Towards a formal compilation stack-frame in quantum computing

Agustín Borgna

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Vers une formalisation d'une chaîne de compilation pour un ordinateur quantique

THÈSE

présentée et soutenue publiquement le 13 Janvier 2022

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Agustín Pablo Borgna

Composition du jury

Président :	Bob Coecke	Quantinuum
Rapporteurs :	Bob Coecke	Quantinuum
	Michael Mislove	Tulane University
Examinateurs :	Natacha Portier	ENS Lyon
	Claudia Faggian	CNRS
Invité :	Miriam Backens	University of Birmingham
Encadrants :	Simon Perdrix	INRIA
	Benoît Valiron	CentraleSupélec, Université Paris-Saclay

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Laboratoire Méthodes Formelles — UMR 9023

Mis en page avec la classe thesul.

Remerciements

I would like to thank my advisors Benoît Valiron and Simon Perdrix for their support and guidance throughout my PhD, specially when the world prevented us from meeting in person for a big part of it.

Similarly, I would like to thank Bob Coecke and Mike Mislove for agreeing to review my manuscript, as well as the rest of my examiners: Natacha Portier, Claudia Faggian, and Miriam Backens.

Likewise, my thanks the whole team at QuaCS for the research discussions and the laughs, whether at the lab or in the pub. Special thanks to my academic brothers Kostia Chardonnet, Louis Lemonnier, and Dongho Lee.

Finally, I would like to thank my family for their transoceanic support and love, and my friends for making my stay in France so enjoyable.

Contents

Introduction **vii**

Chapter 1
Of Qubits and Quantum Machines **1**

1.1 Quantum computation 1
1.1.1 Hilbert spaces 1
1.1.2 Pure quantum states 2
1.1.3 Quantum operators 2
1.1.4 Quantum measurement 4
1.1.5 Mixed quantum systems 5
1.2 Quantum circuits 6
1.2.1 Single-qubit gates 6
1.2.2 Multi-qubit gates 7
1.2.3 Initialisation and measurement of qubits 8
1.2.4 Circuit interpretation 8
1.2.5 Hybrid quantum-classical circuits 9
1.2.6 Universal gate sets 10
1.3 Quantum lambda calculi 11
1.4 Circuit-description languages 12

Chapter 2
The ZX Calculus

2.1 The pure ZX calculus 15

2.2 Grounded ZX 18
2.3 Semantics of the ZX and $ZX_{\Downarrow}$ calculi 19
2.4 The Scalable ZX calculus 20
2.4.1 Flattening SZX diagrams 24
2.5 SZX diagram families and list instantiation 24

Chapter 3
Compiling high-level quantum programs into SZX diagrams

3.1 The λ_D calculus 30
3.2 Operational Semantics of the λ_D calculus 35
3.3 Encoding programs as diagram families 38
3.3.1 Parameter evaluation 38
3.3.2 Diagram encoding 44
3.4 Application example: QFT 53

Chapter 4
Optimizing hybrid quantum-classical circuits with the $ZX_{\Downarrow}$ calculus

4.1 Graph-like diagrams and focused gFlow 58
4.2 Translation of hybrid circuits 61
4.3 Graph-theoretical circuit optimization 65
4.4 Grounded ZX optimization 66
4.5 Ground-cut simplification 68
4.6 The Algorithm 70
4.7 Circuit extraction 72
4.8 Extraction examples 75
4.9 Implementation and benchmarks 76

Chapter 5
Classicality detection using $ZX_{\Downarrow}$ diagrams

5.1 The classicalisation problem 80
5.2 Labelled diagrams as hybrid circuits 81

5.3 Push-relabel algorithm 84
5.4 Global minima computation 88

Conclusion

Bibliography

Introduction

The theory of quantum mechanics has been around for more than a century. The foundations of our current understanding were developed at the beginning of the 1900s and have been refined ever since. This model has been used to successfully explain a wide range of phenomena, from the behaviour of atoms and molecules to the nature of light.

It's only relatively recently that we have wielded the properties of quantum mechanics to develop a new model of computation that is fundamentally different from the classical model. The execution of a quantum computer relies on some quantum property of the system, be it the spin of an electron or the polarization of a photon. It uses that degree of freedom to encode information as a superposition of states that would be intractable to perform on a classical computer.

The development of quantum computing is still in its infancy. The first milestone of *Quantum Supremacy* was achieved in 2019 [4], where a quantum computer was able to solve a task that would take a classical computer an impractical amount of time. However, the current state of quantum computing is still far from being able to solve real world problems. The near-term era of quantum computers is commonly referred to as Noisy Intermediate-Scale Quantum (*NISQ*) [60], where the quantum computers are limited to a few hundred qubits and are highly subject to noise which can corrupt the quantum state of the system. It is expected future quantum computers with a higher number of qubits and less noisy operations will be able to implement fault-tolerant schemes that will allow for the execution of more complex algorithms.

There are many research groups currently working on producing a quantum computer. They are all trying to solve the same problem but utilise for that different physical properties to encode the qubits. Nonetheless, most of these experimental computers share a common interface based on the circuit model. In this model, a quantum computer is composed of a set of qubits, each of which can be initialized in a particular state and then manipulated by a set of quantum gates. The qubits can also be measured, which will collapse the quantum state and return a classical bit.

In the circuit model, a user sends a series of instructions to the quantum computer, including some measurements that return the classical result of the operation.

However, a description of a complex algorithm in terms of low-level operations may be tedious and error prone to produce, and it would be tied to the specific set of gates available on the quantum computer. Instead, it is often more convenient to use a higher-level programming language to abstract over the specific sets of gates, and express generic operations such as working with lists of qubits or performing conditional branches.

One such language is *Quipper* [42], which is a functional quantum programming language able to generically manipulate circuits of qubits and can be compiled to a variety of quantum computing backends. The language has multiple formalisations of its semantics, including a recent variation called *Proto-Quipper-D* which contains a dependent type system, where the type of a term may depend on the value of some parameter.

It is the task of a quantum compiler to take a program written in a quantum programming language such as Quipper and produce a quantum circuit tailored to a specific architecture. This is normally a multi-stage process, where the program passes through various intermediate representations where it can be optimized at different abstraction levels.

In 2019, Duncan et al. [33] presented a new optimization procedure for quantum circuits based on a formal graphical language called the ZX calculus, which gives a more granular representation of linear maps than quantum circuits. ZX diagrams can be seen as open graphs with two kinds of nodes tagged with some labels. The calculus comes with a complete set of rewriting rules ensuring that we can rewrite the diagrams while maintaining the same interpretation. The optimization method takes a quantum circuit and effectively uses the ZX calculus as an intermediate representation to reduce its size and the amount of resources required to execute it.

In this thesis, we present various contributions to the different stages of a quantum compiler toolchain. Our first contribution is the definition of a new intermediate representation for quantum programs based on an extension of the ZX calculus called *Scalable ZX* (SZX), which allows us to compactly represent quantum circuits repeated structure. We introduce a non-trivial fragment of the Proto-Quipper-D language and present its encoding into families of such SZX-diagrams, that can be used to optimize recurrent segments of the program in a single optimization pass. This intermediate representation can then be transformed into a ZX diagram or a quantum circuit to continue the compilation process. This work was first presented at the *International Workshop on Programming Languages for Quantum Computing (PLanQC 2022)*, and a full version was published in the proceedings of the *Quantum Physics and Logic Conference (QPL 2022)* [9].

A second contribution of this work is the definition of a new optimization procedure for quantum circuits containing both quantum and classical segments, based on the method by Duncan et al. [33] We are able to use an extension of the ZX calculus

called *ZX Ground* ($\mathrm{ZX}_{\perp}$) to represent such hybrid programs, and optimize programs that encode some communication between the quantum and classical components. A preliminary version of this work was first presented in the proceedings of the *International Workshop on Quantum Compilation (IWQC 2020)*, and finally published at the *Asian Symposium on Programming Languages and Systems (APLAS 2021)* [11]. We have implemented our optimization procedure as an extension of the ZX-diagram manipulation tool `pyzx`, and upstreamed some of the changes to the main repository.

