation of the second argument of *apply* as input labels, and the fresh labels as outputs.

Definition 4.1 (*appendqop*)**.** We define *appending a quantum operation op to C on $\bar{\ell}$*, which we write $appendqop(C, \bar{\ell}, op)$, as the pair of a circuit and a wire bundle computed as follows:

1. Retrieve the output type T of the operation op.
2. Create fresh labels $\bar{k}$ of type T and different from those in the label context Q of C.
3. Define the new circuit $\mathcal{E} = C; op(\bar{\ell}) \rightarrow \bar{k}$
4. Finally, return $(\mathcal{E}, \bar{k})$.

Similarly to previous rules, the evaluation of apply fails if the terms do not reduce correctly, as shown in rules APPLY-ERR1, APPLY-ERR2, and APPLY-ERR3.

The evaluation of a fold changed since QuRA moved its explicit index towards the terms types. The rule changes accordingly and now requires an auxiliary method *evalfold*, which will be presented in a moment. Except for this change, the FOLD-NIL rule remained the same, and the function and input term of the fold do not need to be, respectively, a lifted term and a value directly, but we take the time to evaluate them first.

Definition 4.2 (*evalfold*)**.** We define the evaluation of a term $\mathsf{fold}\ \mathsf{lift}\ P\ V\ W$ in a circuit C at iteration n, which we write $evalfold(n, C, P, V, W)$, as the configuration (C', V') obtained as follows:

- If W is the empty list nil, return V.
- If $W = \mathsf{rcons}\ X'\ X$:
  - evaluate $(C, P\ @\ n)$ and obtain $(\mathcal{D}, Y)$;
  - evaluate $(\mathcal{D}, Y\ \langle V, X \rangle)$ and obtain the configuration $(\mathcal{E}, V')$ with the updated accumulator;
  - recur on $evalfold(n + 1, \mathcal{E}, \mathsf{fold}\ M\ V'\ X')$.
- if at any step, any of the terms are not in the required form, return an error.

4.1.2 Newly Added Rules

The new element of the language introduced by `QuRA` and presented in Section 3.4.2 is the index application. An index application is structurally very similar to a standard term application. It requires the first term to be reduced to an index abstraction $i \mathbin{@} N$, in order to substitute its index variable with the index specified in the index application. Moreover, the index has to be evaluated to a natural number before substituting, raising an error otherwise. The value of a closed index is represented by the expression $[\![\vdash I]\!]$. Expressions' semantic is trivial after the abstract indexes have been interpreted, which happens during type-checking. The interested reader might refer to the appendix in [9]. During the circuit construction stage, the value of an index is trivial, as it corresponds to the evaluation of common arithmetic operations. Although the standard behavior of an application is to provide an argument to an abstraction, this is not the only allowed case in `QuRA`. The first term of the application could also evaluate to a constant, described by the IAPP-CONST rule: this is quite useful when working with parametric constants such as rotations or $\mathsf{MakeUnitList}\, n$. The function *handleconst* is used to generate an expression that pairs the constant with the index on a case-by-case basis: if the constant is a boxed quantum operation the application does nothing, otherwise the natural number is used to generate an element of the parametric constant family. As a simple example of an index applied to a constant, consider how the QFT algorithm in Figure 3.4 applies controlled rotations R_n using a parametric index. Once the index is evaluated to a number n and applied to $\mathsf{MakeRGate}_{n,I}$, the quantum operation $\mathsf{R}_I\, n$ is created.

4.1.3 Simplifications and Minor Rules

The RETURN rule is removed, as return is no longer part of the language. The LET rule handles pattern matching, namely on variables, tuples, and lists. Variable substitution is handled with the usual capture-avoiding substitution; tuple substitution can be decomposed into a chain of single variable substitution, similar to the old LET-PAIR rule and list substitution can be done in the same way. Note that it is only possible to substitute a tuple of variables for a tuple of expressions with the same shape. A list of variables can be substituted with a list of expressions of equal or larger size; in the latter case, the final variable in the pattern will be assigned to a list with the remaining elements.

We adjust the definitions of tuples and lists as they are considered to be values only if their elements are also values. This is checked with the TUPLE and CONS rules.

As a final remark, in PQ the syntactic distinction between terms and values is no longer present: all constructs are treated uniformly as expressions. Nevertheless, many constructs that were formerly classified as values still behave as such, in the sense that their evaluation terminates immediately and simply returns the construct itself (rule VALUES). A notable exception is the case of variables. According to the VAR rule, evaluating a standalone variable raises an error, since variables are not intended to appear in isolation during evaluation. Instead, they must always be bound within a `let` expression or an abstraction, where substitution is handled by the corresponding evaluation rules.

4.2 Assembling the Circuit Configuration

Having defined the rules to evaluate an expression representing a *Proto-Quipper-RA* program, we are left with the task of merging the top level definitions found in the program to assemble the complete term. It is definitely not enough to simply substitute the body of a definition whenever its name appears in another term. In order for the function to insert to be *functional*, it would need the interfaces to apply arguments to its body. In practice, we are saying that we need to *wrap* the body of the function with abstractions and index abstractions according to the type of function parameters. For example, consider the function: $\mathsf{add}(x, y) := x + y$, where x, y are the variables, in the body $(\mathsf{add}\ 1)\ 2$. Naively replacing add would result in the expression $((x + y)\ 1)\ 2$, but, as we can notice, there is no way to apply the arguments to the operation. What is required to do is wrap add with $\lambda y.\lambda x.$ and obtain the final expression $((\lambda y.(\lambda x.(x + y)))\ 1)\ 2$.

Luckily, QuRA's inference algorithm provides us with a collection of all definitions found during the parsing and type-checking of the input file, together with their signatures and parameters. We can use those to infer the required abstraction type and the names of the variables they should bound. The procedure is fairly simple, as it requires one to scan the signature of the definition while also sliding over the list of formal parameters.

Up to now, there was no need for a *Proto-Quipper-RA* program to have an entry point. This was of course not needed at all, as QuRA intended use was to analyze quantum programs *without* running them. Our extension to the tool aims to produce quantum circuits that can be run on quantum devices; therefore, programs should h ave a main routine in which to prepare all the input values needed for the functions.

The last step in preparing the initial configuration is to pair an empty circuit with the term obtained by substituting the definitions inside the main. The fresh circuit is going to be represented by a id_Q in which $Q = \bullet$, an initially empty set of labels with no operations at all applied to it. During the evaluation of the program expression, quantum operations are appended by concatenating the boxed circuit $(\bar{\ell}, op, \bar{k})$ to C: $C' = C; op(\bar{\ell}) \to \bar{k}$, and then new output wires are added to the context.

4.3 Program Execution and Command-Line Options

The introduced mechanism is implemented in the tools itself, as the evaluation of a program now becomes `QuRA` default behavior, which tries to generate the corresponding circuit whenever a `main` function is defined. Running the command

```
qura FILE [-g|--global-metric-analysis METRIC] [-l|--local-metric-analysis
    METRIC]
```

`QuRA` type-checks the `PQ` program, infers the required metrics, and starts the circuit evaluation process. However, the evaluation phase can be omitted using the `--no-run` flag.

Another important option is *qubit recycling*, which structurally changes the number of wires of the generated circuits: `--no-recycling` forces the circuit to initialize new wires at depth zero, which means that the discarded wires will never be used again and that we always employ fresh bits or qubits. Whenever a program uses auxiliary qubits to perform computation and discards them, the wires are idle, and it would be a great optimization to be able to reuse them. The number of wires needed could explode if the circuit needs to perform many iterations on fresh qubits. The introduced flag will not affect explicitly the width of the circuit generated, as the CRL notation only represents the wires at a high level of abstraction. Recycling only affects the circuit whenever it is required to convert it to some specific language, as in Chapter 6 when we will convert the circuits to `OpenQASM 3.0`.

4.4 Inspecting the Evaluations of Simple Programs

Having defined how to assemble the initial circuit-term configuration to evaluate, we can now explore how our extended tool behaves on concrete examples. This section presents a series of small programs whose evaluation demonstrates the interaction between expressions and quantum circuits, used to test `QuRA` builds. By observing the resulting configurations,

we gain insight into how classical computations are reduced, how quantum operations are introduced, and how the circuit structure evolves step by step. The outputs are obtained by running the command `qura` without any additional flags.

4.4.1 Programs with No Side-Effects

We start by considering a simple program that does not introduce any quantum operations or modify the circuit, behaving as a standard lambda-calculus expression. This example serves as a baseline, showing that our evaluation strategy behaves as expected when no qubits are allocated and no gates are applied. In such cases, the resulting circuit remains the empty identity circuit, whereas the expression is fully reduced to a value. It is useful to test the evaluation mechanism on programs that involve only abstractions, function application, and basic constructs such as `let` and `let-pair`.

04.pq Consider the following program `04.pq`, which combines all of these elements:

```
1 main :: !(Qubit -o Qubit)
2 main = let id = (lift $ \f :: (Qubit -o Qubit) . f)
3     in let (id1, id2) = (force id, force id)
4         in id1 (id2  (\q :: Qubit . q))
```

Program 4.1: The test program `04.pq`

This program defines an identity function `id`, which takes as input a function from qubits to qubits and returns it unchanged. It duplicates id into `id1` and `id2`, then applies id2 to the qubit identity function; id1 is then applied to the result of that.

From the grammar and rules presented earlier, we know that:

- Abstractions are *values*, they evaluate to themselves and are not reduced further.
- Applying an abstraction to an argument results in the substitution of the argument into the abstraction's body.
- The `let` construct behaves similarly, binding values to variables and substituting them in the expression that follows.

Applying these principles, we expect that the chain of identities in `04.pq` ultimately reduces to the qubit identity function:

$$\lambda q : \mathsf{Qubit}.q$$

Running QuRA on the program confirms this intuition:

```
File 'test/Interpreter/pos/04.pq', produced circuit:
> Label Context: [empty]
> Operations:
[no operations]

While evaluating to:
> (\q :: Qubit . q)
```

As expected, the final expression is the identity function on qubits, and the circuit remains *empty* since no quantum operations are involved.

4.4.2 Programs Involving Quantum Operations

Next, we examine programs that introduce quantum operations, such as qubit allocations, gate applications, and measurements. These examples showcase how the evaluation process appends new quantum gates to the circuit and updates the label context as new qubits are introduced or modified.

06.pq Among the simplest programs we could think of is one that simply initializes qubits to 0 and 1. This program can be written in PQ in this way:

```
1 main :: ! (Qubit, Qubit)
2 main = let q = apply(QInit0, ()) in
3          let p = apply(QInit1, ()) in
4          (q, p)
```

Program 4.2: The test program `06.pq`

Before going further, let us briefly analyze the program: the `main` is build of a chain of `let` operations that assigns to qubits an *apply*, with arguments QInit_0 and `()`, and store them in a tuple. Here, QInit is the quantum operation that initializes a qubit, while the empty parentheses represent the unit $*$. Recall from Section 4.1.1 that the second argument has to be a wire bundle.

Evaluating the program, we would expect that at least the qubit initialization operations are added to the circuit operations, acting on an empty label and producing an output label that should also be present in the label context. To verify this, we run QuRA and inspect the outputs:

```
File 'test/Interpreter/pos/06.pq', produced circuit:
> Label Context: q0:Qubit, q1:Qubit
> Operations:
QInit0 (*) -> q0;
QInit1 (*) -> q1.

While evaluating to:
> (q0, q1)
```

It is worth noting that the label context now contains the qubits q_0 and q_1, resulting from the application of QInit_0* and QInit_1*. They correctly transform an empty wire into a qubit, and the result of the operation is a label; labels are values, and therefore the expression is not further reduced.

09.pq We will now present an example program that uses an user-defined `bell` function as subroutine, boxed in the main function and used twice. In particular, the main initializes two qubits in a really similar way to the program presented earlier, then applies them to the boxed `bell` two times. Notice that in order to be able to use other functions in PQ, we have to force them.

```
1  bell :: !((Qubit, Qubit) -o (Qubit, Qubit))
2  bell (q1, q2) =
3      let q1 = apply(Hadamard @0, q1) in
4      let (q1, q2) = apply(CNot @0 @0, (q1, q2)) in
5      (q1, q2)
6
7  main :: !(Qubit, Qubit)
8  main =
9      let q = apply(QInit0, ()) in
10     let p = apply(QInit1, ()) in
11     let boxedBell = box bell in
12     let (q, p) = apply(boxedBell, (q, p)) in
13     apply(boxedBell, (q, p))
```

Program 4.3: The test program `09.pq`

We used two new quantum operations: H and CNot and we can notice the presence of index applications, for example, in the term H @ 0, which will then evaluate to the boxed circuit $(q_1, \mathsf{H}_0, q_1')$. Those are fundamental for QuRA to be able to infer resource usage. A PQ program needs to be well-typed, and hence in the programs indices should be passed to the index abstraction that require them using dummy values such as zeros is also a possibility. In upcoming scenarios, the indexes willplay the role of parameters and will help us describe whole circuit families.

The function `bell` is boxed and then used twice inside the `main`. Evaluating the program correctly puts the qubits in the Bell state a first time and then a second:

```
File 'test/Interpreter/pos/09.pq', produced circuit:
> Label Context: q0:Qubit, q1:Qubit, q2:Qubit, q3:Qubit, q4:Qubit, q5:Qubit,
    q6:Qubit, q7:Qubit
> Operations:
QInit0 (*) -> q0;
QInit1 (*) -> q1;
Hadamard (q0) -> q2;
CNot ((q2, q1)) -> (q3, q4);
Hadamard (q3) -> q5;
CNot ((q5, q4)) -> (q6, q7).

While evaluating to:
> (q6, q7)
```

We can observe that two Hadamard gates and two CNots are applied, and we can verify the labeling with the following image:

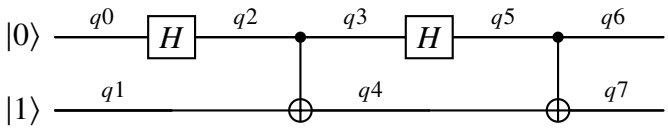


Figure 4.3: Label evolution of the doubly applied Bell state of program `09.pq`.

Boxing is a really useful part of the language, as it allows us to define re-usable circuits, so let us analyze it's behavior by considering the expression of the boxed `bell` function:

```
box (
  lift (\(q1, q2) :: (Qubit, Qubit) .
    (let q1 = apply ((Hadamard @ 0), q1) in
      (let (q1, q2) = apply (((CNot @ 0) @ 0), (q1, q2)) in
        (q1, q2)
)))
```

Following the box rule, we are required to reduce the expression to a lifted term first, which is already the case, and then evaluate that with the identity circuit. The evaluation of the lifted term inside the boxed function produces the following circuit:

```
Evaluating:
> Label Context: q0:Qubit, q1:Qubit, q2:Qubit, q3:Qubit, q4:Qubit
> Operations:
Hadamard (q0) -> q2;
CNot ((q2, q1)) -> (q3, q4).
> Expression:
(q3, q4)
```

which correctly represents the Bell state.