Finally, our last contribution pertains to the detection of classical segments in a hybrid quantum-classical circuit. Some automated optimization processes such as the one we present may produce quantum circuits that computes some operations that could be performed classically using fewer resources. We formalise the classicalisation problem and present an efficient heuristic to detect such segments. This work was developed jointly with the optimization procedure for hybrid quantum-classical circuits, and a succinct version was published at APLAS 2021 [11].

The rest of this thesis is organized as follows.

- In Chapter 1 we present the basic concepts of quantum computing and quantum programming.
- In Chapter 2 we present the ZX calculus and its extensions SZX and $\mathrm{ZX}_{\perp}$, used throughout the rest of this work. We additionally define the families of SZX diagrams and present a notation to easily instantiate a family multiple times in a single diagram.
- In Chapter 3 we present our first contribution; the definition of a Proto-Quipper-D fragment tailored to list manipulation, and its encoding into a new intermediate representation for quantum programs based on families of SZX-diagrams.
- In Chapter 4 we present our second contribution, the definition of a new optimization procedure for hybrid quantum-classical programs based on the pure optimization by Duncan et al.
- In Chapter 5 we present our last contribution, the definition of a heuristic to detect classical segments in a hybrid quantum-classical circuit.

1

Of Qubits and Quantum Machines

Quantum computing is a growing field of research that is expected to have a large impact on the future of computing. Quantum computers are able to solve problems that are intractable for classical computers, and to do so in a much shorter time. This is because quantum computers are able to exploit the quantum mechanical properties of matter, which are not available to classical computers. In this chapter, we will give a brief introduction to quantum mechanics focused on the properties that are relevant to this work. We will then discuss the basic concepts of quantum computing, and the advent of quantum programming languages for this new paradigm. For an extended introduction to the topics presented in this chapter we refer the reader to [56].

1.1 Quantum computation

1.1.1 Hilbert spaces

Hilbert spaces are mathematical objects that are used to model quantum systems. A Hilbert space $\mathcal{H}$ is a vector space equipped with an inner product $\langle \cdot, \cdot \rangle$, which is a conjugate symmetric bilinear map from the vector space to the complex or real numbers. The inner product must satisfy the following three properties:

1. Conjugate symmetric: $\langle x, y \rangle = \overline{\langle y, x \rangle}$
2. Linear: $\langle ax + by, z \rangle = a\langle x, z \rangle + b\langle y, z \rangle$
3. Positive definite: $\langle x, x \rangle \geq 0$ and $\langle x, x \rangle = 0$ if and only if $x = 0$

From properties 1 and 2 it follows that the inner product is conjugate linear in its second argument, $\langle x, ay + bz \rangle = \overline{a}\langle x, y \rangle + \overline{b}\langle x, z \rangle$

Example 1.1 The most common example of a Hilbert space, and the one used to model finite-dimensional quantum systems, is the vector space $\mathbb{C}^n$ of n complex numbers. In this case, the inner product is defined by the standard dot product $\langle x, y\rangle = \sum_{i=1}^{n} x_i \overline{y_i}$.

A useful way to represent vectors in a Hilbert space uses the Dirac notation, also called *braket* notation. A vector $v \in \mathcal{H}$ is represented as the *ket* $|v\rangle = \sum_{i=1}^{n} v_i |i\rangle$, where $|i\rangle$ is the i-th basis vector of the Hilbert space. Correspondingly, a *bra* is defined as the conjugate transpose $\langle v| = |v\rangle^*$ The inner product between two vectors $v, w \in \mathcal{H}$ is then given directly by $\langle v\,|\,w\rangle = \langle v|\,|w\rangle$, and the norm of a vector is given by $\|v\| = \sqrt{\langle v\,|\,v\rangle}$.

1.1.2 Pure quantum states

A quantum system is a composition of discrete elements called *qubits*. The state of a system at a given time is described by a *quantum state*, characterised by a unit vector in the Hilbert space $\mathbb{C}^{2^n}$ for an n-qubit system.

The state of a single qubit system is commonly expressed as a linear combination over the computational basis $\{|0\rangle, |1\rangle\} = \{(1,0)^\dagger, (0,1)^\dagger\}$ or the diagonal matrix $\{|+\rangle, |-\rangle\}$ where $|\pm\rangle = |0\rangle \pm |1\rangle$. A third, less commonly used basis called Y is formed by the vectors $|\circlearrowright\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$ and $|\circlearrowleft\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$.

Multi-qubit spaces are described by the composition of smaller spaces using a tensor product. Notice that while the spaces are compositions of smaller spaces, the state of such multi-qubit state may not be separable into the tensor product of smaller states. When this is the case, the state is said to be *entangled*.

Example 1.2 The state of a two-qubit system can be expressed as a vector in $\mathbb{C}^2 \otimes \mathbb{C}^2 = \mathbb{C}^4$. Let $|\phi\rangle = \frac{|00\rangle + |01\rangle}{\sqrt{2}}$ notice that $|\phi\rangle$ is a separable state as it can be written as the tensor product $|0\rangle \otimes |+\rangle$. On the other hand, the state $|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$ is entangled as it cannot be written as a tensor product of single-qubit states.

1.1.3 Quantum operators

The evolution of a closed quantum system is described by a unitary operator acting on the state of the system. A unitary operator is an operator $U : \mathcal{H} \to \mathcal{H}$ that satisfies the following conditions:

1. Linear: $U(x + y) = U(x) + U(y)$ and $U(\lambda x) = \lambda U(x)$
2. Invertible: $UU^{-1} = I$ and $U^{-1}U = I$

3. $U^{-1} = U^\dagger$

The action of a unitary operator U on a vector v is given by the matrix product Uv. The inverse of a unitary operator U is given by U^{-1}.

A consequence of the unitary property of quantum operators is that they do not allow the duplication of information. This is known as the *no-cloning theorem*. Indeed, suppose there is some unitary operator $U_n \in \mathbb{C}^{2^n \times 2^n}$ and $e \in \mathbb{C}^{2^n}$ such that for any state $|\phi\rangle \in \mathbb{C}^{2^n}$, $U|\phi e\rangle = |\phi\phi\rangle$. Then, the following must hold for any $\phi, \psi \in \mathbb{C}^{2^n}$:

$$\langle\phi|\psi\rangle = \langle\phi e|\psi e\rangle = \langle\phi e| U^\dagger U |\psi e\rangle = \langle\phi\phi|\psi\psi\rangle = \langle\phi|\psi\rangle^2$$

Which does not hold for any non-orthogonal or equal states.

Example 1.3 The most common examples of single-qubit unitary operators are the Pauli matrices $\sigma_x, \sigma_y, \sigma_z$. These operators are defined as follows:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Each of these operators is self-adjoint, and therefore unitary. The action of these operators on the computational basis is given by:

$$\begin{aligned} \sigma_x|0\rangle &= |1\rangle & \sigma_y|0\rangle &= i|1\rangle & \sigma_z|0\rangle &= |0\rangle \\ \sigma_x|1\rangle &= |0\rangle & \sigma_y|1\rangle &= -i|0\rangle & \sigma_z|1\rangle &= -|1\rangle \end{aligned}$$

These operators are commonly used as the basic building blocks for more complex operators, where they can be composed using the tensor product to operate over multi-qubit systems.

Example 1.4 The Hadamard operator is a single-qubit unitary operator that maps the computational basis $\{|0\rangle, |1\rangle\}$ to the diagonal basis $\{|+\rangle, |-\rangle\}$:

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

The action of the Hadamard operator on the computational basis is given by:

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

1.1.4 Quantum measurement

A measurement is the process by which a quantum system is observed. This process is described by a collection of *measurement operators* $\{M_m\}$ acting on the Hilbert space of the system, for a set of measurement results $\{m\}$. The measurement operators are required to satisfy the completeness condition:

$$\sum_m M_m^\dagger M_m = I$$

When a measurement is performed, the state of the system $|\phi\rangle$ is projected onto one of the candidate measurement results. The probability of obtaining a result m is given by the following equation:

$$p(m) = \langle\phi| M_m^\dagger M_m |\phi\rangle$$

The measurement process *collapses* the state of the system to the observed result, producing a new state $|\phi'\rangle$ given by:

$$|\phi'\rangle = \frac{M_m |\phi\rangle}{\sqrt{p(m)}}$$

Example 1.5 The most common example of a measurement is a single-qubit measurement in the computational basis. In this case, the measurement operators are given by:

$$M_0 = |0\rangle\langle 0|$$
$$M_1 = |1\rangle\langle 1|$$

The probability of obtaining the result m given a state $|\phi\rangle = \frac{1}{\sqrt{3}}|0\rangle + \frac{\sqrt{2}}{\sqrt{3}}|1\rangle$ is:

$$p(0) = \langle\phi| M_0^\dagger M_0 |\phi\rangle = \langle\phi| |0\rangle\langle 0| |\phi\rangle = \frac{1}{3}$$
$$p(1) = \langle\phi| M_1^\dagger M_1 |\phi\rangle = \langle\phi| |1\rangle\langle 1| |\phi\rangle = \frac{2}{3}$$

After the measurement, the state will collapse to either the state $|0\rangle$ or $|1\rangle$ with probability $p(0)$ and $p(1)$ respectively.

1.1.5 Mixed quantum systems

The previous presentation discussed closed quantum systems that are described by a single *pure* state and completely isolated from the environment. However, in reality, quantum systems are affected by the environment, and the state of the system may not be a pure state but a probability distribution over multiple states, $\{(p_i, |\phi_i\rangle)\}$. This is known as a *mixed state*. In this case, the state of the system can be compactly described by a *density matrix* ρ.

A density matrix $\rho \in \mathbb{C}^{2^n \times 2^n}$ is a positive semidefinite operator that satisfies the following conditions:

1. Positive semidefinite: $\langle\phi| \rho |\phi\rangle \geq 0$ for all $|\phi\rangle \in \mathbb{C}^{2^n}$
2. Hermitian: $\rho^\dagger = \rho$
3. Unitary trace: $\mathrm{tr}(\rho) = 1$

Given a mixed state characterised by the probability distribution $\{(p_i, |\phi_i\rangle)\}$, its density matrix is given by $\rho = \sum_i p_i |\phi_i\rangle\langle\phi_i|$. A density matrix corresponds to a pure state if and only if $\mathrm{tr}(\rho^2) = 1$.

Example 1.6 Consider a mixed quantum system in a state $|+\rangle$ that has been measured in the computational basis. The new state of the system is a probability distribution $\{(\frac{1}{2}, |0\rangle), (\frac{1}{2}, |1\rangle)\}$.

We can describe this mixed state using a density matrix:

$$\rho = \frac{1}{2} |0\rangle\langle 0| + \frac{1}{2} |1\rangle\langle 1| = \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Notice that this same density matrix can be obtained from the probability distribution $\{(\frac{1}{2}, |+\rangle), (\frac{1}{2}, |-\rangle)\}$, or generally from any other single-qubit orthogonal basis where all the states are equally likely. This mixed state is commonly referred to as the *maximally mixed state*.

The unitary and measurement operators can be applied over mixed states in a similar way to pure states. The action of a unitary operator U on a mixed state is given by:

$$\rho \mapsto U\rho U^\dagger$$

The application of a collection of measurement operators $\{M_m\}$ to a mixed state, can be described by the following equation:

$$\rho \mapsto \sum_m p(m) M_m \rho M_m^\dagger$$

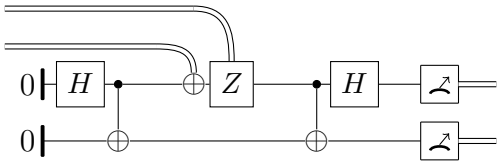
1.2 Quantum circuits

The most common representation of quantum computations is based on the circuit model. In this model a quantum computer is a black box that contains a number of qubits serving as a memory. This memory is interacted with using a fixed set of *quantum gates* that apply elementary unitary operations to the qubits.

The composition of multiple quantum gates is known as a *quantum circuit*. The circuit is depicted as a diagram where each gate is represented by a labelled box with a set of horizontal input and output wires to its left and right respectively. In pure circuits, each wire represents a qubit, and the input and output wires of a gate are connected to the wires of the qubits that the gate acts on. When representing unitary operations, the circuit will contain the same number of inputs and outputs.

It is also possible to represent non-unitary circuits that include classical data. For this case, a second kind of wire depicted as double lines is used to represent classical data in the form of bits. The gates may contain both types in their inputs and outputs, and the composition must only be performed between compatible sets of wires.

Example 1.1 The following is an example of a quantum circuit that performs the quantum experiment known as the superdense coding protocol. In the rest of this section, we will describe each part of the circuit in detail.



1.2.1 Single-qubit gates

The most basic quantum kind of gate is the *single-qubit gate*. The most common examples of these gates are the *Pauli gates*, which apply the corresponding to the Pauli operators σ_x, σ_y, and σ_z. These gates receive the name X, Y, and Z respectively.

A family of rotation gates can be derived by partially applying the Pauli gates. These operators are defined by the equations

$$R_x(\theta) = \cos(\theta/2)I - i\sin(\theta/2)\mathsf{X} = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

$$R_y(\theta) = \cos(\theta/2)I - i\sin(\theta/2)\mathsf{Y} = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

$$R_z(\theta) = \cos(\theta/2)I - i\sin(\theta/2)\mathrm{Z} = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

Other commonly used gates include the Hadamard gate H, which applies the Hadamard operator described in Section 1.1.3, the phase gate S, and the $\pi/8$ gate T. The last two gates are defined by the equations:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

The S and T gates are partial applications of the Pauli Z gate. They satisfy the following relation:

$$Z = S^2 = T^4$$

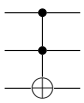
1.2.2 Multi-qubit gates

In addition to single-qubit gates it is also possible to apply unitary operations to multiple qubits simultaneously. These operations are known as *multi-qubit gates*. The most common example of a multi-qubit gate is the controlled-NOT gate, also known as the *CNOT gate* or *controlled-X gate*. This gate is defined by the equation:

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

The CNOT gate can be thought of as a single-qubit X gate that is applied to the second *target* qubit only when the first *control* qubit is in the state $|1\rangle$. It is normally represented using a special notation in quantum circuits:

There exists an extended version of the CNOT gate that contains two control qubits, and only applies the X operation on the target when the controls are in the state $|11\rangle$. This gate is known as the *Toffoli gate* and is written as:



In general, we can define the controlled version of any unitary operator U as

$$C(U) = \begin{pmatrix} I & 0 \\ 0 & U \end{pmatrix}$$

and depict it as follows:

Notice that the Toffoli gate corresponds to a controlled-controlled-NOT, Toffoli = $C(\text{CNOT}) = C(C(X))$.

A last example of a multi-qubit gate is simply the gate that swaps the states of two qubits. This gate is known as the *SWAP gate* and is written as:

Its definition as a unitary operator is given by:

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

1.2.3 Initialisation and measurement of qubits

During the execution of a quantum circuit, it may be desired to store some auxiliary information in the form of extra qubits. This information is known as an *ancilla* and is represented by a qubit that is not present as an input or output of the circuit. The ancilla qubit is initialized in the state $|0\rangle$, and is used to store and modify the state of other qubits. The ancilla is then discarded at the end of the computation. Similarly, it is possible to write non unitary circuits that create or discard qubits in the middle of the computation.

We represent the initialisation of new qubits in the state $|0\rangle$ and the discard operation as follows:

It is important to note that the discard operation requires the qubit to be in the state $|0\rangle$. If the qubit is in any other state, the discard operation will not preserve the norm of the state and the circuit will not be valid. In terms of linear maps the operations change the size of the vector space, either adding or removing a qubit:

$$\text{init} = |0\rangle \qquad \text{terminate} = \langle 0|$$

The termination of a qubit in an specific state is generally implemented as a measurement of the qubit followed by an assertion of its value. If an incorrect value is obtained the operation is aborted and the circuit is restarted. This is commonly known as *post-selection*.