If we examine intermediate evaluation results during the complete evaluation of the PQ program, we can focus our attention in the moment of the second application of the boxed circuit to the underlying circuit being produced:

```
Evaluating:
> Label Context: q0:Qubit, q1:Qubit, q2:Qubit, q3:Qubit, q4:Qubit
> Operations:
QInit0 (*) -> q0;
QInit1 (*) -> q1;
Hadamard (q0) -> q2; <-- first box is appended here
CNot ((q2, q1)) -> (q3, q4).
> Expression:
apply((boxed ((q0, q1), [BOXED CIRC], (q3, q4))), (q3, q4)) <-- the term
    being evaluated
```

We expect the input labels of the boxed circuit and the labels that the circuit is applied to to become the same, whereas the other labels of the box to be mapped to names not existing in the underlying circuit. In fact, that is the case, as shown in the evaluated circuit:

```
Evaluating:
> Label Context: q0:Qubit, q1:Qubit, q2:Qubit, q3:Qubit, q4:Qubit, q5:Qubit,
    q6:Qubit, q7:Qubit
> Operations:
QInit0 (*) -> q0;
QInit1 (*) -> q1;
Hadamard (q0) -> q2; <-- first box appended here
CNot ((q2, q1)) -> (q3, q4);
Hadamard (q3) -> q5; <-- second box appended here
CNot ((q5, q4)) -> (q6, q7).
> Expression:
> (q6, q7)
```

which is also the fully evaluated configuration of the program.

4.5 Evaluating More Complex Programs

After validating our evaluation strategy for simple programs, we now turn to more complex PQ programs that implement real algorithms, exploiting the full spectrum og the language's features. These examples are designed to test the full expressive power of the language and to showcase how the interpreter handles non-trivial evaluation scenarios.

Unlike the simpler cases seen before, these programs interact with quantum resources in more sophisticated ways. During evaluation, qubits may be allocated, manipulated, and measured in different contexts, leading to non-trivial transformations of the circuit structure. This allows us to observe how our semantics correctly tracks label contexts, appends gates in the right order, and ensures that the resulting circuit faithfully represents the intended computation.

Finally, these examples provide a bridge to the next chapter, where the metrics of the evaluated circuits are compared with the estimates inferred by QuRA, with the assurance that the generated circuits faithfully match their expected structure.

4.5.1 Teleportation Protocol

Inspecting the programs presented in the last sections helped us understand the effects of quantum operations on the underlying circuit. This concept was the reason to introduce an operation buffer and a label context, that together represent the CRL expression of the circuit. When we first presented it, we used the teleportation protocol as example of its usage (Section 3.2.1). We will now verify that QuRA output matches the expected representation of the program; the PQ implementation is shown in Program 4.4.

```
1  -- put q and p into the entangled |+> state
2  bell :: !((Qubit, Qubit) -o (Qubit, Qubit))
3  bell (q, p) =
4      let q = (force hadamard @0) q in
5      let (q,p) = (force cnot @0 @0) q p in
6      (q,p)
7  --- Alice's part of the teleportation protocol
8  alice :: !((Qubit, Qubit) -o (Bit, Bit))
9  alice (p, r) =
10     let (r,p) = (force cnot @0 @0) r p in
11     let r = (force hadamard @0) r in
```

```
12      let c = (force meas @0) p in
13      let d = (force meas @0) r in
14      (c,d)
15  --- Bob's part of the teleportation protocol
16  bob :: !((Qubit, Bit, Bit) -o Qubit)
17  bob (q, c, d) =
18      let (c,q) = (force ccnot @0 @0) c q in
19      let (d,q) = (force ccz @0 @0) d q in
20      let _ = (force cdiscard @0) c  in
21      let _ = (force cdiscard @0) d  in
22      q
23  --- teleport the state of qubit r (at depth i) into qubit q
24  teleport :: !(Qubit -o Qubit)
25  teleport r =
26      let p = force qinit0 in
27      let q = force qinit0 in
28      let (q,p) = force bell (q,p) in
29      let (c,d) = force alice (p,r) in
30      force bob (q,c,d)
31  --- MAIN
32  main =
33      let q = force qinit1 -- initialize the qubit to |1>
34      in force teleport q  -- teleport it
```

Program 4.4: PQ implementation of the Quantum Teleportation Protocol.

The reader may notice a small change in the way quantum operations are applied. Previously, we applied operations directly to their arguments via the apply construct, as with the CNot of Program 4.4.2. However, this approach required matching the exact type of the operation, which means that *all* parameters needed to be provided at once, typically bundled into a tuple. This was somewhat restrictive, as it prevented partially applying an operation to one argument and supplying the remaining arguments later.

To improve usability, we now rely on the corresponding *library functions* that wrap each quantum operation. These wrappers expose the operations as ordinary functions, allowing us to apply arguments sequentially in a natural and functional style, as demonstrated with the CNot used in the teleportation example. Semantically, nothing changes, as the resulting circuits remain identical, but the syntax becomes more convenient and expressive.

Figure 4.4 shows the evaluation output of the program and the graphical representation of the circuit labels. The obtained operations rigidly follow the CRL representation introduced in Section 3.2.1, with the only exception being the qubit to teleport q, already present in the circuit.

```
> Operations:
QInit1 (*) -> q0;
QInit0 (*) -> q1;
QInit0 (*) -> q2;
Hadamard (q2) -> q3;
CNot ((q3, q1)) ->
    (q4, q5);
CNot ((q0, q5)) ->
    (q6, q7);
Hadamard (q6) -> q8;
Meas (q7) -> b0;
Meas (q8) -> b1;
CCNot ((b0, q4)) ->
    (b2, q9);
CCZ ((b1, q9)) ->
    (b3, q10);
CDiscard (b2) -> *;
CDiscard (b3) -> *.

While evaluating to:
> q10
```

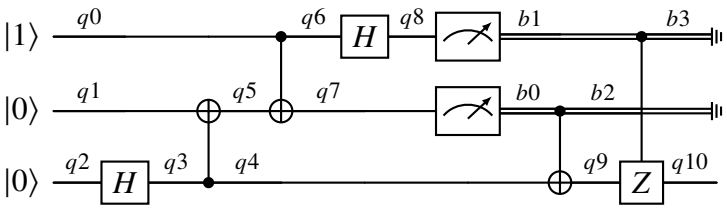


Figure 4.4: The evaluated configuration of the Quantum Teleportation Protocol and the corresponding circuit with annotated wires. The label context is omitted for simplicity.

4.5.2 Mapping the Hadamard Gate to a List

The next program we present serves as an introduction to parametric constructions. PQ allows the definition of functions whose output type can depend on its inputs. For example, it could be used to define a function which takes as input a list of n Qubits at depth d and outputs the same list with an Hadamard gate applied to each qubit.

The simple program we describe can be implemented in PQ as follows:

```
1 mapHadamard :: !forall n. List[_<n] Qubit -o List[_<n] Qubit
2 mapHadamard n list =
3     let hadaStep = lift forall step. \(qs ,q) :: (List[_<step] Qubit,
          Qubit). qs: (force hadamard @0) q
4     in fold(hadaStep , [], list)
```

Evaluating this program on lists of qubits with increasing length, we obtain the outputs captured in Figure 4.5. Besides the initialization of progressively more qubits, the number of Hadamard gates is also increasing, each being applied to a different qubit.

```
File
    'mapHadamard.pq',
    produced circuit:
> Label Context:
    q0:Qubit,
    q1:Qubit,
    q2:Qubit, q3:Qubit
> Operations:
QInit0 (*) -> q0;
QInit0 (*) -> q1;
Hadamard (q1) -> q2;
Hadamard (q0) -> q3.
```

(a) `mapHadamard` on a 2-qubit list.

```
File
    'mapHadamard.pq',
    produced circuit:
> Label Context:
    q0:Qubit,
    q1:Qubit,
    q2:Qubit,
    q3:Qubit,
    q4:Qubit, q5:Qubit
> Operations:
QInit0 (*) -> q0;
QInit0 (*) -> q1;
QInit0 (*) -> q2;
Hadamard (q2) -> q3;
Hadamard (q1) -> q4;
Hadamard (q0) -> q5.
```

(b) `mapHadamard` on a 3-qubit list.

```
File
    'mapHadamard.pq',
    produced circuit:
> Label Context:
    q0:Qubit,
    q1:Qubit,
    q2:Qubit,
    q3:Qubit,
    q4:Qubit,
    q5:Qubit,
    q6:Qubit, q7:Qubit
> Operations:
QInit0 (*) -> q0;
QInit0 (*) -> q1;
QInit0 (*) -> q2;
QInit0 (*) -> q3;
Hadamard (q3) -> q4;
Hadamard (q2) -> q5;
Hadamard (q1) -> q6;
Hadamard (q0) -> q7.
```

(c) `mapHadamard` on a 4-qubit list.

Figure 4.5: `QuRA` outputs of the function `mapHadamard` on different register lengths.

If we focus on the output of Figure 4.5c, we can verify the expected behavior of the function:

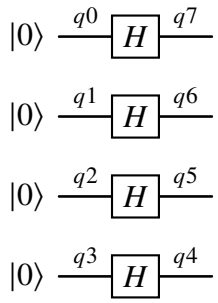


Figure 4.6: Label annotations on the output circuit of `mapHadamard` on 4 qubits.

4.5.3 Quantum Fourier Transform

The QFT program, whose code is presented in Appendix A.2, makes extensive use of folds, needed to iterate over a list of qubits parametrically over a certain index; in this program, the iteration is used to determine which controlled rotation to apply and to reverse lists.

As discussed in previous sections, the Quantum Fourier Transform (QFT) algorithm constructs a family of circuits in a systematic and well-defined manner, parametrically on n. For i going from 0 to $n-1$, $i-1$ controlled rotations are applied to the i-th qubit, followed by an Hadamard gate. The reader might refer to Fig. 3.4 to confirm the structure of the circuit.

From the evaluation of the program, we expect to find a chain of controlled rotations on

the wires, controlled by subsequent qubits:

```
File 'examples/qft.pq', produced circuit:
> Label Context: [...]
> Operations:
QInit0 (*) -> q0;
QInit0 (*) -> q1;
QInit0 (*) -> q2;
QInit0 (*) -> q3;
Hadamard (q3) -> q4;
CR2 ((q4, q2)) -> (q5, q6);
Hadamard (q6) -> q7;
CR3 ((q5, q1)) -> (q8, q9);
CR2 ((q7, q9)) -> (q10, q11);
Hadamard (q11) -> q12;
CR4 ((q8, q0)) -> (q13, q14);
CR3 ((q10, q14)) -> (q15, q16);
CR3 ((q12, q16)) -> (q17, q18);
Hadamard (q18) -> q19.

While evaluating to:
> ((((]:q13):q15):q17):q19)
```

We can verify the correctness of both the labels and the sequence of operations by reconstructing the annotated circuit from the output. This also allows us to confirm that the final expression, in this case the list of labels, indeed corresponds to the output wires of the circuit. The representation obtained is shown in Figure 4.7.

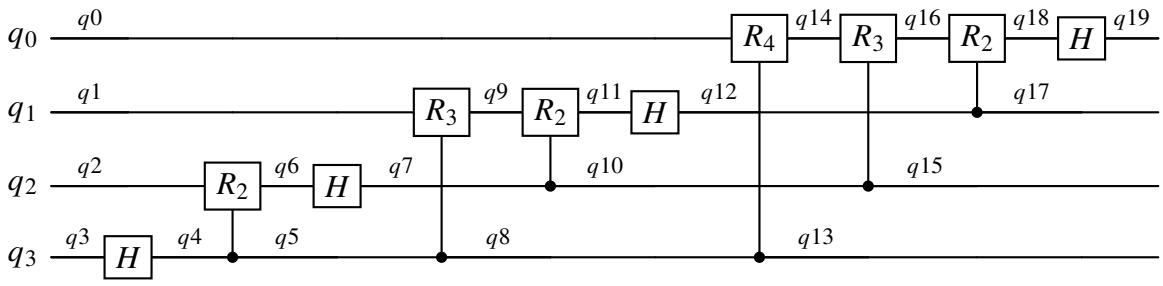


Figure 4.7: Wire names for the QFT circuit acting on 4 qubits.

Chapter 5

Validating Resource Estimates via Concrete Circuits

In the previous chapters, we introduced the evaluation of PQ programs and described how the resulting configurations capture the families of generated circuits. Although we have verified the *structural correctness* of these circuits by comparing their shapes with the corresponding CRL representations, we are still missing a detailed analysis of *resource metrics* such as gate counts, circuit depth, and qubit usage.

This chapter is devoted to assessing the accuracy of QuRA's resource estimation capabilities. Specifically, we compare the metrics inferred by QuRA with those computed directly from the concrete circuits produced during the evaluation of the program. Although QuRA is formally *sound* [9], meaning that its inferred bounds never underestimate the actual resource consumption, it is important to evaluate the *tightness* of these estimates. In other words, we investigate whether the bounds closely reflect the true cost of executing a program, or if they tend to be overly conservative. This empirical validation provides insight into the practical reliability of QuRA's estimates.

For clarity, we categorize quantum programs into two classes:

- **Non-Parametric Circuits:** Circuits whose structure and size are fixed, independent of any external parameter, such as input size or iteration count.

- **Parametric Circuit Families:** Circuits whose structure depends on input parameters, such as the number of qubits, the desired precision, or the number of iterations in the algorithm.

This classification allows us to examine both simple, fixed-size programs and scalable algorithmic families, providing a comprehensive assessment of QuRA's resource estimation performance.

5.1 CRL Metric Computation

To perform this validation, we analyze the circuits generated during program evaluation, computing their width, depth, and gate counts. It is important to note that QuRA analysis follows a strict rule: the width of a circuit is always calculated assuming that qubit recycling is possible, while the depth calculation always assumes that new initializations are performed at depth zero, and therefore without wire recycling. Gatecount is not affected by the notion of recycling, as the number of gates does not vary depending on which wires they are applied to.

To align QuRA results with the metrics computed for the generated circuits, whenever we want to compute depth, we must explicitly adopt the `--no-recycling` flag.

Before being able to compare the circuit resources, we have to formally define them for a circuit in the CRL notation.

Definition 5.1 (Width of a Circuit)**.** Let C be a circuit. The *width of* C, written $width(C)$, is defined recursively on C as follows:

$$\begin{aligned} width(id_Q) &= |Q|; \\ width(C; g(\bar{\ell}) \to \bar{k}) &= width(C) + ((|\bar{k}| - |\bar{\ell}|) \dot{-} reusable(C)); \end{aligned}$$

$$reusable(C) = width(C) - outputs(C),$$

where $|Q|$ is the cardinality of the domain of Q, $|\bar{\ell}|$ is the number of labels occurring in $\bar{\ell}$, and $n \dot{-} m$ denotes natural subtraction of m from n, which is 0 if $n \leq m$. $outputs(C)$ is the number of output wires of C.

The depth definition of a CRL circuit that we adopt in this chapter is slightly more compact than the generic one proposed in [10]. In our specific scenario, all circuits produced by a program are closed, that is the labels existing in the label context can only originate from the circuit itself. This simplifies the notion of depth, since we can simply assume that every label

starts at a depth of zero. The depth of a circuit followed by an operation can be computed with the more standard approach of putting the output labels of an operation at the depth of the deepest input label plus one, which is the depth of the operation itself.

Definition 5.2 (Depth of a Circuit)**.** Let C be a circuit and let $FL(\bar{\ell})$ denote the set of label names occurring in a wire bundle $\bar{\ell}$. The *depth of label t in C*, written $depth(C, t)$, is defined as follows:

$$depth(id_\bullet, t) = 0;$$

$$depth(C; g(\bar{\ell}) \to \bar{k}, t) = \begin{cases} \max\{depth(C, \ell) \mid \ell \in FL(\bar{\ell})\} + 1 & \text{if } t \in FL(\bar{k}), \\ depth(C, t) & \text{otherwise.} \end{cases}$$

As introduced earlier, the *gatecount* metric is quite easy as it is enough to count the number of operations that appear in the circuit, with the empty circuit and the meta-operations – such as QInit, CInit, QDiscard, CDiscard – having a gatecount of zero.

Definition 5.3 (Gate Count of a Circuit)**.** Let C be a circuit. The *gate count of C*, written $gatecount(C)$, is defined by structural induction on C as follows:

$$gatecount(id_Q) = 0;$$

$$gatecount(C; g(\bar{\ell}) \to \bar{k}) = gatecount(C),\ g \in \{\mathsf{QInit}, \mathsf{CInit}, \mathsf{QDiscard}, \mathsf{CDiscard}\};$$

$$gatecount(C; g(\bar{\ell}) \to \bar{k}) = gatecount(C) + 1,\ \text{otherwise.}$$

5.2 Resources Analysis of Non-Parametric Circuits

5.2.1 The Quantum Teleportation Algorithm

As a canonical example of a non-parametric program, we revisit the teleportation protocol. Let us start by executing `QuRA` on the program to collect the metrics values. To do so, we refine the code, presented earlier in Section 4.5.1, with the required signatures for our metrics. For this first example, we show the extended program annotated with *width* and *depth*, Program 5.1, and with *gatecount* and *depth*, provided in Appendix A.1.

```
1 -- teleportation.pq annotated for width and depth
2 --- put q and p into the entangled |+> state
3 bell :: ![0](forall[0,0] dq. forall[0,0] dp.
```

```
  (Qubit{dq}, Qubit{dp}) -o[2,0] (Qubit{max(dq+1, dp) + 1}, Qubit{max(dq+1, dp) + 1}))
bell dq dp (q, p) =
    let q = (force hadamard @dq) q in
    let (q,p) = (force cnot @dq+1 @dp) q p in
    (q,p)
--- Alice's part of the teleportation protocol
alice :: ![0](forall[0,0] dp. forall[0,0] dr.
  (Qubit{dp}, Qubit{dr}) -o[2,0] (Bit{max(dp, dr) + 2}, Bit{max(dp, dr) + 3}))
alice dp dr (p, r) =
    let (r,p) = (force cnot @dr @dp) r p in
    let r = (force hadamard @ max(dp, dr) + 1) r in
    let c = (force meas @ max(dp, dr) + 1) p in
    let d = (force meas @ max(dp, dr) + 2) r in
    (c,d)
--- Bob's part of the teleportation protocol
bob :: ![0](forall[0,0] dq. forall[0,0] dc. forall[0,0] dd.
  (Qubit{dq}, Bit{dc}, Bit{dd}) -o[3,0] Qubit{max(dd, max(dc, dq) + 1) + 1})
bob dq dc dd (q, c, d) =
    let (c,q) = (force ccnot @dc @dq) c q in
    let (d,q) = (force ccz @dd @ max(dc, dq) + 1) d q in
    let _ = (force cdiscard @ max(dc, dq) + 1) c  in
    let _ = (force cdiscard @ max(dd, max(dc, dq) + 1) + 1) d  in
    q
--- teleport the state of qubit r (at depth dr) into qubit q
teleport :: ![0](forall[0,0] dr. Qubit{dr} -o[3,0] Qubit{dr+6})
teleport dr r =
    let q = force qinit0 in
    let p = force qinit0 in
    let (q,p) = (force bell @0 @0) (q,p) in
    let (c,d) = (force alice @2 @dr) (p,r) in
    (force bob @2 @ max(2, dr) + 2 @ max(2, dr) + 3)  (q,c,d)
```

Program 5.1: PQ implementation of the Quantum Teleportation Protocol, annotated with width and depth signatures.

Width

```
$ qura teleportation-width-depth.pq -g width
[... more outputs, omitted for brevity ...]

teleport :: ![0](forall[0, 0] dr. Qubit -o[3, 0] Qubit)
```

The annotation of the arrow type tells us that when applying this function to a qubit, it produces a circuit of width at most 3.

Depth

```
$ qura teleportation-gatecount-depth.pq -l depth --no-recycling
[... more outputs, omitted for brevity ...]

teleport :: !(forall dr. Qubit{dr} -o Qubit{dr + 6})
```

The depth annotations on the qubits `{dr}` and `{dr+6}`, tells us that the teleportation function takes as input a qubit at depth *dr* and outputs a qubit at depth *dr* + 6, for each possible value of *dr*.

Gatecount

```
$ qura teleportation-gatecount-depth.pq -g gatecount
[... more outputs, omitted for brevity ...]

teleport :: ![0](forall[0, 0] dr. Qubit -o[8, 0] Qubit)
```

The global annotation of the arrow type, `-o[8, 0]`, tells us that *teleportation* is a function that constructs a circuit composed of 8 gates.

Size of the Generated Circuit

From the previous `QuRA` outputs, we know that if we prepare a qubit to teleport using the *teleportation* function, we expect that the circuit is composed of 8 gates, has a width of 3, while placing the qubit 6 steps deeper.

To verify these values, we simply define a `main` function that initializes a qubit to $|0\rangle$, at depth 0, and applies the teleportation function to *teleport* it:

```
$ qura teleportation.pq
[... more outputs, omitted for brevity ...]

Size of the produced circuit:
> Metric Values:
 - Width: 3
 - Depth: 6
 - Gatecount: 8
```

With this simple program, we are able to verify that `QuRA` predicted metric values and that the generated program size are exactly the same.

5.3 Resources Analysis of Parametric Families of Circuits

For parametric algorithms, we automate the validation process by appending a minimal `main` function that initializes the required number of input qubits, passes them to the target function, and extracts the resulting resource metrics directly from the `QuRA` command-line output.

In the rest of this chapter, we will refer to the metrics computed on the produced CRL as *ground truth*.

5.3.1 The `mapHadamard` Function

As a first example of a program building a parametric family of circuits, we inspect the simple `mapHadamard` function, which applies the Hadamard gate to a list of qubits. The annotated program is presented in Figure 5.2. In this specific case, the global annotations work for both the width and the gatecount; this is, of course, not true in general.

```
1 mapHadamard :: ![0](forall [0 ,0] d. forall [0 ,0] n. (List[_<n] Qubit{d
    }) -o[n ,0] (List[_<n] Qubit{d+1}))
2 mapHadamard d n list =
3   let hadaStep = lift forall step. \(qs ,q) :: (List[_<step] Qubit{d+1},
      Qubit{d}). qs: (force hadamard @d) q
4   in fold(hadaStep , [], list)
```

Program 5.2: The `mapHadamard` function in Proto-Quipper-RA, with global and local annotations.

Of course, we expect that the program width is the same as the size of the input register, while the depth is 1, as one gate is applied to each qubit and the gatecount is equal to the number of Hadamard gates applied which is equal to the width of the circuit.

This can be easily verified with `QuRA`:

```
$ qura mapHadamard.pq -g width
[... more outputs, omitted for brevity ...]

mapHadamard :: ![0](forall[0, 0] d. forall[0, 0] n. List[_ < n] Qubit -o[n,
    0] List[_ < n] Qubit)
```

and:

```
$ qura mapHadamard.pq -g gatecount -l depth
[... more outputs, omitted for brevity ...]
```

```
mapHadamard :: ![0](forall[0, 0] d. forall[0, 0] n. List[_ < n] Qubit{d}
    -o[n, 0] List[_ < n] Qubit{d + 1})
```

The signatures of the function provide us with the following information:

- `mapHadamard` constructs a circuit of width at most n,
- the number of gates is n,
- and each output qubit sits ad depth $d + 1$.

Resource Metrics Results Evaluating the program on a register of n qubits initialized at depth $d = 0$ results in the values presented in Table 5.1, which confirm the `QuRA` estimates.

Width						
n	1	2	3	4	5	...
QuRA estimate: n	1	2	3	4	5	...
Ground truth:	1	2	3	4	5	...

(a) Widths

Depth						
n	1	2	3	4	5	...
QuRA estimate: 1	1	1	1	1	1	...
Ground truth:	1	1	1	1	1	...

(b) Depths

Gatecount						
n	1	2	3	4	5	...
QuRA estimate: n	1	2	3	4	5	...
Ground truth:	1	2	3	4	5	...

(c) Gatecounts

Table 5.1: Comparison of `QuRA`-inferred and ground-truth results for `mapHadamard` on n qubits.

5.3.2 The Quantum Fourier Transform

As QFT circuits [29] exhibit well-understood resource requirements in both gate count and depth, they provide a rigorous benchmark for validating `QuRA`'s symbolic estimations.

After updating the code in Program 3.1 to correctly behave for the gatecount analysis, Program 5.3, we obtain the two programs for which `QuRA` infers the required resources.

```
-- qft.pq annotated with gatecount and depth analysis
--- HELPER FUNCTIONS ---
-- invert the list of intermediate qubits at iteration iter
qrev = forall iter. forall d.
  \reg :: List[i<iter] Qubit{d+iter+i}.
    let revStep = lift forall step.
      \(rev, q) :: (List[i<step] Qubit{d+2*iter-(i+1)}, Qubit{d+2*iter-(step+1)}).
        rev:q in
    fold(revStep, [], reg)
-- apply the controlled rotation gate to the target qubit trg at iteration iter
rotate = forall d. forall iter. lift forall step.
  \((ctrls, trg), ctrl)::((List[i<step] Qubit{d+iter+i+1}, Qubit{d+iter+step}), Qubit{d+
      iter+step}).
    let (ctrl, trg) = (force cr @((iter+1)-step) @(d+iter+step) @(d+iter+step)) ctrl trg in
    (ctrls:ctrl, trg) -- :: (List[i<step+1] Qubit{d+iter+i+1}, Qubit{d+iter+step+1})
--- QUANTUM FOURIER TRANSFORM ---
--- Parameters:
--- n     : size of the input to the QFT
--- d     : initial depth of the input qubits
--- iter  : current iteration of the QFT
-- apply the Quantum Fourier Transform to n qubits at depth d
qft :: ![0](forall[0,0] n. forall[0,0] d. List[i<n] Qubit{d} -o[sum[iter < n]iter+1,0] List
    [i<n] Qubit{d+n+i})
qft n d reg =
    let qftIter = lift forall iter. -- define the iteration of the QFT
      \(ctrls, trg)::(List[i<iter] Qubit{d+iter+i}, Qubit{d}).
        let revctrls = (force qrev @iter @d) ctrls in -- List[i<iter] Qubit{d+2*iter-(i+1)}
        let (ctrls, trg) = fold(force rotate @d @iter, ([], trg), revctrls) in
        -- note (ctrls, trg) :: (List[i<iter] Qubit{d+iter+i+1}, Qubit{d+2*iter})
        let trg = (force hadamard @(d+2*iter)) trg in
        ctrls:trg -- List[i<iter+1] Qubit{d+iter+1+i}
    in fold(qftIter, [], reg) -- List[i<n] Qubit{d+n+i}
```

Program 5.3: PQ implementation of the Quantum Fourier Transform, annotated with gatecount and depth signatures.

Width

```
$ qura qft-width-depth.pq -g width
[... more outputs, omitted for brevity ...]

qft :: ![0](forall[0, 0] n. forall[0, 0] d. List[i < n] Qubit -o[n, 0] List[i
    < n] Qubit)
```

As expected, the QFT circuit takes as input a list of n qubits and returns a list of n qubits, without introducing any additional wire. We can inspect the size of the generated circuits in

the following table:

Width												
n	1	2	3	4	5	...	10	11	...	50	51	...
QuRA estimate: n	1	2	3	4	5	...	10	11	...	50	51	
Ground truth:	1	2	3	4	5	...	10	11	...	50	51	

Table 5.2: Comparison of QuRA-inferred widths and generated QFT circuit widths on n qubits.

Depth

```
$ qura qft-gatecount-depth.pq -l depth --no-recycling
[... more outputs, omitted for brevity ...]

qft :: !(forall n. forall d. List[i < n] Qubit{d} -o List[i < n] Qubit{d + n
    + i})
```

The function takes as input a list of n qubits at depth d and outputs a list of n qubits, where the i-th qubit sits at depth $d+n+i$. The depth of the whole circuit is then equal to the deepest output qubit, $d + n + (n - 1)$. In our testing scenario, input qubits are freshly initialized, therefore we can fix $d = 0$.

QuRA estimates hold, as we can see in the following results:

Depth												
n	1	2	3	4	5	...	10	11	...	50	51	...
QuRA estimate: $2n-1$	1	3	5	7	9	...	19	21	...	99	101	
Ground truth:	1	3	5	7	9	...	19	21	...	99	101	

Table 5.3: Comparison of QuRA-inferred depths and generated QFT circuit depths on n qubits.

Gatecount

```
$ qura qft-gatecount-depth.pq -g gatecount
[... more outputs, omitted for brevity ...]

qft :: ![0](forall[0, 0] n. forall[0, 0] d. List[i < n] Qubit -o[sum[iter <
    n]iter + 1, 0] List[i < n] Qubit)
```

The function `qft` produces a circuit whose size can be read in the arrow-type signature: `-o[sum[iter<n]iter+1, 0]`, which represents the quantity $\sum_{iter=0}^{n-1}(iter+1) = \frac{n(n+1)}{2}$.

The number of gates grows quite faster compared to the other metrics: in particular, gatecount grows quadratically, whereas the other metrics remain linear with reference to the input size. We can observe that the values inferred by `QuRA` match exactly the generated circuits:

Gatecount

n	1	2	3	4	5	...	10	11	...	50	51	...
`QuRA` estimate: $n(n+1)/2$	1	3	6	10	15	...	55	66	...	1275	1326	...
Ground truth:	1	3	6	10	15	...	55	66	...	1275	1326	...

Table 5.4: Comparison of `QuRA`-inferred gatecounts and generated QFT circuit gatecounts on n qubits.

5.3.3 Grover's Algorithm

Grover's search algorithm [23] is one of the many examples of quantum speedup, providing a quadratic improvement over classical exhaustive search. Its circuit structure consists of three main components: the initialization of a uniform superposition, the application of an *oracle* that marks the solution states, and the *Grover diffusion operator* that amplifies their amplitude. This sequence of oracle calls and diffusion steps is repeated for a number of iterations proportional to $\sqrt{N}$, where N is the size of the search space. In our analysis, we do not need that this values matches this rule, as we are only analyzing the resources of algorithms, which depend on their parameters.

Circuit Structure

The whole circuit is composed of the initialization of an ancilla and a register in superposition, applying the oracle and the *diffusion* operator for the given number of iterations. Together, the generic structure of the algorithm is shown in Figure 5.1, where U_f is the oracle.

Oracle Implementation

We select a simple oracle that outputs *true* whenever all qubits of the input register are equal to 1. This condition can be encoded with a multi-controlled NOT (MCNot), and we define the auxiliary function `nconj`(n, d) to represent the corresponding family of oracles as an $(n+2)$-controlled NOT applied to a register of initial depth d. The `PQ` implementation is shown in Program 5.4.