1.2.4 Circuit interpretation

A quantum circuit can be interpreted into completely positive maps (CPM) acting over mixed quantum states. We denote the interpretation of a circuit C as $[\![C]\!]$. We

interpret a pure quantum gate described by the operator U as the following map:

$$\left[\!\left[-\boxed{U}- \right]\!\right] = \rho \mapsto U \rho\, U^\dagger$$

The parallel and sequential composition of circuits is defined as follows:

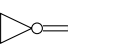
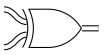
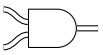
$$\left[\!\left[-\boxed{U}-\boxed{V}- \right]\!\right] = \left[\!\left[-\boxed{V}- \right]\!\right] \circ \left[\!\left[-\boxed{U}- \right]\!\right]$$

$$\left[\!\left[\begin{array}{l} -\boxed{U}- \\ -\boxed{V}- \end{array} \right]\!\right] = \rho_u \otimes \rho_v \mapsto \left[\!\left[-\boxed{U}- \right]\!\right](\rho_u) \otimes \left[\!\left[-\boxed{V}- \right]\!\right](\rho_v)$$

1.2.5 Hybrid quantum-classical circuits

In addition to the quantum gates described above, it is also possible to combine quantum and classical operations in the same circuit. This type of circuit is known as a *hybrid quantum-classical circuit* and allows us to represent the interaction between a quantum processor and a classical computer.

The classical part of the circuit is composed of classical gates, such as the well-known *NOT gate*, *AND gate*, and *XOR gate*. These gates are interconnected using classical wires that carry bits, represented by doubled lines. We use the traditional notation for classical logic gates:

NOT :  XOR :  AND : 

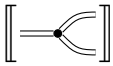
Since the classical gates operate over classical bits, their effect on the state of the system can be written directly as collapsing the states to the computational basis. We define these linear transformations as follows:

$$[\![\text{NOT}]\!] := \rho \mapsto |0\rangle\langle 1|\,\rho\,|1\rangle\langle 0| + |1\rangle\langle 0|\,\rho\,|0\rangle\langle 1|$$

$$[\![\text{XOR}]\!] := \rho \mapsto \sum_{i,j\in\{0,1\}} |(i \oplus j)\rangle\langle ij|\,\rho\,|ij\rangle\langle (i \oplus j)|$$

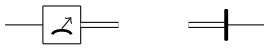
$$[\![\text{AND}]\!] := \rho \mapsto \sum_{i,j\in\{0,1\}} |(i * j)\rangle\langle ij|\,\rho\,|ij\rangle\langle (i * j)|$$

Additionally, we can also consider the fan-out and swap of wires as classical operations:

$$\left[\!\left[\text{  } \right]\!\right] := \rho \mapsto \sum_{i\in\{0,1\}} |ii\rangle\langle i|\,\rho\,|i\rangle\langle ii|$$

$$\left[\!\left[\text{  } \right]\!\right] := \rho \mapsto \sum_{i,j\in\{0,1\}} |ji\rangle\langle ij|\,\rho\,|ij\rangle\langle ji|$$

In the boundary between the quantum and classical parts of the circuit we must also insert *measurement* operations to collapse the quantum state carried by qubit wires into classical bits. Symmetrically, we can also prepare a new qubit from a classical bit using a *preparation* operation. The measurement operation is represented as a box with a gauge symbol, and the preparation gate is represented as a vertical line:



The interpretation of the measurement gate performs the operation described in Section 1.1.4. Since the preparation gate always has a classical bit as input, it can be described directly as the symmetric operation:

$$[\![\text{—}\boxed{\nearrow}\!\!=\!\!=]\!] = \rho \mapsto \sum_{i\in\{0,1\}} |i\rangle\langle i|\,\rho\,|i\rangle\langle i| \qquad [\![\!=\!\!|\text{—}]\!] = \rho \mapsto \sum_{i\in\{0,1\}} |i\rangle\langle i|\,\rho\,|i\rangle\langle i|$$

Classical qubits can be initialized and freely discarded, we include two special gates to represent these operations, and interpret them as follows:

$$[\![\!=\!|\,]\!] = \rho \mapsto \sum_{i\in\{0,1\}} \langle i|\,\rho\,|i\rangle \qquad [\![0 \,|\!=\!]\!] = |0\rangle\langle 0|$$

Finally, we can define the classical counterpart of the quantum-controlled gates presented in Section 1.2.2. The following gates correspond to a classically controlled NOT and to a classically controlled Z gate, respectively:

The interpretation of these gates is given by:

$$\left[\!\!\left[\boxed{U} \right]\!\!\right] := ([\![\!=\!|\,]\!] \otimes I) \circ \left[\!\!\left[\boxed{U} \right]\!\!\right]$$

1.2.6 Universal gate sets

The different realisations of the circuit model may be better suited to apply some gates more efficiently than others. For example, a common set of gates provided by experimental quantum processors is the set of operations $\{\mathrm{CNOT}, \mathrm{X}_\alpha, \mathrm{Z}_\alpha, \mathrm{H}\}$.

Although the set of gates provided by a quantum processor may be limited, they are generally capable of implementing any other quantum unitary by composing multiple gates. We call this kind of set of gates a *universal gate set*. In some other cases, the set of gates is not capable of exactly implementing all unitaries, but it is possible to approximate to any desired precision. In this case, we call the set an *approximate universal gate set*. Such is the case of the commonly used $\{\mathrm{CNOT}, R_x, R_z, \mathrm{H}, \mathrm{T}\}$ set.

1.3 Quantum lambda calculi

The quantum lambda calculus [64] is a quantum version of the lambda calculus, a formalism that was developed by Church and Curry in the 1930s. The introduction of the lambda calculus was a major step in the development of formal logic, as it allowed the definition of a simple formal system that encapsulates the concept of computation and can express arbitrarily complex computations. The original lambda calculus [7] is a very simple language, with only three different syntactic constructs: variables, abstractions and applications. The lambda calculus is a powerful tool for understanding the concept of computation and has been used to develop many different formal systems of computation, such as the family of quantum lambda calculi and a multitude of functional programming languages [51, 36, 49].

In general, the quantum lambda calculi manipulate quantum states or references to qubits inside the terms. These definitions usually require a linear typing system to ensure that well-typed terms do not copy the states, an operation which would violate the no-cloning theorem. In a linearly typed system, any quantum variable is required to be used exactly once in the term. This prevents both cloning the quantum state and inadvertently discarding it. Sometimes a less strict variation using affine types is used instead, which allows a variable to be implicitly discarded.

An important categorization of quantum lambda calculi is whether they allow quantum superposition of terms in addition to the quantum superposition of data. Such a feature in a quantum calculus is called *quantum control* [68] as it allows to model a program that can execute different code paths in superposition. This is a very powerful feature, but it is also very difficult to implement in a physically realizable quantum computer. A quantum lambda calculus with support for quantum control and data was described by Arrighi et al. [3], and further developed by Noriega and Diaz-Caro [57].

In this work, we will work solely with *classical control, quantum data* lambda calculi that do not allow superposition of terms, as dubbed by Selinger in [61]. This is an assumption that closely matches the circuit architecture described in Section 1.2, where a series of quantum operations are sent classically to the quantum processor. A well-known formulation of a classical control, quantum data lambda calculus was introduced by Selinger and Valiron [65] in 2009. They model the quantum computations as terms operating over pointers to qubits in an external quantum memory, closely resembling the circuit architecture. They further ensure a valid use of the qubit pointers by using a linear type system. Another formulation of a quantum lambda calculus with similar features was introduced in 2017 by Diaz-Caro [32]. Their model represents mixed quantum states using density matrices as part of the terms. A variation of their formulation expands on the concept of classical control by introducing probabilistic choices between terms that do not allow for quantum superposition of

terms.

A last property that is sometimes considered in quantum lambda calculi is having a *dependent type system* [8]. This is a system where the type of a term can depend on the value of some variable. For example, it can be used to generically model fixed-length lists of qubits. This is a very powerful feature, as it allows to generically operate over different types of data.