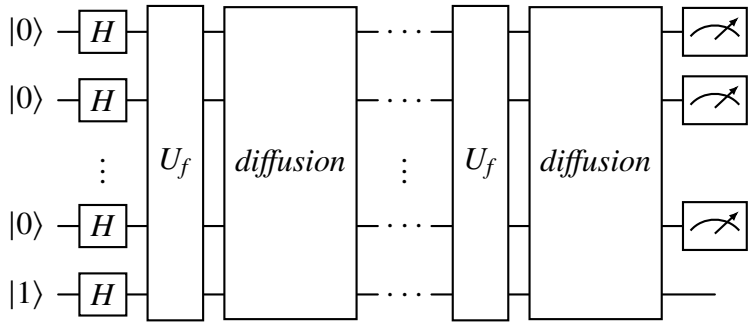


Figure 5.1: The generic Grover's algorithm circuit. U_f is the oracle function, *diffusion* is the diffusion operator.

```
1 -- The oracle nconj : nconj(x) = 1 <==> x = 11...1
2 nconj = forall n. forall d.
3   box $ lift \(reg, a) :: (List[_<n+2] Qubit{d}, Qubit{d}).
4     (force mcnot @ n @ d @ d) reg a
```

Program 5.4: PQ implementation of the *nconj* oracle for Grover's search algorithm, annotated with the width and depth of the *mcnot* implementation used [39].

The generic *MCNot*, which will also be used later in the diffusion operator, has been implemented using a function that creates a circuit according to the structure proposed by Siddhartha Sinha and Peter Russer [39]. The circuit is quite simple: it starts by assigning the intermediate value of the control to a first ancilla, using the first two controls Then, new ancillas are created at each step, using the last ancilla and the next control, until all the controls have been conjoined. The remaining ancilla will then be used to control the target qubit, and all the computations will be reversed using the same Toffoli gates applied backward.

Of course, many improvements can be made in terms of reducing the number of ancillas and gates, for example, using the last Toffoli directly on the target of the multi-controlled Not, but this is outside the scope of this example.

The PQ implementation of the multi-controlled-not is provided in Appendix A.3, whereas the circuit in Figure 5.2 shows the structure of the circuits generated using such function.

Grover's Diffusion Operator

This operator is used to amplify the amplitude of the state *marked* by the oracle. It consists of applying an Hadamard gate followed by an X gate to each qubit of the register, applying an MCNot controlled by the register and targeting an additional qubit, applying X and Hadamard

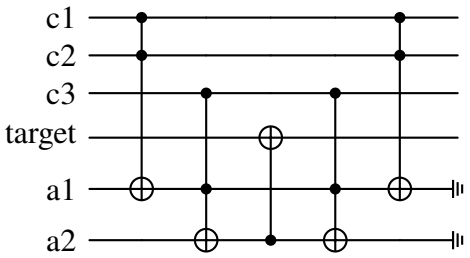


Figure 5.2: Visualization of the 3-CNot circuit, generated from the `mcnot.pq` function. The first three qubits are the controls, followed by the target. The last two wires are ancillas.

again. The diffusion operator on a three-qubit register is depicted in Figure 5.3.

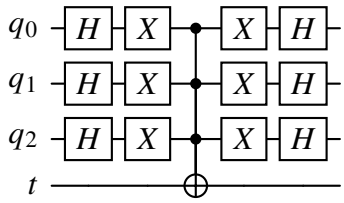


Figure 5.3: Diffusion operator of a 3-qubit register, which uses an ancillary qubit t.

Note that the multi-controlled not used in the analyzed implementation of Grover's algorithm is the same as the one used previously in the oracle.

Visualization of the Complete Circuit

The circuit portrayed in Figure 5.4 presents the structure of the algorithm for a three-qubit register, with the diffusion operator replaced by its implementation. The last substitution consists of replacing the *conj* oracle and the MCNots with the structure presented previously in this section.

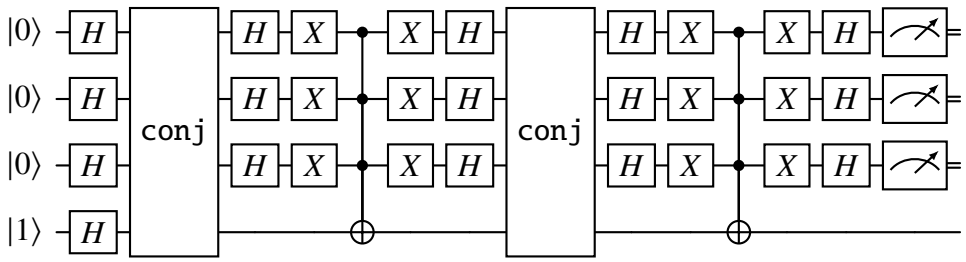


Figure 5.4: The circuit built by the `grover` function applied to the `conj` oracle, with $r = 2$ iterations.

Resource Analysis

The PQ implementation of Grover's algorithm is provided in Program 5.5 and Appendix A.4, respectively annotated with width and gatecount, plus depth. It defines a parametric family of circuits that perform r iterations on a register of size n, using an oracle of *size* os and depth od. Since that the oracle that we are using is implemented with an MCNot, the values of os an od depend on the chosen implementation of multi-controlled gates [39]; in our case, the depth of the oracle is $od = 2(n+2)$, while its size depends on wether we are considering the width or the gatecount. The width of the MCNot we use is $2(n+2)$ and requires $2(n+1)+1$ gates. The PQ implementation can be consulted in Appendix A.3.

```
--- GROVER'S ALGORITHM annotated with width and depth---
-- the diffusion operator on n qubits with ancilla a (all qubits at depth d)
diffusion :: ![0](forall[0,0] n. forall[0,0] d. List[_<n+2] Qubit{d} -o[0,0] Qubit{d} -o
    [2*(n+2), n+2] (List[_<n+2] Qubit{d+4+2*(n+2)}, Qubit{d+4+n}))
diffusion n d reg a =
  let reg = (force mapHadamard @ n+2 @ d) reg in
  -- begin negatively controlled not
  let reg = (force mapQnot @ n+2 @ d+1) reg in
  let (reg,a) = (force mcnot @ n @ d+2 @ d) reg a in
  let reg = (force mapQnot @ n+2 @ d+2+2*(n+2)) reg in
  -- end negatively controlled not
  let reg = (force mapHadamard @ n+2 @ d+3+2*(n+2)) reg in
  (reg, a)
-- perform a single grover iteration on n qubits at depth d, using an ancilla a at depth d,
    and an oracle of size os and depth od
groverIteration :: ![0](forall[0,0] n. forall[0,0] d. forall[0,0] os. forall[0,0] od.
  (forall[0,0] d. Circ[os]((List[_<n+2] Qubit{d}, Qubit{d}), (List[_<n+2] Qubit{d + od},
      Qubit{d + od})))
  -o[0,0] List[_<n+2] Qubit{d} -o[0,0] Qubit{d}
  -o[max(max(n+3,os),2*(n+2)),n+2] (List[_<n+2] Qubit{d+od+4+2*(n+2)}, Qubit{d+od+4+n}))
groverIteration n d os od oracle reg a =
  let (reg, a) = apply(oracle @ d, (reg, a)) in
  (force diffusion @ n @ d+od) reg a
-- run Grover's algorithm on an oracle of input size n, width os and depth od
grover :: ![0](forall[0, 0] r. forall[0, 0] n. forall[0, 0] os. forall[0, 0] od.
  (forall[0,0] d. Circ[os]((List[_<n+2] Qubit{d}, Qubit{d}), (List[_<n+2] Qubit{d+od},
      Qubit{d+od})))
  -o[max(max(n+3,os),2*(n+2)),0] List[_<n+2] Bit{2 + r*(od+4+2*(n+2))})
grover r n os od oracle =
  -- prepare working qubits
  let wqs = force qinit0Many @ n+2 in
  let wqs = (force mapHadamard @ n+2 @ 0) wqs in
  -- prepare ancilla
  let a = force qinit1 in
  let a = (force hadamard @ 0) a in
```

```
32  -- iterate Grover's algorithm
33  let iteration = lift forall step. \((wqs, a), _) :: ((List[_<n+2] Qubit{1+step*(od+4+2*(n
        +2))}, Qubit{1+step*(od+4+2*(n+2))}), ()).
34    (force groverIteration @ n @ 1+step*(od+4+2*(n+2)) @ os @ od) oracle wqs a
35  in let (wqs, a) = fold(iteration, (wqs, a), force range @r) in
36  let _ = (force qdiscard @ 1 + r * (od+4+2*(n+2))) a in
37  (force mapMeasure @ n+2 @ 1 + r * (od+4+2*(n+2))) wqs
```

Program 5.5: PQ implementation of the Grover's Algorithm, annotated with width and depth signatures.

This setting allows us to analyze the scaling of Grover's algorithm in terms of both resource consumption and iteration count and to verify whether QuRA bounds are only loose upper estimates or if they can still be precise in the face of complex control flow.

In the subsequent analysis, circuits are generated for different oracle sizes. Recall that the length of the register on which the oracle operates in $n + 2$, where n is the argument of the `conj` oracle.

Width Obtain the width annotation of the *Grover* function:

```
$ qura grover-width-depth.pq.pq -g width
[... more outputs, omitted for brevity ...]

grover :: ![0](forall[0, 0] r. forall[0, 0] n. forall[0, 0] os. forall[0, 0]
    od. (forall[0, 0] d. Circ[os]((List[_ < n + 2] Qubit, Qubit), (List[_ < n
    + 2] Qubit, Qubit))) -o[max(max(n + 3, os), 2 * (n + 2)), 0] List[_ < n +
    2] Bit)
```

As expected, the signature of the function tells us that the number of iterations of the algorithm does not influence the width of the circuit. The function is annotated for a generic oracle, so we can substitute the width of our oracle, $2(n + 2)$, for os in the signature of the arrow type:

$$width = \max(\max(n + 3, os), 2(n + 2)) = \max(\max(n + 3, 2n + 4), 2(n + 2)) = 2(n + 2).$$

We generate circuits for a different number of iterations, while also providing the correct depth and width of the oracle, which depend on n. The results are shown in Table 5.5, we can observe that the number of iterations does not influence the width of the generated circuits, as deduced from QuRA.

Depth Execute QuRA on the program to obtain the depth annotation of *grover*:

Width, $r = 1, 2, 3$

n	0	1	2	3	4	...	10	11	...	50	51	...
QuRA estimate: $2(n+2)$	4	6	8	10	12	...	24	26	...	104	106	...
Ground truth:	4	6	8	10	12	...	24	26	...	104	106	...

Table 5.5: Comparison of QuRA-inferred width and generated width of the circuits for Grover algorithm on a register of $n + 2$ qubits, with $r = 1, 2, 3$ iterations.

```
$ qura grover-gatecount-depth.pq -l depth --no-recycling
[... more outputs, omitted for brevity ...]

grover :: !(forall r. forall n. forall os. forall od. (forall d. Circ((List[_
    < n + 2] Qubit{d}, Qubit{d}), (List[_ < n + 2] Qubit{d + od}, Qubit{d +
    od}))) -o List[_ < n + 2] Bit{2 + r * (od + 4 + 2 * (n + 2))})
```

Recalling the depth of our oracle, $od = 2(n + 2)$, the overall depth of Grover's circuits becomes

$$depth = 2 + r(od + 4 + 2(n + 2)) = 2 + r(2(n + 2) + 4 + 2(n + 2)) = 2 + 4r(n + 3)$$

This time, we prepare tests that generates circuits that implement the algorithm with a different number of iterations, so we also have the r parameter, in addition to the size of the register n. The comparison between QuRA's predicted depth and the depth of the generated circuits is shown in Table 5.6.

DEPTH

QuRA estimate: $2 + 4r(n + 3)$

	n	0	1	2	3	4	...	10	11	...	50	51	...
$r = 1$	EST:	14	18	22	26	30	...	54	58	...	214	218	...
	GT:	12	16	20	24	28	...	52	56	...	212	216	...
$r = 2$	EST:	26	34	42	50	58	...	106	114	...	426	434	...
	GT:	22	30	38	46	54	...	102	110	...	422	430	...
$r = 3$	EST:	38	50	62	74	86	...	158	170	...	638	650	...
	GT:	32	44	56	68	80	...	152	164	...	632	644	...

Table 5.6: Comparison of QuRA-inferred depth and generated depth of the circuits for Grover algorithm on a register of $n + 2$ qubits, with $r = 1, 2, 3$ iterations. The value estimated by QuRA is on the **EST** row, the **GT** row represents the ground truth.

We can observe from the results that a small over-approximation is performed by QuRA, as the real depth of the generated circuits is always smaller than the predicted value by a

constant $2r$. QuRA always tries to match the type signature written in the PQ program, without trying to optimize it further. It is therefore possible to annotate with suboptimal resource values and obtain a loose upper bound. In this case, it is possible that one or more function annotation could be stricter.

Gatecount Lastly, let us analyze the *gatecount* of the function:

```
$ qura grover-gatecount-depth.pq -g gatecount
[... more outputs, omitted for brevity ...]

grover :: ![0](forall[0, 0] r. forall[0, 0] n. forall[0, 0] os. forall[0, 0]
    od. (forall[0, 0] d. Circ[os]((List[_ < n + 2] Qubit, Qubit), (List[_ < n
    + 2] Qubit, Qubit))) -o[1 + 2 * (n + 2) + r * (os + 4 * (n + 2) + 2 * (n +
    1) + 1), 0] List[_ < n + 2] Bit)
```

The given formula for the gatecount can be simplified and the size of the oracle, in this case the gatecount $os = 2(n+1)+1$, can be substituted in the following way:

$$gatecount = 1 + 2(n+2) + r(os + 4(n+2) + 2(n+1) + 1) = 5 + 2n + 2r(4n+7)$$

Results can be consulted in Table 5.7 , QuRA is able to correctly infer the exact number of gates required for the given PQ implementation of the Grover algorithm.

GATECOUNT

QuRA estimate: $5 + 2n + 2r(4n+7)$

	n	0	1	2	3	4	...	10	11	...	50	51	...
$r = 1$	EST:	19	29	39	49	59	...	119	129	...	519	529	...
	GT:	19	29	39	49	59	...	119	129	...	519	529	...
$r = 2$	EST:	33	51	69	87	105	...	213	231	...	933	951	...
	GT:	33	51	69	87	105	...	213	231	...	933	951	...
$r = 3$	EST:	47	73	99	125	151	...	307	333	...	1347	1373	...
	GT:	47	73	99	125	151	...	307	333	...	1347	1373	...

Table 5.7: Comparison of QuRA-inferred gatecount and generated gatecount of the circuits for Grover algorithm on a register of $n+2$ qubits, with $r = 1, 2, 3$ iterations. The value estimated by QuRA is on the **EST** row, the **GT** row represents the ground truth.

5.3.4 Quantum Adder

The next program proposed is taken from the QuRA GitHub repository, used to define the circuit family of quantum binary adders, `adder.pq`. At the time of writing this thesis, the

```
adder :: ![0](forall[0,0] n. (List[_<n+1] Qubit, List[_<n+1] Qubit) -o
    [3*(n+1) + 1, 0] (List[_<n+1] Qubit, List[_<n+2] Qubit))
adder n (a, b) =
  let (rest:((c,a),b),lsb) = (force adderFirstPhase @n) (a,b) in
  let (a,b) = force cnot a b in
  let (c,a,b) = force csum (c,a,b) in
  let (lastc, final) = (force adderSecondPhase @n) (c, rest) in
  let _ = (force qdiscard) lastc in
  let complete = (((force revpair @n) final) : (a,b)) in back in
  let (ares,b) = (force qunzip @ n+1) complete in
  let bres = b : lsb in
  ((force rev @ n+1) ares, (force rev @ n+2) bres)
```

Program 5.6: The main body of the binary adder written in PQ.

program is only annotated with the width metric, so we limit our comparison to the width of the generated circuits.

The main adder function, shown in Program 5.6, performs the sum between two registers of size $n + 1$, returning the first register and a new register of size $n + 2$. The inputs to the function are n and the two numbers to add, which we can trivially initialize to $|00\ldots0\rangle$, as bit and qubit initializations are operation with zero depth and gatecount.

Width

As a first step, we inspect the output produced by QuRA when computing the width of this circuit:

```
$ qura adder.pq -g width
[... more outputs, omitted for brevity ...]

adder :: ![0](forall[0, 0] n. (List[_ < n + 1] Qubit, List[_ < n + 1] Qubit)
    -o[3 * (n + 1) + 1, 0] (List[_ < n + 1] Qubit, List[_ < n + 2] Qubit))
```

We obtain a signature telling us that the produced circuit has size $3(n + 1) + 1$. We test this by generating many circuits while varying the size of the input registers. Recall from the start of the example, that n is not exactly the size of the input registers, but they have size $n + 1$. An example of a generated circuit, summing registers of two bits (so, $n = 1$) initialized to 0, is presented: the CRL obtained is shown in Figure 5.7 and its graphical representation is depicted in Figure 5.5.

```
$ qura adder-width.pq
[... more outputs, omitted for brevity ...]

> Operations:
QInit0 (*) -> q0;
```

```
QInit0 (*) -> q1;
QInit0 (*) -> q2;
QInit0 (*) -> q3;
QInit0 (*) -> q4;
QInit0 (*) -> q5;
QInit0 (*) -> q6;
Toffoli ((q0, q2, q5)) -> (q7, q8, q9);
CNot ((q7, q8)) -> (q10, q11);
Toffoli ((q6, q11, q9)) -> (q12, q13, q14);
Toffoli ((q1, q3, q4)) -> (q15, q16, q17);
CNot ((q15, q16)) -> (q18, q19);
Toffoli ((q14, q19, q17)) -> (q20, q21, q22);
CNot ((q18, q21)) -> (q23, q24);
CNot ((q23, q24)) -> (q25, q26);
CNot ((q20, q26)) -> (q27, q28);
Toffoli ((q10, q13, q27)) -> (q35, q36, q37);
CNot ((q35, q36)) -> (q38, q39);
Toffoli ((q12, q39, q37)) -> (q40, q29, q30);
CNot ((q38, q29)) -> (q31, q32);
CNot ((q40, q32)) -> (q33, q34);
QDiscard (q30) -> *;
QDiscard (q33) -> *.

While evaluating to:
> ((([]:q25):q31), ((([]:q22):q28):q34))
```

Program 5.7: Generated circuit from the `adder` function, on two 2-bit registers initialized to zero. The output of the function is the first register and the updated second register, plus one wire for the offset

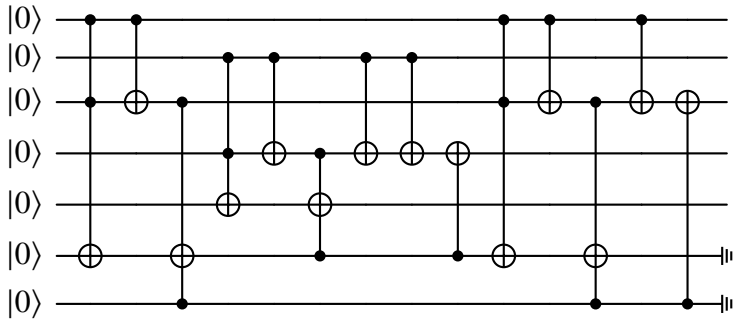


Figure 5.5: Graphical representation of the generated circuit from the `adder` function, on two 2-bit registers initialized to zero.

The metric results are shown in Table 5.8, in which `QuRA` estimates are exact.

Width												
n	0	1	2	3	4	...	10	11	...	50	51	...
QURA estimate: $3(n+1)+1$	4	7	10	13	16	...	34	37	...	154	157	...
Ground truth:	4	7	10	13	16	...	34	37	...	154	157	...

100	...	125	...	150	...	200	...	500	...	1000	...
304	...	379	...	454	...	604	...	1504	...	3004	...
304	...	379	...	454	...	604	...	1504	...	3004	...

Table 5.8: Comparison of QuRA-inferred widths and generated adder circuit widths, summing two registers of $n + 1$ qubits.

5.3.5 Shor's Algorithm

Shor's algorithm [38] is a landmark quantum algorithm for integer factorization, providing an exponential speedup over the best-known classical algorithms. Its circuit can be divided into two main components: the *modular exponentiation oracle*, which encodes the function $f(x) = a^x \bmod N$, and the *quantum Fourier transform* (QFT) that enables period finding. Together, these components allow the extraction of the factors of a composite integer N with high probability.

In our PQ implementation, the modular exponentiation is encapsulated in the oracle function vbe [46]. This function takes as input n and creates a boxed circuit that performs the necessary modular exponentiation operations to encode the function $f(x)$ on a $n+1$ qubit register. The width of this construction has been verified with QuRA[1] to be $7(n + 1) + 4$. Unfortunately, at the moment of writing this thesis the program is not annotated with other resources, other than width, so we limit our analysis to it.

The PQ implementation of Shor's algorithm [1] constructs a parametric family of circuits for an input register of size $n+1$. The algorithm applies the vbe oracle followed by an inverse QFT on the first register, repeats the required measurement and classical post-processing steps, and extracts the period of the function. An excerpt of the PQ program which we are testing can be found in Program 5.8.

Width

If we run QuRA on the program to analyze the width, we obtain the following outputs:

```
$ qura shor.pq -g width
[... more outputs, omitted for brevity ...]
```

```
--- Oracle VBE ---
vbe :: ![0](forall[0, 0] n. Circ[7*(n+1)+4]((List[_ < n + 1] Qubit, List[_ < n + 1] Qubit,
    List[_ < n + 1] Bit), (List[_ < n + 1] Qubit, List[_ < n + 1] Qubit, List[_ < n + 1]
    Bit)))
vbe n = box $ lift \(x, reg0, regMod) :: (List[_<n+1] Qubit, List[_<n+1] Qubit, List[_<n+1]
     Bit).
    -- init register 1, I initialize it here in order to keep the unitarity
    let reg1 = (force qinit1Many @n) in
    -- Modular exponentiation
    let (r, a, reg0, regMod) = (force modularExp @n) (x,reg1,reg0,regMod) in
    let _ = (force qdiscardMany @n) reg0 in
    (r, a, regMod)

--- Shor circuit ---
shor :: ![0](forall[0, 0] n. forall[0, 0] we. (Circ[we]((List[_ < n + 1] Qubit, List[_ < n
    + 1] Qubit, List[_ < n + 1] Bit), (List[_ < n + 1] Qubit, List[_ < n + 1] Qubit, List[_
     < n + 1] Bit)), List[_ < n + 1] Bit) -o[max(n + 1 + n + 1 + n + 1, we), 0] (List[i < n
     + 1] Bit, List[i < n + 1] Bit, List[_ < n + 1] Bit))
shor n we (oracle, regMod) =
  -- prepare n qubits in the state |0> then apply hadamard
  let x = (force mapHadamard @ n) (force qinit0Many @ n) in
  -- prepare n qubits in the state |0>
  let w = force qinit0Many @ n in
  -- oracle function
  let (x, a, regMod) = apply(oracle, (x, w, regMod)) in
  -- phase estimation throught inverse QFT
  let r = (force iqft @ n @ 0) x in
  -- measure the results
  ((force mapMeasure @ n) r, (force mapMeasure @ n) a, regMod)
  -- |x>|w> -> (r, a), where r is the period and a is the base (a^r mod N)
```

Program 5.8: An excerpt of the PQ implementation of Shor's factorization algorithm.

```
shor :: ![0](forall[0, 0] n. forall[0, 0] we. (Circ[we]((List[_ < n + 1]
    Qubit, List[_ < n + 1] Qubit, List[_ < n + 1] Bit), (List[_ < n + 1]
    Qubit, List[_ < n + 1] Qubit, List[_ < n + 1] Bit)), List[_ < n + 1] Bit)
    -o[max(n + 1
+ n + 1 + n + 1, we), 0] (List[i < n + 1] Bit, List[i < n + 1] Bit, List[_ <
    n + 1] Bit))
```

which tells us that the size of the generated circuits depends on the size of the oracle *we*, and it evaluates to $\max(n + 1 + n + 1 + n + 1, we)$. We know from the previous Section, that the *vbe* oracle has size $7 * (n + 1) + 4$, so we can substitute it inside the expression and obtain that:

$$width = max(n + 1 + n + 1 + n + 1, we) = \max(3(n + 1), 7(n + 1) + 4) = 7(n + 1) + 4.$$

Finally, we can generate circuits with different values of n and inspect their resulting widths:

The generated circuits have a width that is always 2 smaller than QuRA's estimate. This is first of all a good result, as we obtained circuits which are in practice smaller than the estimates, and the estimates are actually quite tight, since they are only off by a constant factor. The latter

Width												
n	1	2	3	4	5	...	10	11	...	50	51	...
QuRA expr.: $7(n+1)+4$	18	25	32	39	46	...	81	88	...	361	368	...
Generated Circuit:	16	23	30	37	44	...	79	86	...	359	366	...

Table 5.9: Comparison of QuRA-inferred width and generated width of the Shor's algorithms circuits, on an input register of size $n+1$.

might signify that the annotations in the PQ implementation of the algorithm may be slightly too conservative, and a deeper analysis could point out which signature is overestimating the width resources, if any.

Part II

Conversion of Produced Circuits to QASM

Having established the evaluation of QuRA programs and validated the precision of its resource estimates, we now turn to the problem of making the resulting circuits *executable* and *inspectable*. To this end, we extend QuRA with a compilation pipeline that exports evaluated configurations into OpenQASM 3.0, the standard intermediate representation for quantum circuits.

This translation bridges the gap between the abstractions of QuRA and practical quantum toolchains. By importing the generated OpenQASM 3.0 code into frameworks such as Qiskit, we can visualize the circuits, confirm their structural correctness, and prepare them for execution on simulators or hardware backends. We revisit the representative examples encountered previously, including the quantum Fourier transform and Grover's algorithm, to illustrate how the exported circuits align with their representation language.

The results of this part show that QuRA is not only a framework for theoretical resource analysis but also a practical toolchain component: programs can be evaluated, translated and executed in many quantum computing environments.

Chapter 6

From Circuits to QASM Programs

In the previous chapters, we focused on evaluating *Proto-Quipper-RA* programs and obtaining their corresponding circuit representations. However, these evaluated circuits remain an abstract, internal representation used by QuRA for resource analysis and verification. To make the resulting quantum programs executable on real quantum hardware or simulators, we need to translate these circuits into a widely supported, low-level representation.

For this purpose, we adopt OpenQASM 3.0 [15, 16] (**Quantum Assembly Language**), an intermediate language designed to describe quantum circuits in a hardware-agnostic yet execution-ready format. QASM is supported by several major quantum toolchains and backends, such as Qiskit [32] and IBM Quantum devices [33], making it an ideal target for code generation. Its syntax allows us to explicitly declare quantum and classical registers, initialize qubits and bits, and apply quantum operations using standard instructions.

This chapter introduces the translation pipeline that converts evaluated circuits into OpenQASM 3.0 programs. We first provide a brief overview of QASM, focusing on its syntax and how qubits, classical bits, and quantum operations are represented. After that, we present the mapping between quantum operations defined inside QuRA and their QASM counterparts, ensuring circuit equivalency during translation. We then describe the circuit simplification phase, where the CRL representation of the circuit is translated to be more similar to the actual QASM implementation. Finally, we discuss how simplified circuits can be used to compute relevant resource metrics such as circuit width, depth, and gatecount, enabling accurate comparisons between the abstract evaluation provided by QuRA and the hardware-ready implementation. New metrics oriented to QASM need to be developed to correctly infer the resources of the generated circuits, according to the OpenQASM 3.0 set of instructions.

By bridging the gap between abstract circuit representations and concrete QASM code, this chapter completes the transition from program evaluation to executable quantum circuits. This extension significantly enhances QuRA's capabilities, transforming it from a purely analytical tool into a practical frontend for generating runnable quantum programs.

6.1 OpenQASM 3.0: A Brief Overview

QASM is an imperative, hardware-agnostic assembly language designed as an intermediate representation for quantum computations. It serves as a bridge between high-level quantum programming languages and quantum hardware, making it a common compilation target and a de facto standard for describing quantum circuits [16].

The last version of QASM introduced many improvements to the language, but we will focus only on its basic constructs, such as initialization, gate application, and measurements, which will be presented briefly.

6.1.1 Qubit and Bit Initialization

Qubits, as well as registers of qubits, are declared explicitly:

```
1 qubit q;
2 qubit[5] qreg;
```

These declarations create *virtual qubits*—abstract references to quantum resources. A qubit is always initialized in the computational basis state $|0\rangle$, and QASM provides the `reset` instruction to be able to reset it to this value:

```
1 reset q;
```

Bits are initialized like almost any other language by simply assigning them to a value:

```
1 bit b=1;
```

6.1.2 Native and Derived Gates in OpenQASM 3

OpenQASM 3.0 provides a versatile framework for describing quantum circuits at a high level of abstraction. However, despite the broad set of gates commonly used in quantum algorithms,

the language specification deliberately defines only a minimal set of *native gates*. All other gates are introduced as *derived* operations, either hierarchically or by applying modifiers to existing gates.

Native Gates

The OpenQASM 3.0 specification defines exactly two primitive gates:

- **U**(θ, ϕ, λ)**:** A fully parameterized single-qubit unitary gate, capable of representing any arbitrary single-qubit operation up to a global phase. Its matrix form is given by:

$$U(\theta, \phi, \lambda) := \frac{1}{2}\begin{pmatrix} 1 + e^{i\theta} & -ie^{i\lambda}(1 - e^{i\theta}) \\ ie^{i\phi}(1 - e^{i\theta}) & e^{i(\phi+\lambda)}(1 + e^{i\theta}) \end{pmatrix}$$

- **gphase**(γ)**:** A global phase gate that applies a phase factor $e^{i\gamma}$ to the entire quantum state:

$$\text{gphase}(\gamma) := e^{i\gamma} I_m,$$

  where I_m denotes the identity matrix with size 2^m.

These two gates form the *complete set of native operations* in OpenQASM 3.0. Any other gate supported by the language is expressed in terms of these primitives or of other user-defined gates.

Derived Gates

Derived gates fall into two categories:

Hierarchically Defined Gates Users can define named gates using the `gate` keyword. For example, the Hadamard gate (`h`) can be expressed in terms of `U` and `gphase`:

```
gate h q {
    U(pi/2, 0, pi) q;
    gphase -pi/4;
}
```

Here, `h` is not primitive; instead, it expands into a sequence of native operations.

Modifier-Based Gates OpenQASM 3.0 also supports *gate modifiers* that derive new operations from existing ones without requiring explicit definitions. The most relevant modifiers include:

- `ctrl @ G`: produces a controlled version of gate G (e.g., `ctrl @ X` is equivalent to a CNot).
- `inv @ G`: produces the inverse of gate G.
- `pow(k) @ G`: applies gate G raised to a real-valued power k.