For this work we will reference a quantum lambda calculus called Proto-Quipper-D, introduced by Fu et al. [41], which introduces dependent types to model higher-order operations over qubits. This feature is particularly interesting in relation to the linear type system, as the requirement of using a variable exactly once may clash with its use as a dependent parameter if not properly handled. This language departs from the standard quantum lambda calculus model describing operations over quantum states and instead takes the approach of the Circuit-description languages discussed in Section 1.4, generating a list of quantum gate operations when evaluated.

1.4 Circuit-description languages

In the last decade a multitude of quantum programming languages have been proposed to implement quantum algorithms. Most of these languages are high-level languages that are not intended to be directly compiled to a quantum processor, but rather to be compiled to a more low-level language that can be compiled to a quantum processor. These high-level languages have been designed with the goal of being easy to read and understand, and to be able to express a wide range of quantum algorithms.

The most notable quantum programming languages are the ones developed by IBM, Google, and Microsoft. These languages are based on the Qiskit [37], Cirq [31], and Q# [67] frameworks respectively. They are all based on the circuit-description formalism and share a common set of quantum gates that are provided by the different quantum processors. They also provide a way to describe quantum circuits, and a way to compile them to specific quantum architectures.

Another quantum programming language is Quipper [42], introduced by Green et al. in 2013. The language is a functional programming model for quantum programs and is closely related to the quantum lambda calculus. Although the original definition implemented as a Haskell DSL does not use linear typing, a number of formal models of the language have been proposed under the name of Proto-Quipper [39, 40]. These extensions range from adding dependent types to the language, to allowing for interactions between the quantum execution and program generation using dynamic lifting of quantum programs. The former extension, called Proto-Quipper-D, has an alternative description using the equally named lambda calculus by Fu et al. [41]

mentioned in Section 1.3.

Most high-level quantum programming languages are compiled into quantum circuit descriptions using a quantum assembly language. That is, a low-level language that directly describes quantum circuits as a procedural list of operations applied over explicitly declared quantum registers. These languages are at the lowest level representation, that is then directly compiled to instructions for the quantum processor. The usual choice for this representation is *OpenQASM* [25], which is a standard used by most quantum processor providers to describe the quantum circuits that are to be executed.

The translation between a high-level language and the OpenQASM representation is usually done by a *quantum compiler* that in addition to performing a direct translation, may also perform a number of optimizations to reduce the number of gates and the number of qubits required to implement the algorithm. This step allows the high-level representation to ignore some of the technical details of the quantum processor, and to focus on a simple encoding of the algorithms. In this work we will discuss multiple contributions to the compilation pipeline for quantum programs.

2

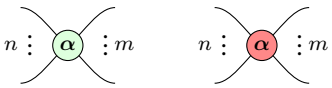
The ZX Calculus

The ZX calculus is a formal graphical language which provides a fine-grained representation of quantum operations. In this chapter we give an introduction to it and define a number of extensions that extend the representable operations and provide compact representations of repeated structure. An extensive introduction to the family of ZX calculi can be found in [69]. In Section 2.5 we introduce a novel definition of families of diagrams that we will require for compiling high level languages in Chapter 3.

2.1 The pure ZX calculus

In its pure version, the ZX calculus is a diagrammatic language that can encode any linear map between qubits. In particular, there is a direct translation from pure quantum circuits representing unitary operators into ZX diagrams.

The building blocks of a ZX diagram are the so-called *spiders*. These are represented as vertices in a graph, annotated with a *phase* in $[0, 2\pi)$. The spiders come in two varieties, *Z-spiders* drawn as green dots and *X-spiders* drawn as red dots,

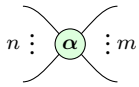


These spiders form the vertices of an open graph, with inputs on the left and outputs on the right.

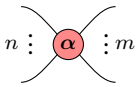
We can think of the phase of a spider as an angle of rotation on the corresponding basis. For example, a Z-spider with phase α and a single input and output represents the same computation as a $Rz(\alpha)$ gate,

We call a spider with a phase in $\{0, \pi\}$ a *Pauli spider*, since it represents a Pauli operation. Similarly, when the phase is $\frac{\pi}{2}$ we call it a *Clifford spider*.

The spiders can have multiple inputs and outputs. We can interpret these as a cloning and post-selection of the qubits over the corresponding basis. As a linear map, they can be interpreted as:



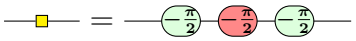
$$:= |0\rangle^{\otimes m} \langle 0|^{\otimes n} + e^{i\alpha} |1\rangle^{\otimes m} \langle 1|^{\otimes n}$$



$$:= |+\rangle^{\otimes m} \langle +|^{\otimes n} + e^{i\alpha} |-\rangle^{\otimes m} \langle -|^{\otimes n}$$

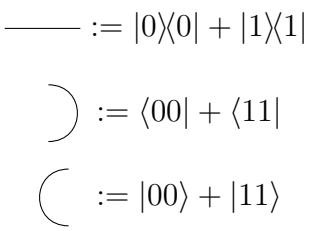
These multi-legged spiders do not have a direct equivalent as a circuit gate, but instead encode more granular operations (not necessarily unitary) that can be composed to represent complex gates.

There is a third generator of the ZX calculus that represents the Hadamard operator, drawn as a yellow box with exactly one input and output. It can be composed from the Z and X spiders using Euler's decomposition. To do so, we interconnect the outputs and inputs of the spiders using *wires* that carry the information of one qubit.



$$:= |+\rangle\langle 0| + |-\rangle\langle 1|$$

The straight wires of a ZX diagrams can be interpreted as simply identity operations, and the "curved" wires with only inputs or only outputs form similar maps. We call these last two *cups* and *caps*.



$$:= |0\rangle\langle 0| + |1\rangle\langle 1|$$

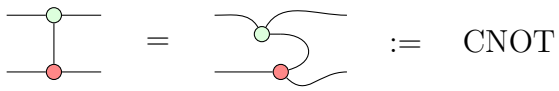
$$:= \langle 00| + \langle 11|$$

$$:= |00\rangle + |11\rangle$$

The diagrams can be composed sequentially ($\circ$) and in parallel ($\otimes$), and their interpretations are directly obtained from the corresponding operations. From this definition we can see that the specific structure of the wires is irrelevant to the semantics,

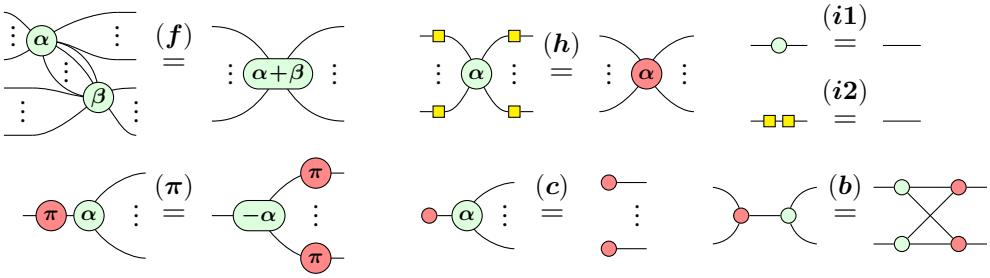
$$:= \quad (\langle 00| + \langle 11|) \otimes (|0\rangle\langle 0| + |1\rangle\langle 1|)) \circ ((|0\rangle\langle 0| + |1\rangle\langle 1|) \otimes (|00\rangle + |11\rangle))$$

$$= \quad |0\rangle\langle 0| + |1\rangle\langle 1| \quad =: \quad$$



In general, a ZX diagram can be freely deformed by moving the generators around and bending the wires without changing the interpretation. We say that *only the topology matters*, and therefore we can directly reason about the diagrams as open graphs.

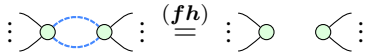
In addition to the topological properties, we can define several equalities between diagrams that let us rewrite parts of it without changing their semantics. Although multiple equalities are possible, here we present a small set that is *complete* [44]. This means that if two ZX diagrams represent the same linear map, there is a sequence of these rules that transforms one into the other.