These modifiers are combinators: they do not introduce new primitives but rather compose and modify existing gates.

Version String and Included Files

An OpenQASM 3.0 program may begin with an optional version declaration on the first non-comment line, specifying the major and minor version numbers:

```
1 OPENQASM 3.0;
```

If the minor version number is omitted, it is assumed to be zero by default. The version declaration, if present, must appear only once and exclusively as the first non-comment line of the program.

QASM supports the inclusion of external files using the `include` directive:

```
1 // First non-comment is a version string
2 OPENQASM 3.0;
3 include "stdgates.qasm";
4 // Rest of QASM program
```

The contents of the included file are parsed as if they were directly inserted at the location of the statement. `stdgates.qasm`[1] is a particularly useful standard library, as it provides simple implementations of the most common quantum gates, relieving the programmer from having to manually define them using the primitive gates `U` and `gphase` and the modifiers presented in the previous section.

[1] See the file `stdgates.inc` at `https://github.com/Qiskit/qiskit/blob/main/qiskit/qasm/libs/stdgates.inc`.

This library includes a comprehensive set of one-qubit rotation gates, such as `rx`, `ry`, and `rz`, which are parameterized rotations around the respective Bloch-sphere axes. It additionally defines the commonly used Pauli gates `x`, `y`, and `z`, as well as the Hadamard gate `h` (introduced earlier), each expressed internally in terms of native primitives.

Beyond single-qubit operations, `stdgates.qasm` also provides definitions for widely used multi-qubit gates, including controlled operations like `cx` (CNot), `cy`, and `cz`, as well as more general controlled phase rotations, e.g. `cp` and `cu`. Furthermore, higher-level gates are provided, such as the Toffoli (`ccx`) and the swap gate (`swap`), again constructed using only the primitive gates and modifiers.

```
1 // Toffoli
2 gate ccx a, b, c { ctrl @ ctrl @ x a, b, c; }
```

Applying Quantum Gates

Quantum gates are applied using a simple and expressive syntax:

```
1 h q1;
2 cx q1, q2;
```

OpenQASM also supports *broadcasting*, allowing a gate to be applied to all aligned qubits in a register at once:

```
1 h qreg; // Applies Hadamard to all qubits in qreg
```

This mechanism simplifies the expression of symmetric or repetitive operations.

Classically controlled gates do not exist, as the classical `if` statement checks dynamically the value of a bit.

Measuring a Qubit

Measuring a qubit collapses its quantum state into a classical bit value. This is done using the `measure` instruction, which maps the result of a quantum measurement onto a classical register:

```
1 qubit q;
2 bit c;
```

```
3 h q; // Put the qubit in superposition
4 c = measure q;
```

In this example, the qubit `q` is first placed in a superposition by applying an Hadamard gate. The `measure` operation then collapses its state, storing the result in the classical bit `c`. The outcome will be `0` or `1` with equal probability.

6.2 From Quantum Operations to QASM Instructions

In the previous Section, we briefly introduced the basic syntax of QASMand the gates introduced in the library `stdgates.qasm`, which will be enough to cover all the operations defined inside QuRA. In this Section, we will define how to convert the quantum operations into valid QASM instructions.

The conversions between the individual elements of the QuRA CRL and the corresponding OpenQASM `3.0` instructions are shown in Table 6.1. The most important aspect of the QASM notation is that whenever you apply an instruction to a qubit or a bit, the result will still be the same qubit, but transformed. This is quite different from how quantum operations behave in CRL, consuming the input labels and creating new output labels. Resource annotations are also completely ignored, as there is no such counterpart in QASM. In a later section, we will see how the circuit needs to be modified in order to correctly protract the names of the wires through the operations.

When we first introduced OpenQASM `3.0`, we explained that qubits are always initialized to the $|0\rangle$ state, but we are often interested in starting from state $|1\rangle$. To do so, we have to negate the qubit once after initialization. Here, a large discrepancy with QuRA occurs, as initializations were always considered to have depth and gatecount of 0. Of course, applying a X gate raises both quantities to 1. Later, we will handle this gracefully, introducing new metrics specifically designed for the QASM backend.

In OpenQASM `3.0` there is no such concept as *qubit discarding*, but the `reset` instruction is used instead. Their behavior can be similar, as both remove qubits from the superposition, but QASM allows us to put the qubit back to $|0\rangle$ and use it later. This design choice allows us to *recycle* qubits, that is, using previously discarded wires for the new qubits instead of using a fresh wire. Potentially, being able to reuse existing wires can noticeably reduce the width of circuits. In order to be able to control this, we introduced a flag to QuRA: `--no-recycling`,

QuRA Boxed Circuit	OpenQASM 3.0 Instruction
$(*, \mathsf{QInit}_0, q)$	`qubit q;`
$(*, \mathsf{QInit}_1, q)$	`qubit q; x q;`
$(q, \mathsf{QDiscard}, *)$	`reset q;`
(q, Meas, b)	`b = measure q;`
$(*, \mathsf{CInit}_n, b)$	`bit[1] b = "n";`
$(b, \mathsf{CDiscard}, *)$	`// -`
(q_1, H, q_2)	`h q1;`
(q_1, X, q_2)	`x q1;`
(q_1, Y, q_2)	`y q1;`
(q_1, Z, q_2)	`z q1;`
(q_1, T, q_2)	`t q1;`
$(q_1, \mathsf{R}\, n, q_2)$	`rz(pi/k) q1;` with $k = 2^{n-1}$
$(q_1, \mathsf{Rinv}\, n, q_2)$	`rz(-pi/k) q1;` with $k = 2^{n-1}$
$((q_1, q_2), \mathsf{CNot}, (q_3, q_4))$	`cx q1, q2;`
$((q_1, q_2), \mathsf{CZ}, (q_3, q_4))$	`cz q1, q2;`
$((q_1, q_2), \mathsf{CR}\, n, (q_3, q_4))$	`crz(pi/k) q1, q2;` with $k = 2^{n-1}$
$((q_1, q_2), \mathsf{CRinv}\, n, (q_3, q_4))$	`crz(-pi/k) q1, q2;` with $k = 2^{n-1}$
$((b_1, q_1), \mathsf{CCNot}, (b_2, q_2))$	`cx b1, q1;`
$((b_1, q_1), \mathsf{CCZ}, (b_2, q_2))$	`cz b1, q1;`
$((q_1, q_2, q_3), \mathsf{Toffoli}, (q_4, q_5, q_6))$	`ccx q1, q2, q3;`

Table 6.1: Quantum operations conversion table between QuRA and OpenQASM 3.0

allowing us to decide whether to initialize new logical qubits on previously discarded wires or to always initialize them at the start of the circuit (at depth 0).

The classically-controlled gates in CRL are translated into quantum-controlled operations, since the behavior of classical controls can be carried by a qubit whose value is classical. Of course, this simplification does not work if the classical control is not the result of a measured qubit, but is a simple classical wire. In that case, a wire of type qubit, initialized to the required classical value of the bit, is used instead.

No QASM operation is defined for discarding bits, as they are not interfering with the circuit. Lastly, the rotations defined in QuRA are rotations around the z axis, which corresponds to `rz` in QASM. Furthermore, there is no distinction between the R_n gate and the Rinv_n gate, given that $\mathsf{R}_{-n} = \mathsf{Rinv}_n$, and we can therefore use `rz` for both operations.

6.3 Circuit Simplification

Inspecting the conversion table between QuRA and OpenQASM 3.0 from Table 6.1, we find that QASM instructions do not produce output labels, but rather act in place on the input variables. In this section, our aim is to provide enough motivation as to why the CRL circuit representation needs to be adapted to be more similar to the QASM behavior. Then, we present the proposed solution to the problem that simplifies the circuit labeling so that it becomes easier to convert the operations into QASM instructions.

To better understand the need for a strategy that unifies the names of the circuit, consider the following example, presenting an excerpt of the operations produced by QuRA and a naive conversion to QASM:

```
> Operations:
QInit0 (*) -> q0;
QInit1 (*) -> q1;
Hadamard (q0) -> q2;
CNot ((q2, q1)) -> (q3, q4);
```

```
1 qubit q0;
2 qubit q1;
3 x q1;
4 h q0;
5 cx q2, q1;
```

Figure 6.1: Naive conversion of a circuit generated by QuRA into OpenQASM 3.0.

This is not a valid OpenQASM 3.0 program, as the variable q_2 is undefined. The problem is caused by the fact that CRL re-binds outputs, but QASM does not. For example, the Hadamard gate acting on q_0 produces the label q_2, and the CNot tries to act on this *new* wire.

6.3.1 Label Renaming

The circuit in the previous example, Figure 6.1, could easily be converted if the output of each operations produces a wire bundle with the same names as the inputs. To achieve this new labeling, it is enough to step through the operations, create a renaming from the output labels to the input labels and propagate it to the rest of the operations until the last operation. The only exception is with measurements, as QASM explicitly requires that a bit is assigned to the measurement result, so we cannot keep the qubit in the output. In that case, the output bit is kept, but the renaming is propagated, since we now use the collapsed qubit instead of the classical bit. For this reason, we also convert classically controlled gates into quantum gates in order to preserve a typed circuit.

Unfortunately, this simple solution only works if qubit recycling is not involved. In fact, a new qubit initialization should consider previously discarded qubits and should change the name of the output wire to a name not in use anymore. This is done by being more careful when handling operations such as discards and initializations. A set of names is used to track the currently discarded names, and is used whenever a new variable needs initialization. Once the name has been selected, either from the discarded set or the actual name assigned by QuRA, the renaming is performed as before. If chosen from the discarded set, the label with the lesser depth is selected.

To demonstrate the differences, suppose the evaluation of a simple program being:

```
> Label Context: q0:Qubit, q1:Qubit
> Operations:
QInit0 (*) -> q0;
QDiscard (q0) -> *;
QInit1 (*) -> q1;
```

The simplified circuits obtained with the recycling strategy and without it are shown in Figure 6.2: we can notice that the second qubit is initialized on the previously existing wire, instead of using a fresh qubit.

```
> Simplified Circuit:
> Label Context: q0:Qubit
> Operations:
QInit0 (*) -> q0;
QDiscard (q0) -> *;
QInit1 (*) -> q0;
```

(a) Qubit recycling.

```
> Simplified Circuit:
> Label Context: q0:Qubit, q1:Qubit
> Operations:
QInit0 (*) -> q0;
QDiscard (q0) -> *;
QInit1 (*) -> q1;
```

(b) No recycling.

Figure 6.2: Comparison between the simplified circuits generated using qubit recycling (a) and without the recycling option (b).

6.3.2 Metric Computation of the Simplified Circuits

Once the evaluated configuration has been reduced to its *simplified circuit* form, the computation of the converted circuit resource metrics becomes natural and efficient. The simplified representation preserves all essential information about the requirements of the quantum computation, as it only renames the wires to produce consistent input and output labels of the operations, and to specialize the circuit whenever qubit recycling is involved.

This compact structure allows us to directly compute the most relevant resource metrics:

- **Gate count**: obtained simply by counting the number of operations in the sequence, considering that QInit_1 now requires one gate.

- **Circuit depth**: computed by tracking the maximum number of operations applied sequentially on any given qubit, considering that the QInit_1 operation now has a depth of one.

- **Width**: deduced from the number of names that appear in the label context of the circuit.

This makes the representation suitable not only for conversion into `QASM`, but also as an efficient basis for accurate resource analysis. Note that the simplified circuit is already specialized for recycling, so the metric computation remains the same.

6.3.3 Introducing an Updated Set of Metrics for `QuRA`

The semantics of `OpenQASM 3.0` introduce certain behaviors that differ from those assumed in `QuRA`, particularly with respect to the initialization and measurement of qubits. For instance, initializing a qubit to the state $|1\rangle$ should contribute to both the gate count and the circuit depth, each by one. Similarly, measurements in `OpenQASM 3.0` produce new classical wires rather than changing the type of the existing qubit wires, thus increasing the width of the circuit. These differences require the introduction of `QASM`-specific metrics to allow a meaningful comparison between the inferred resources in `QuRA` and the concrete values obtained from the generated circuits.

Extending `QuRA` with additional metrics is straightforward, as documented in the official documentation [8]. In our work, we added two global metrics, *qasmwidth* and *qasmgatecount*, accessible via `-g [qasmwidth|qasmgatecount]`, as well as a local metric, *qasmdepth*, accessible via `-l qasmdepth`. These behave analogously to their generic counterparts, with the key modifications that QInit_1 contributes a cost of one to both gate count and depth, and measurements introduce an additional wire to the circuit, increasing the width.

By aligning the metrics in this way, we obtain a consistent framework that allows us to quantitatively validate the correctness of the generated circuits and assess the precision of `QuRA`'s resource estimations against the concrete `OpenQASM 3.0` outputs.

6.4 Converting the Simplified Circuit to OpenQASM 3.0

In previous sections, we defined a conversion between QuRA quantum operations and QASM instructions and a simplified circuit representation that can be used as a closer intermediate representation to QASM. Most of the required elements are already present in the simplified circuit, which already accounts for the case of qubit recycling. The only thing that we are missing is a subtle attention towards qubit and bit initializations.

Whenever an instruction acts on a variable, that variable needs to be already initialized, otherwise an error occurs. In addition to this, we cannot initialize a variable more than once. In simplified circuits, both scenarios might occur: storing the result of a measurement requires that the bit already exists, which could not be the case; recycling an existing wire requires a new initialization, but that is not correct since the variable already exists. In order to produce valid OpenQASM 3.0 code, during the conversion of the operations, we also need to prepare a set containing all the initialized variables, to avoid double initializations.

Thanks to the circuit simplification step, we obtain a streamlined representation of the program that significantly reduces the complexity of the conversion process. In particular, the simplified circuit is *directly translatable* to OpenQASM 3.0: qubit allocations, measurements, and gate applications are presented in a clean, ordered sequence acting on consistent wires. As a result, the conversion can be carried out in just two steps:

1. Prepend the version string, `OPENQASM 3.0;`, and include the `stdgates.qasm` library.

2. Traverse the simplified circuit and translate each operation according to Table 6.1. Whenever an operation involves the creation of new labels, for example, through qubit initializations or measurements, we check whether the corresponding variable is already defined; if not, it is properly initialized and added to the set of existing variables.

This reduction ensures that the simplified circuit acts as a *ready-to-convert* intermediate representation, that is also well suited to compute the resource analysis of the circuit, as presented in the previous Section.

6.4.1 Inspecting the Generated QASM Programs

To validate the translation from evaluated configurations to QASM, we examine several representative circuits. By importing the generated QASM code into Qiskit and visualizing the

resulting circuits, we can confirm that their structure and gate sequence correspond precisely to the original CRL representations.

The Teleportation Protocol

The teleportation circuit has been a running example in this thesis, used to verify the correctness of the various steps of the interpretation.

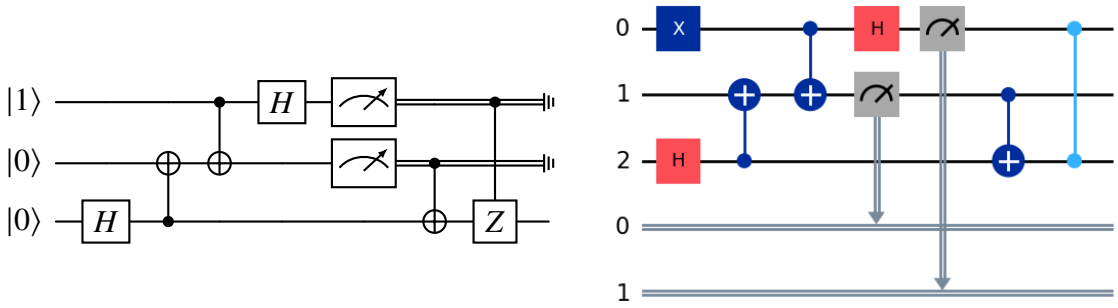


Figure 6.3: Comparison between the circuit to teleport the value 1 and the obtained QASM circuit, drawn using Qiskit.

We can observe that the two circuits present the same structure, but they differ slightly in some details:

- To perform the initialization of the first Qubit to $|1\rangle$, an X gate is applied.
- The classically controlled gates are now controlled by quantum wires after measuring them in the two wires below.
- The controlled Z gate is represented by the last vertical line with a dot at each end, instead of the classical representation of an X gate on the target wire.

Quantum Fourier Transform

The quantum Fourier transform (QFT) provides an excellent benchmark to validate the correctness of generated parametric circuits. Figure 6.4 presents again the canonical structure of the QFT on a n-qubit register: the most significant qubit undergoes a Hadamard gate, followed by a sequence of controlled phase rotations R_k with the remaining qubits. Afterward, the QFT is applied recursively to the other $n - 1$ qubits.

To illustrate the recursive structure of the generated QFT circuits, we present the cases for $n = 1, 2, 3$, and 4 qubits in Figure 6.5. As expected, each increment in n introduces

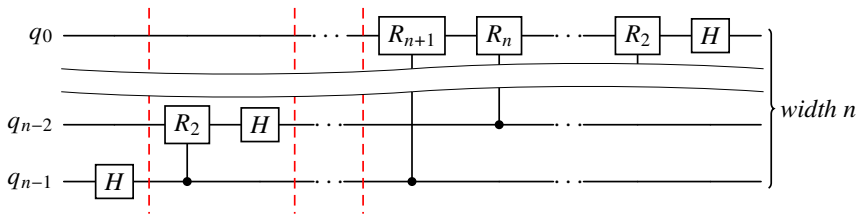
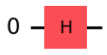
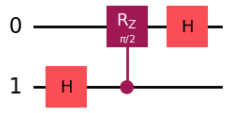


Figure 6.4: The generic QFT circuit on an n-qubit register.

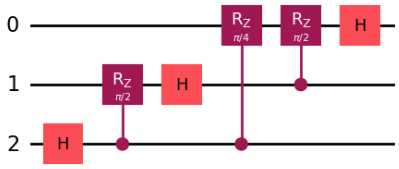
one additional Hadamard gate and a sequence of controlled rotations R_k applied to the first qubit. This progression confirms that the generated circuits follow the canonical parametric definition of the QFT.



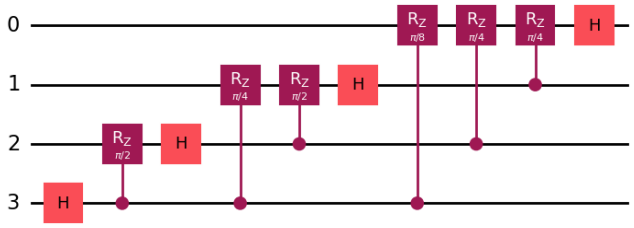
(a) $n = 1$ qubit



(b) $n = 2$ qubits



(c) $n = 3$ qubits



(d) $n = 4$ qubits

Figure 6.5: QFT circuits generated by QuRA for $n = 1, 2, 3, 4$ qubits, visualized using Qiskit. The recursive structure is clearly visible: each additional qubit introduces one Hadamard gate and a growing sequence of controlled rotations R_k, as expected.

Recall that each controlled phase rotation R_k is defined as:

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}, \quad \theta = \frac{\pi}{2^{k-1}}.$$

Since the PQ implementation of the QFT is defined using folds, this visual verification for small n directly supports the *correctness of the entire parametric family*: for $n = 1$, the generated circuit consists solely of a single Hadamard gate, matching the canonical definition; assuming the $(n-1)$-qubit circuit is correctly generated, the n-qubit circuit is obtained by appending the recursive $(n-1)$-qubit QFT, the appropriate sequence of controlled rotations R_k, and one Hadamard gate. This recursive structure guarantees that the correctness observed

for small values of n extends to arbitrary circuit sizes.

Grover's Algorithm

In Chapter 5.3.3, we compared the resources of Grover's algorithm and we presented a graphical representation of the algorithm elements and the overall circuit. Executing `QuRA` on `grover.pq` generated to perform 2 iterations on a register of three qubits gives us the circuit of Figure 6.6, displayed using `Qiskit`.

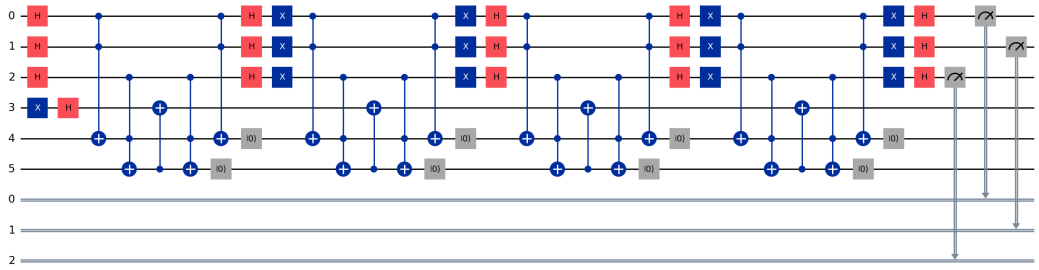


Figure 6.6: Grover's algorithm circuit generated by `QuRA` with two iterations, using the *nconj* oracle on a 3 qubits register, visualized using `Qiskit`.

Similarly to the teleportation protocol, the `QASM` conversion uses many new bit wires to store the classical result of measuring the quantum registers. Other than that, we can clearly see that it matches the circuit representation of Figure 5.4.

By default `QuRA` tries to reuse qubits, and we can see that it is the case, since all the ancillas needed for the multi-controlled operations lays on the same wires. If we were to execute the program with the `--no-recycling` tag, we would obtain a circuit that always initializes new ancillary qubits for the MCNots. This behavior can be observed in Figure 6.7, where the *grover* function is executed using the same parameters.

Quantum Adder

The last conversion example that we provide is the previously analyzed adder function. Given the classical nature of the circuit, we can also verify that it correctly computes the addition of two binary numbers through simulations.

The circuit generated for the function that sums two registers of 2 bits, the first initialized to `01` and the second to `11`, is shown in Figure 6.8. Note that the structure of the obtained circuit is really similar to the CRL obtained in Section 5.3.4 and showed in Figure 5.5, with the exception that the first three operations have been moved two steps to the right.

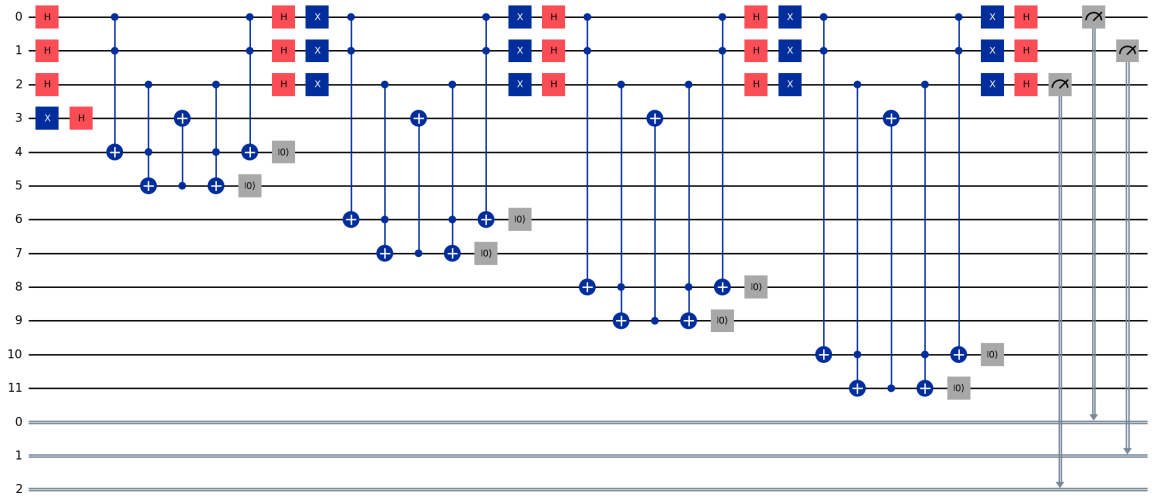


Figure 6.7: Grover's algorithm circuit generated by QuRA with two iterations, using the *nconj* oracle on a 3 qubits register without enabling wire recycling, visualized using Qiskit.

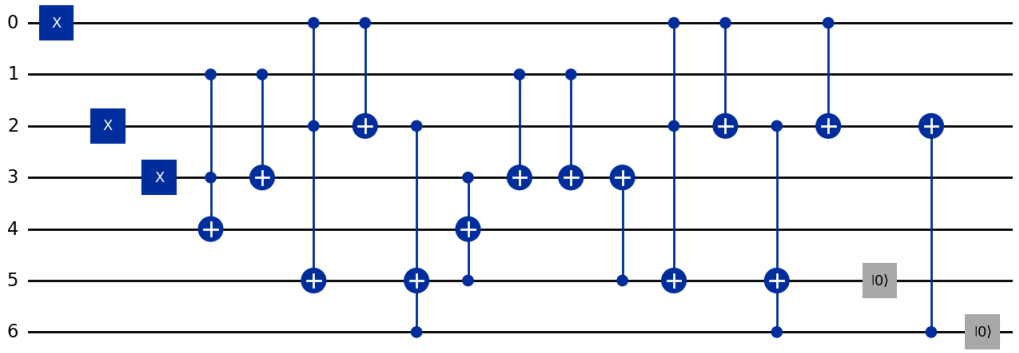


Figure 6.8: Adder circuit generated by QuRA for two 2-qubit registers, visualized using Qiskit.

As we can see from the figure, the Pauli-X gates are used to initialize the registers to the desired values. If we run a simulation, we expect that the output register holds the value 100, which corresponds to $01_2 + 11_2 = 100_2$. The simulation is run through Qiskit and the results are shown in Figure 6.9, confirming our expectations.

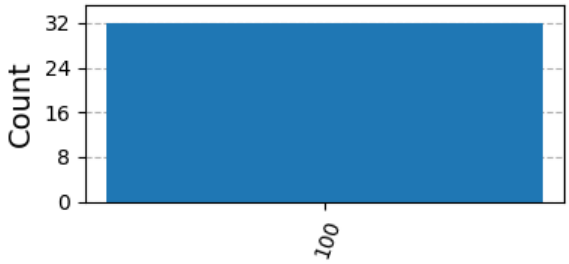


Figure 6.9: Simulation results of the adder circuit generated by QuRA for two 2-qubit registers, run through Qiskit.

Chapter 7

Conclusions

7.1 Contributions of this Thesis

The contributions of this thesis are threefold:

1. **Formal Evaluation Engine for *Proto-Quipper-RA*:** We formalized the reduction of `PQ` programs into a Circuit Representation Language (CRL) and implemented an interpreter based on big-step semantics. This enables explicit circuit construction from well-typed programs and provides a uniform basis for both dynamic resource accounting and code generation.

2. **Validation of Resource Estimates:** We validated `QuRA`'s inferred resource bounds against metrics measured directly from the generated circuits. For simple, non-parametric circuits (e.g., the Teleportation Protocol), the inferred and measured values coincided exactly. For scalable families (QFT, Grover, Adder), the symbolic formulas inferred by `QuRA` closely matched the empirical growth across input sizes. In more complex cases (e.g., Shor's algorithm), bounds remained sound but slightly conservative, with constant to linear overapproximatinos, highlighting opportunities for a refinement on the analysis.

3. **Conversion to Executable `OpenQASM 3.0`:** We developed a translation pipeline from evaluated circuits to `OpenQASM 3.0`, allowing visualization and execution through frameworks such as `Qiskit`. To support this, we introduced `QASM`-specific metrics (`qasmwidth`, `qasmgatecount`, `qasmdepth`) that accurately capture the semantics of initialization and measurement. This ensures meaningful comparisons between

inferred bounds and concrete circuit costs.

Significance and Future Directions

This work substantially extends QuRA's scope, turning it into a practical toolchain component that spans the entire lifecycle of quantum program development: from type-driven analysis to concrete circuit execution. By providing a modular evaluation and translation pipeline, it bridges the gap between theoretical resource reasoning and practical backend compatibility. Furthermore, our contribution made it possible to empirically validate QuRA's static resource estimates against the metrics of the generated circuits, strengthening confidence in its soundness and practical applicability.

Several directions for future work are particularly promising:

- **Syntactic Sugar and New Language Constructs:** Introduce higher-level constructs in PQ that simplify the expression of common quantum subroutines (e.g., modular exponentiation, controlled operations). These constructs could be systematically desugared into existing primitives, making *Proto-Quipper-RA* programs more concise and accessible.

- **Refinement of Resource Estimates:** Investigate ways to narrow the gap between inferred and concrete metrics, for example by refining typing rules for complex algorithms or incorporating backend-specific cost models.

- **Expanded Backend Compatibility:** Generalize the translation layer to target multiple intermediate representations beyond QASM, such as MLIR (Multi-Level Intermediate Representation [27]) or other quantum intermediate representations, enabling broader adoption across different quantum toolchains.

In conclusion, this thesis validates and extends the QuRA tool advancing the state-of-the-art in preparing quantum algorithms for the Noisy Intermediate-Scale Quantum era and beyond.

Appendix A

PQ Programs

A.1 Quantum Teleportation Protocol

```
-- teleportation.pq annotated for gatecount and depth
--- put q and p into the entangled |+> state
bell :: ![0](forall[0,0] dq. forall[0,0] dp.
  (Qubit{dq}, Qubit{dp}) -o[2,0] (Qubit{max(dq+1, dp) + 1}, Qubit{max(dq+1, dp) + 1}))
bell dq dp (q, p) =
    let q = (force hadamard @dq) q in
    let (q,p) = (force cnot @dq+1 @dp) q p in
    (q,p)
--- Alice's part of the teleportation protocol
alice :: ![0](forall[0,0] dp. forall[0,0] dr.
  (Qubit{dp}, Qubit{dr}) -o[4,0] (Bit{max(dp, dr) + 2}, Bit{max(dp, dr) + 3}))
alice dp dr (p, r) =
    let (r,p) = (force cnot @dr @dp) r p in
    let r = (force hadamard @ max(dp, dr) + 1) r in
    let c = (force meas @ max(dp, dr) + 1) p in
    let d = (force meas @ max(dp, dr) + 2) r in
    (c,d)
--- Bob's part of the teleportation protocol
bob :: ![0](forall[0,0] dq. forall[0,0] dc. forall[0,0] dd.
  (Qubit{dq}, Bit{dc}, Bit{dd}) -o[2,0] Qubit{max(dd, max(dc, dq) + 1) + 1})
bob dq dc dd (q, c, d) =
    let (c,q) = (force ccnot @dc @dq) c q in
    let (d,q) = (force ccz @dd @ max(dc, dq) + 1) d q in
    let _ = (force cdiscard @ max(dc, dq) + 1) c  in
    let _ = (force cdiscard @ max(dd, max(dc, dq) + 1) + 1) d  in
    q
--- teleport the state of qubit r (at depth dr) into qubit q
teleport :: ![0](forall[0,0] dr. Qubit{dr} -o[8,0] Qubit{dr+6})
teleport dr r =
    let q = force qinit0 in
    let p = force qinit0 in
    let (q,p) = (force bell @0 @0) (q,p) in
    let (c,d) = (force alice @2 @dr) (p,r) in
    (force bob @2 @ max(2, dr) + 2 @ max(2, dr) + 3)  (q,c,d)
```

Program A.1: PQ implementation of the Quantum Teleportation Protocol, annotated with gatecount and depth signatures.