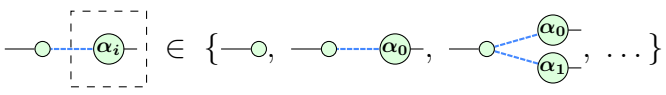
The rule (***f***) is called *spider-fusion*, as it takes multiple connected spiders of the same colour and fuses them together, adding their phases. Similarly, (***h***) is referred to as *colour changing rule*, as it swaps the colours of the generators. Using rules (***h***) and (***i2***), we can swap the colours of all the diagrams and derive the same equalities with the opposite colours.

For simplicity in our diagrams, we will sometimes use a notation in which we replace solely the Hadamard boxes with *Hadamard wires* drawn in blue, as follows.

We introduce the following equation, derived by Duncan et al. [33], to remove parallel Hadamard wires.



We utilise !-box notation [54] to represent infinite families of diagrams with segments that can be repeated 0 or more times. In the following sections it will be useful to use this notation for depicting more complex diagrams. Here we present an example of its usage.

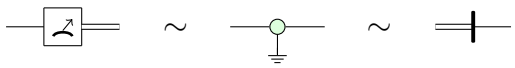


2.2 Grounded ZX

The description of quantum algorithms commonly involves quantum operations interacting with classical data in its inputs, outputs, or intermediary steps via measurements or state preparations. Some applications such as quantum error correction [34, 28] and quantum assertions [50, 71] explicitly introduce classical measurements and logic between quantum computations. In general, quantum programming languages usually allow for measurements and classically controlled quantum operators mixed-in with unitary gates [42, 25, 46, 66]. Furthermore, Jozsa [45] conjectured that any polynomial-time quantum algorithm can be simulated by polylogarithmic-depth quantum computation interleaved with polynomial-depth classical computation. As such, there is interest in representing this kind of structure in the ZX calculus.

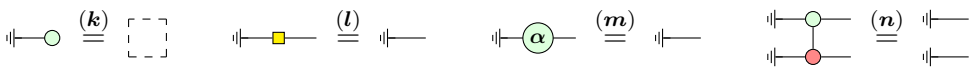
The $\mathrm{ZX}_{\fatslash}$ calculus [16] is an extension to the ZX-calculus which is able to easily describe such interactions with the environment. The diagrams have a standard interpretation as completely positive linear maps between quantum mixed states, cf. Section 2.3 for a formal description.

In addition to the ZX generators and rewrite rules, the calculus introduces a *ground* generator ($\fatslash$) which represents the tracing operation, or the discarding of information. When connected to a degree-3 green spider, this can correspond to a measurement operation over the computational basis or symmetrically to a qubit initialization from a bit.

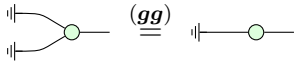


We refer to the spiders attached to $\fatslash$ generators as $\fatslash$ -spiders. Notice that we use the same kind of wire for both classical and quantum data, since we can always carry the information of the former over the latter. We will discuss a method to differentiate between the two types of wire in a diagram by using the $\fatslash$ -spiders in Chapter 5.

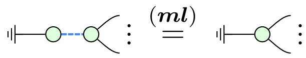
The $\mathrm{ZX}_{\fatslash}$ calculus extends the set of rewriting rules with the following additions.



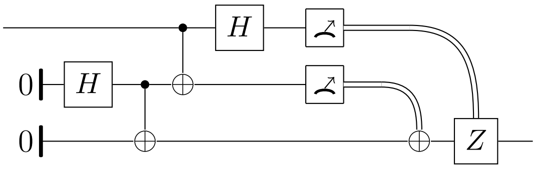
Intuitively, the $\fatslash$ generator discards any operation applied over a single qubit. Multiple discards can be combined into one via the following rule, derived from rules $(\boldsymbol{m})$, $(\boldsymbol{n})$, and $(\boldsymbol{k})$.



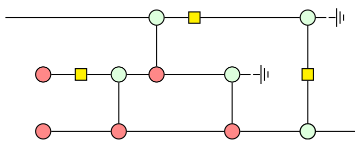
When using the Hadamard wire notation, we can use a ground to discard such wires using a combination of rules ($\boldsymbol{m}$) and ($\boldsymbol{l}$).



Example 2.1 Using the ZX$_{\text{⏚}}$ calculus, it is then possible to encode hybrid quantum-classical experiments as diagrams. For example, consider the following circuit representation of the quantum teleportation protocol.



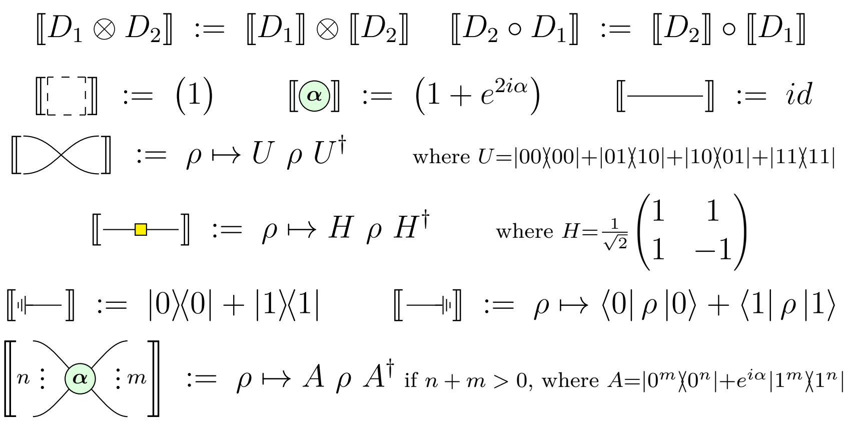
We can encode the same operations as a ZX$_{\text{⏚}}$ diagram as follows.

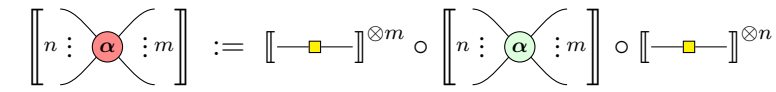


2.3 Semantics of the ZX and ZX$_{\text{⏚}}$ calculi

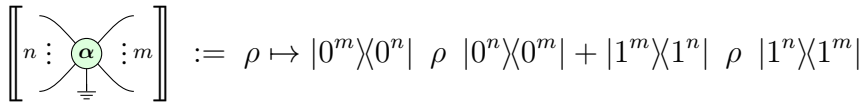
Carette et al. [16] define an interpretation of ZX$_{\text{⏚}}$ diagrams using a CPM construction [62]. We describe it here as an interpretation of diagrams into density matrices and completely positive maps modulo scalars. This definition can be used for the pure ZX calculus as well, by ignoring the cases pertaining the ⏚ generator.

Let $\mathcal{D}_n \subseteq \mathbb{C}^{2^n \times 2^n}$ be the set of n-qubit density matrices. There exists a map $[\![\cdot]\!] : \mathrm{ZX}_{\text{⏚}} \to \mathbf{CPM}(\mathbf{Qubit})$ which associates to any diagram $D : n \to m$ a completely positive map $[\![D]\!] : \mathcal{D}_n \to \mathcal{D}_m$, inductively defined as follows.

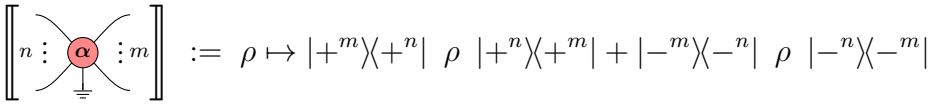




From the compositional rules, we can see that a ground attached to a green spider corresponds to a measurement over the computational basis.



It follows from rule (***h***) that the red ⏚ -spider corresponds to a measurement over the diagonal basis:



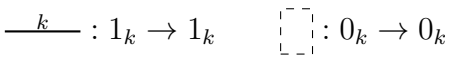
Notice that, in accordance to rule (***m***), the phase of a ⏚ -spider is irrelevant.

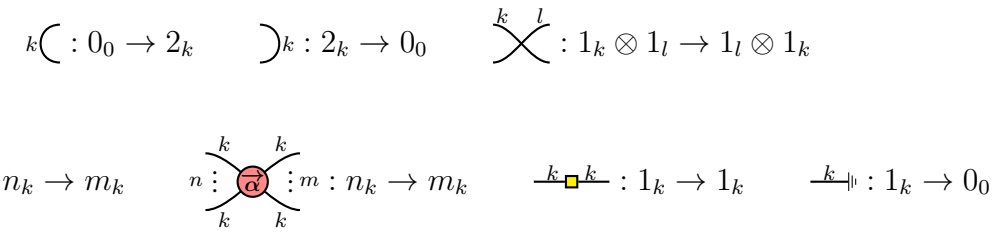
2.4 The Scalable ZX calculus

The ZX calculus as presented in Section 2.1 is a powerful tool for reasoning about quantum systems. However, it is not very efficient for large operations with repeated structure. Consider, for example, a parallel application of Z rotations over a n-qubit register. To represent such an operation a ZX-diagram requires n different green spiders composed in parallel. This linearly growing diagram can be replaced with a single-node diagram using the *Scalable* extension to the ZX-calculus.

The SZX calculus [15, 13] is an extension to the ZX calculus that generalizes the primitive constructors to work with arbitrarily sized qubit registers. This facilitates the representation of diagrams with repeated structure in a compact manner. Carette et al. [13] have shown that the scalable and the grounded extensions described in Section 2.2 can be directly composed, the resulting $\text{SZX}_{⏚}$ calculus will be referred to directly as SZX for simplicity.

Bold wires in a SZX diagram are tagged with a non-negative integer representing the size of the qubit register they carry, and other generators are marked in bold to represent a parallel application over each qubit in the register. Bold spiders with multiplicity k are tagged with k-sized vectors of phases $\overline{\alpha} = \alpha_1 :: \cdots :: \alpha_k$. Diagrams in the SZX calculus can be typed according to their input and output wires. We write 1_k to denote a wire multiplicity k, and $1_k \otimes 1_l$ to denote the parallel composition of two wires. For simplicity we will write $n_k = \bigotimes_{i=1}^{n} 1_k$. A diagram is then typed as the map between two wire multiplicities. The natural extension of the ZX generators corresponds to the following primitives:





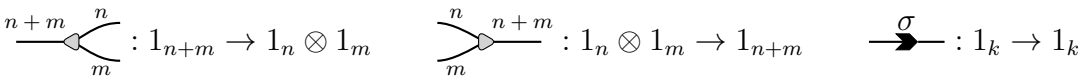
And the compositions,

$$D_1 \otimes D_2 : a_i \otimes c_k \to b_j \otimes d_l \qquad D_2 \circ D_1 : a_i \to d_l$$

where $D_1 : a_i \to b_j$, $D_2 : c_k \to d_l$, and $b_j = c_k$ for the sequential composition.

Wires of multiplicity zero are equivalent to the empty mapping. The *cup* and *cap* primitives that connect two outputs or two inputs together bend the wires while keeping the internal order of qubits in each register. We may omit writing the wire multiplicity if it can be deduced by context.

The extension defines two additional generators; a *split* node to split registers into multiple wires, and a function arrow to apply arbitrary functions over a register. In this work we restrict the arrow functions to permutations $\sigma : [0 \dots k) \to [0 \dots k)$ that rearrange the order of the wires. Cf. [13] for a description of the calculus including the generalized arrow generators. Using the split node and the wire primitives we can derive the rotated version, which we call a *gather*.

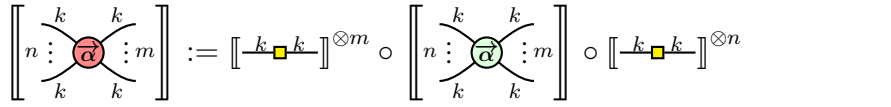


We reproduce here the standard interpretation of $\mathrm{SZX}_{\frac{1}{2}}$ diagrams as density matrices and completely positive maps [16, 13], modulo scalars. Let $D_n \subseteq \mathbb{C}^{2^n \times 2^n}$ be the set of n-qubit density matrices. We define the map $[\![\cdot]\!] : \mathrm{ZX}_{\frac{1}{2}} \to \mathbf{CPM}(\mathbf{Qubit})$ which associates to any diagram $D : n \to m$ a completely positive map $[\![D]\!] : \mathcal{D}_n \to \mathcal{D}_m$, inductively as follows.

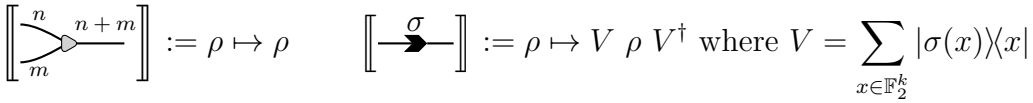
$$[\![D_1 \otimes D_2]\!] := [\![D_1]\!] \otimes [\![D_2]\!] \qquad [\![D_2 \circ D_1]\!] := [\![D_2]\!] \circ [\![D_1]\!]$$

$$[\![\ \overset{k}{\square}\ ]\!] := \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x,y \in \mathbb{F}_2^k} (-1)^{x \bullet y} |y\rangle\langle x|$$

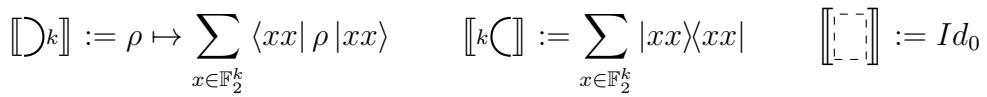
$$\left[\!\!\left[\begin{array}{c} \overset{k}{\cdots}\ \overset{k}{\cdots} \\ n \vdots \ \boxed{\vec{\alpha}}\ \vdots m \end{array} \right]\!\!\right] := \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k} e^{ix \bullet \vec{\alpha}} |x\rangle^{\otimes m} \langle x|^{\otimes n}$$



$$[\![\overset{k}{\text{—}}]\!] := \rho \mapsto \sum_{x \in \mathbb{F}_2^k} \langle x| \rho |x\rangle \qquad [\![\overset{k}{\text{⊣}}]\!] := \sum_{x \in \mathbb{F}_2^k} |x\rangle\langle x| \qquad [\![\overset{k}{\text{|}}]\!] := \rho \mapsto \rho$$



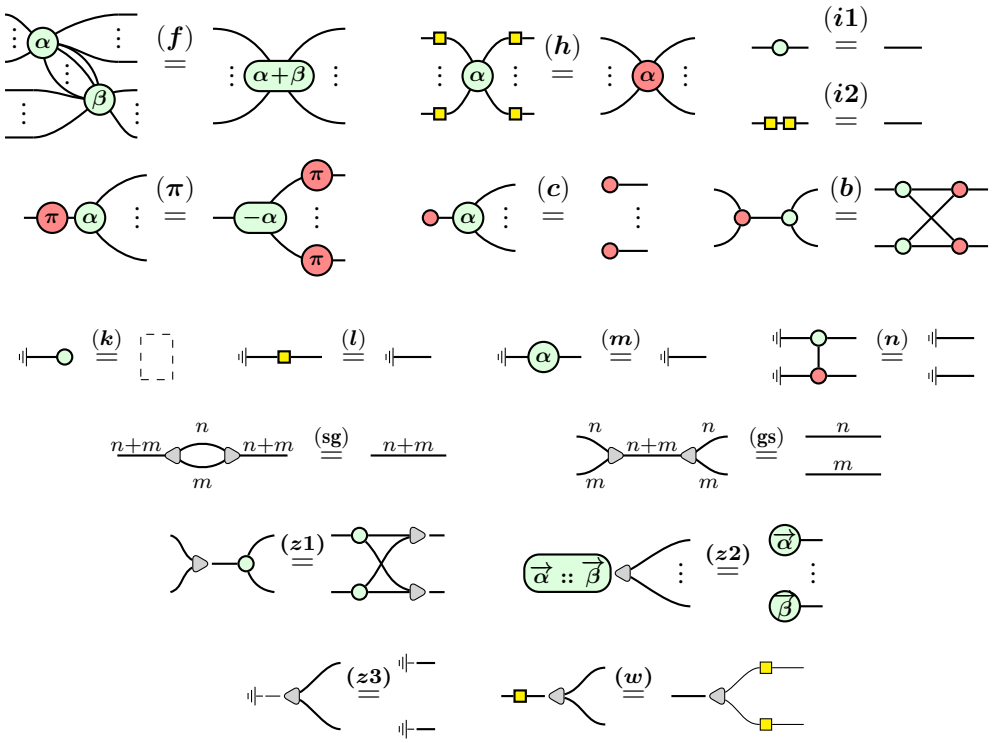
$$:= \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k} |\sigma(x)\rangle\langle x|$$