A.2 Quantum Fourier Transform

```
-- qft-n4.pq
--- HELPER FUNCTIONS ---
-- invert the list of intermediate qubits at iteration iter
qrev :: !(forall iter. List[i<iter] Qubit -o List[i<iter] Qubit)
qrev iter reg =
    let revStep = lift forall step.
      \(rev, q) :: (List[i<step] Qubit, Qubit).
        rev:q in
    fold(revStep, [], reg)
-- apply the controlled rotation gate to the target qubit trg at iteration iter
rotate :: !(forall iter. !(forall step.
  ((List[i < step] Qubit, Qubit), Qubit) -o (List[i < step + 1] Qubit, Qubit)))
rotate iter = lift forall step.
  \((ctrls, trg), ctrl)::((List[i<step] Qubit, Qubit), Qubit).
    let (ctrl, trg) = (force cr @(iter+1-step) @0 @0) ctrl trg in
    (ctrls:ctrl, trg)
--- QUANTUM FOURIER TRANSFORM ---
qftIteration :: !(forall iter. (List[i<iter] Qubit, Qubit) -o List[i<iter+1] Qubit)
qftIteration iter (ctrls, trg) =
    let revctrls = (force qrev @iter) ctrls in
    let (ctrls, trg) = fold(force rotate @iter, ([], trg), revctrls) in
    let trg = (force hadamard @0) trg in
    ctrls:trg
-- apply the Quantum Fourier Transform to n qubits at depth d
qft :: !(forall n. List[i<n] Qubit -o List[i<n] Qubit)
qft n reg = fold(qftIteration, [], reg)
--- MAIN ---
main =
  let w = force qinit1 in
  let x = force qinit1 in
  let y = force qinit1 in
  let z = force qinit1 in
  (force qft @4) [w,x,y,z]
```

Program A.2: PQ implementation of the QFT on a 4-qubit register.

A.3 Multi-Controlled Not

```
--- MCNOT
-- actually, is (n+2)cnot, so n = m-2
mcnot :: ![0](forall[0,0] n. forall[0,0] dc. forall[0,0] dt. List[_<n+2] Qubit{dc} -o[0,0] Qubit{dt} -o[2*(n+2),n+2] (List[_<n+2] Qubit{max(dc+n+2,dt+1)+n+2}, Qubit{max(1+dc+n+1,1+dt)}))
mcnot n dc dt ctrls trgt =
  let restControls:q2:q1 = ctrls in
  let a1 = force qinit0 in -- first ancilla
  --- create the first intermediate value ---
  let (q1,q2,a1) = (force toffoli @dc@dc@0) q1 q2 a1 in
  --- forward pass ---
  let toffoliStep = lift forall step. \((cs, as, a), c) :: ((List[_<step] Qubit{dc+step+2}, List[_<step] Qubit{dc+step+2}, Qubit{dc+step+1}), Qubit{dc}).
    let new = force qinit0 in
    let (a,c,new) = (force toffoli @max(1,dc+1)+step @dc @0) a c new in
    (cs:c, as:a, new)
  in let (restControls, ancillas, lastAncilla) = fold(toffoliStep, ([],[],a1), restControls) in
  --- control the target ---
  -- let ancillas:lastAncilla = ancillas in -- extract last ancilla
  let (lastAncilla, trgt) = (force cnot @dc+n+1 @dt) lastAncilla trgt in -- apply cnot to the target
  --- backwards pass ---
  let ca = (force qzip @n @dc+n+2) ((force rev @n @dc+n+2) restControls, (force rev @n @dc+n+2)ancillas) in -- zip controls and ancillas
  let bwToffoliStep = lift forall step. \((cs,nextTarget),(c,a)) :: ((List[_<step] Qubit{max(dc+n+2,dt+1)+step+2}, Qubit{max(dc+n+2,dt+1)+step+1}),(Qubit{dc+n+2},Qubit{dc+n+2})).
      let (c,a,nextTarget) = (force toffoli @dc+n+2 @dc+n+2 @max(dc+n+2,dt+1)+step+1) c a nextTarget in
      let _ = (force qdiscard @max(dc+n+2,dt+1)+step+2) nextTarget in
      (cs:c, a)
  in let (restControls, a1) = fold(bwToffoliStep, ([],lastAncilla), ca) in
  let (q1,q2,a1) = (force toffoli @dc+1 @dc+1 @max(dc+n+2,dt+1)+n+1) q1 q2 a1 in
  let _ = (force qdiscard @max(dc+n+2,dt+1)+n+2) a1 in
  --- return the mcnot outputs ---
  (restControls:q2:q1, trgt) -- reorder the controls
```

Program A.3: PQ implementation of the multi-controlled Not function.

A.4 Grover's Algorithm

```
--- GROVER'S ALGORITHM annotated with gatecount and depth ---
-- the diffusion operator on n qubits with ancilla a (all qubits at depth d)
diffusion :: ![0](forall[0,0] n. forall[0,0] d. List[_<n+2] Qubit{d} -o[0,0] Qubit{d} -o
    [4*(n+2)+2*(n+1)+1, 0] (List[_<n+2] Qubit{d+4+2*(n+2)}, Qubit{d+4+n}))
diffusion n d reg a =
  let reg = (force mapHadamard @ n+2 @ d) reg in
  -- begin negatively controlled not
  let reg = (force mapQnot @ n+2 @ d+1) reg in
  let (reg,a) = (force mcnot @ n @ d+2 @ d) reg a in
  let reg = (force mapQnot @ n+2 @ d+2+2*(n+2)) reg in
  -- end negatively controlled not
  let reg = (force mapHadamard @ n+2 @ d+3+2*(n+2)) reg in
  (reg, a)
-- perform a single grover iteration on n qubits at depth d, using an ancilla a at depth d,
    and an oracle of size os and depth od
groverIteration :: ![0](forall[0,0] n. forall[0,0] d. forall[0,0] os. forall[0,0] od.
  (forall[0,0] d. Circ[os]((List[_<n+2] Qubit{d}, Qubit{d}), (List[_<n+2] Qubit{d + od},
      Qubit{d + od})))
  -o[0,0] List[_<n+2] Qubit{d} -o[0,0] Qubit{d}
  -o[os+4*(n+2)+2*(n+1)+1, 0] (List[_<n+2] Qubit{d+od+4+2*(n+2)}, Qubit{d+od+4+n}))
groverIteration n d os od oracle reg a =
  let (reg, a) = apply(oracle @ d, (reg, a)) in
  (force diffusion @ n @ d+od) reg a
-- run Grover's algorithm on an oracle of input size n, width os and depth od
grover :: ![0](forall[0, 0] r. forall[0, 0] n. forall[0, 0] os. forall[0, 0] od.
  (forall[0,0] d. Circ[os]((List[_<n+2] Qubit{d}, Qubit{d}), (List[_<n+2] Qubit{d+od},
      Qubit{d+od})))
  -o[n+2+1+r*(os+4*(n+2)+2*(n+1)+1)+n+2, 0] List[_<n+2] Bit{2 + r*(od+4+2*(n+2))})
grover r n os od oracle =
  -- prepare working qubits
  let wqs = force qinit0Many @ n+2 in
  let wqs = (force mapHadamard @ n+2 @ 0) wqs in
  -- prepare ancilla
  let a = force qinit1 in
  let a = (force hadamard @ 0) a in
  -- iterate Grover's algorithm
  let iteration = lift forall step. \((wqs, a), _) :: ((List[_<n+2] Qubit{1+step*(od+4+2*(n
      +2))}, Qubit{1+step*(od+4+2*(n+2))}), ()).
    (force groverIteration @ n @ 1+step*(od+4+2*(n+2)) @ os @ od) oracle wqs a
  in let (wqs, a) = fold(iteration, (wqs, a), force range @r) in
  let _ = (force qdiscard @ 1 + r * (od+4+2*(n+2))) a in
  (force mapMeasure @ n+2 @ 1 + r * (od+4+2*(n+2))) wqs
```

Program A.4: PQ implementation of the Grover's Algorithm, annotated with gatecount and depth signatures.

Bibliography

[1] O. Ayache. *Resource Estimation of Quantum Programs through Type Inference: QuRA and Shor's Algorithm*. Bachelor's Thesis, Department of Computer Science and Engineering, University of Bologna, Bologna, Italy, July 2025. Relator: Prof. Ugo Dal Lago; Co-Relator: Dott. Andrea Colledan.

[2] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. Cvc5: a versatile and industrial-strength smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[3] C. Barrett and L. de Moura. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(7):107–115, 2011. DOI: 10.1145/1995376.1995394.

[4] P. Benioff. The computer as a physical system: a microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22(5):563–591, 1980.

[5] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Physical Review Letters*, 70(13):1895–1899, 1993. DOI: 10.1103/PhysRevLett.70.1895.

[6] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, F. Felser, T. Killoran, and N. Killoran. Pennylane: automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018. URL: https://pennylane.ai.

[7] F. Bloch. Nuclear induction. *Physical Review*, 70(7-8):460–474, 1946. DOI: 10.1103/PhysRev.70.460.

[8] A. Colledan. Extending qura. QuRA Documentation, Read the Docs, 2025. Retrieved September 2025, from `https://qura.readthedocs.io/en/latest/tool/extensions/`.

[9] A. Colledan. *Resource Verification of Quantum Circuit Description Languages*. PhD thesis, ALMA MATER STUDIORUM – University of Bologna, Ph.D. in Computer Science and Engineering, Bologna, Italy, 2024. Supervisor: Prof. Ugo Dal Lago; Ph.D. Coordinator: Prof. Ilaria Bartolini; Cycle XXXVII.

[10] A. Colledan and U. Dal Lago. Flexible type-based resource estimation in quantum circuit description languages. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2025)*, 2025.

[11] A. Colledan, U. Dal Lago, and N. Vazou. Circuit width estimation via effect typing and linear dependency. *ACM Transactions on Programming Languages and Systems*, May 2025.

[12] A. Colledan and U. D. Lago. Circuit width estimation via effect typing and linear dependency: Proto-quipper-r. *Programming Languages and Systems (ESOP 2024)*, 14649:3–30, 2024. DOI: `10.1007/978-3-031-57267-8_1`.

[13] H. Committee. *Haskell 2010 Language Report*. https://www.haskell.org/, 2010.

[14] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017. URL: `https://arxiv.org/abs/1707.03429`.

[15] A. W. Cross, S. Sheldon, L. S. Bishop, and J. M. Gambetta. Openqasm 3: a broader and deeper quantum assembly language. *Quantum Science and Technology*, 7(2):025005, 2022. DOI: `10.1088/2058-9565/ac5fc5`.

[16] Cross, Andrew W. and Bishop, Lev S. and Smolin, John A. and Gambetta, Jay M. Openqasm 3.0 specification. `https://openqasm.com/versions/3.0/index.html`, 2023. Accessed: YYYY-MM-DD.

[17] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985. DOI: `10.1098/rspa.1985.0070`.

[18] C. Developers. Cirq: a python framework for noisy intermediate scale quantum circuits. https://github.com/quantumlib/Cirq, 2020.

[19] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982. DOI: 10.1007/BF02650179.

[20] P. Fu, K. Kishida, and P. Selinger. Linear dependent type theory for quantum programming languages: extended abstract. In *Proc. of LICS 2020*, 2020.

[21] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. *ACM SIGPLAN Notices*, 48(6):333–342, 2013.

[22] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–342, 2013. DOI: 10.1145/2491956.2462177.

[23] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996. DOI: 10.1145/237814.237866.

[24] S. Grul and J. Doe. Challenges in classical simulation of quantum circuits. *Quantum Information Processing*, 19:123–145, 2020.

[25] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. DOI: 10.2307/1995087.

[26] R. S. Ingarden. *Quantum Information Theory*. PWN, Warsaw, 1976.

[27] C. Lattner, V. Adve, J. Aldrich, M. Baboulin, Y. Chen, F. Cordeiro, C. Cummins, R. Cytron, T. Grosser, R. Hadidi, and ... Mlir: a compiler infrastructure for the end of moore's law. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, pages 1–16, 2019. DOI: 10.1145/3314221.3314602.

[28] R. Milner. A theory of type polymorphism in programming. In *Proceedings of the Royal Society of Edinburgh, Section A*, volume B 73, pages 147–162, 1978.

[29] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 10th anniversary edition edition, 2010. ISBN: 978-1107002173.

[30] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5:4213, 2014. DOI: 10.1038/ncomms5213.

[31] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[32] I. Quantum. Qiskit: an open-source framework for quantum computing. https://qiskit.org, 2021.

[33] I. Research. Ibm quantum experience: a cloud-based quantum computing platform for education and research. *Bulletin of the American Physical Society*, 2017. https://quantum-computing.ibm.com.

[34] F. Rios and P. Selinger. A categorical model for a quantum circuit description language. In *Proc. of QPL 2017*, 2017.

[35] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. DOI: 10.1145/359340.359342.

[36] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In *Mathematics of Program Construction (MPC 2006)*, volume 4014 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2006. DOI: 10.1007/11783596_21.

[37] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[38] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Proceedings 35th Annual Symposium on Foundations of Computer Science*:124–134, 1994. DOI: 10.1109/SFCS.1994.365700.

[39] S. Sinha and P. Russer. Quantum computing algorithm for electromagnetic field simulation. *Quantum Information Processing*, 9(4):385–404, 2010. DOI: 10.1007/s11128-009-0133-x.

[40] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan. T|ket⟩: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 5(3):034001, 2020. DOI: 10.1088/2058-9565/ab8e92. URL: https://doi.org/10.1088/2058-9565/ab8e92.

[41] Smite-Meister. The bloch sphere, a geometrical representation of a two-level quantum system. Wikimedia Commons, January 2009. URL: https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg. Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0).

[42] D. S. Steiger, T. Häner, and M. Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018. DOI: 10.22331/q-2018-01-31-49. URL: https://doi.org/10.22331/q-2018-01-31-49.

[43] K. M. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. Q#: enabling scalable quantum computing and development with a high-level domain-specific language. *Quantum Science and Technology*, 3(2):020501, 2018. DOI: 10.1088/2058-9565/aaadf6. URL: https://doi.org/10.1088/2058-9565/aaadf6.

[44] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*. 2nd series, 42:230–265, 1936. DOI: 10.1112/plms/s2-42.1.230. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[45] G. Van Rossum and F. L. Drake. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[46] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, July 1996. ISSN: 1094-1622. DOI: 10.1103/PhysRevA.54.147. URL: http://dx.doi.org/10.1103/PhysRevA.54.147.

[47] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982. DOI: 10.1038/299802a0.

[48] L. Zhou and M. Li. Scalability challenges in classical simulation of quantum algorithms. *Physical Review A*, 102:012345, 2020.