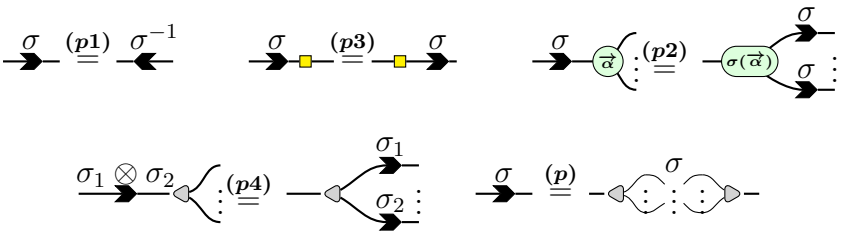
$$:= \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k, y \in \mathbb{F}_2^l} |yx\rangle\langle xy|$$

where $\forall u, v \in \mathbb{R}^n, u \bullet v = \sum_{i=1}^m u_i v_i$.

The $\text{SZX}_{\frac{1}{2}}$ calculus defines a set of rewrite rules by extending the ones of the ZX calculus and including extra rules pertaining to the gathers. We show this definition below.

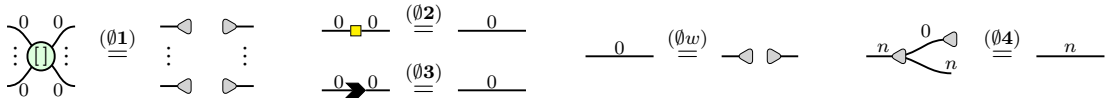


Additionally, for the arrows restricted to permutations of wires we include the following rules [13]:

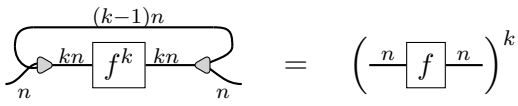


From the rewriting rules of the calculus, it follows that the wires of a SZX diagram can be freely deformed, and therefore it can be considered as an open graph where only the topology of its nodes and edges matters. We may also depict composition of gathers as single multi-legged generators. In an analogous manner, we will use a legless gather ⊣ to terminate wires with cardinality zero. This could be encoded as the zero-multiplicity spider ⊣[], which represents the empty mapping.

Since wires with cardinality zero correspond to empty mappings they can be discarded from the diagrams. The additional reduction rules representing this equivalence are given below.

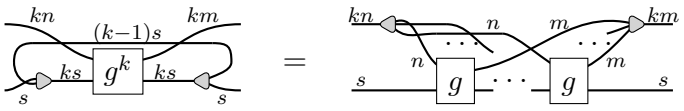


Carette et al. [13] showed that the SZX calculus can encode the repetition of a function $f : 1_n \to 1_n$ an arbitrary number of times $k \geq 1$ as follows:

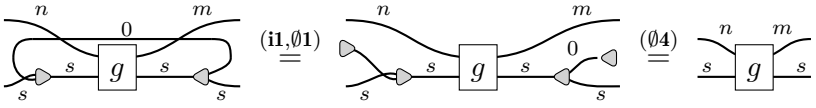


where f^k corresponds to k parallel applications of f. With a simple modification we can extend this construction to be able to encode an accumulating map operation.

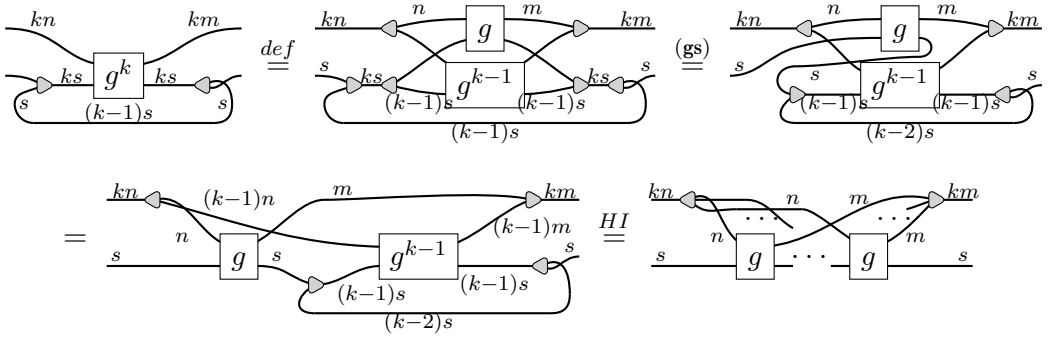
Lemma 2.1 Let $g : 1_n \otimes 1_s \to 1_m \otimes 1_s$ and $k \geq 1$, then



Proof By induction on k. If $k = 1$,



If $k > 1$,



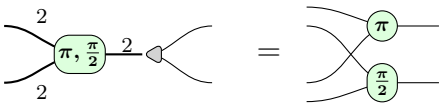
□

As an example, given a list $N = [n_1, n_2, n_3]$ and a starting accumulator value x_0, this construction would produce the mapping $([n_1, n_2, n_3], x_0) \mapsto ([m_1, m_2, m_3], x_3)$ where $(m_i, x_i) = g(n_i, x_{i-1})$ for $i \in 1 \ldots 3$.

2.4.1 Flattening SZX diagrams

Given a SZX diagram, it can always be flattened into a ZX diagram by translating the bold generators into multiple ZX generators composed in parallel.

Example 2.2 The following SZX diagram manipulating wires of size two can be flattened into a ZX diagram by cloning the spider and assigning the corresponding phases. Notice that the gather generator is simply a rearrangement of wires in the flattened diagram.



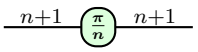
Using this flat interpretation, we can directly reason about the underlying open-graph of a SZX diagram. Notice, however, that the size of the graph depends directly on the multiplicity of the diagram generators, and may be much larger than the size of the diagram itself. In addition, each member of a diagram family may have a different underlying graph and must be considered separately.

2.5 SZX diagram families and list instantiation

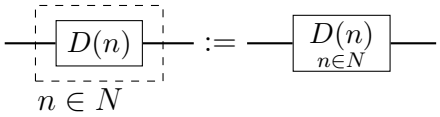
In addition to the compact representation allowed by the scalable extension, in some cases we want to represent generic operations indexed by an external variable. In this section we discuss a new definition of families of diagrams and introduce a syntactic sugar we call list instantiation to quickly describe such generic diagrams.

We introduce the definition of a family of SZX diagrams $D : \mathbb{N}^k \to \mathcal{D}$ as a function from k integer *parameters* to SZX diagrams. We require the structure of the diagrams to be the same for all elements in the family, parameters may only alter the wire tags and spider phases. Partial application is allowed, we write $D(n)$ to fix the first parameter of D.

Example 2.1 The following example describes a family of diagrams D that applies Z-rotations with angle π/n on $n+1$ qubits.

$$D := n \mapsto$$ 

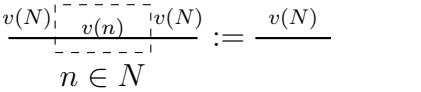
Since instantiations of a family share the same structure, we can compose them in parallel by merging the different values of wire tags and spider phases. We introduce a shorthand for instantiating a family of diagrams on multiple values and combining the resulting diagrams in parallel. This definition is strictly more general than the *thickening endofunctor* presented by Carette et al. [13], which replicates a concrete diagram in parallel. A *list instantiation* of a family of diagrams $D : \mathbb{N}^{k+1} \to \mathcal{D}$ over a list N of integers is written as $(D(n), n \in N)$. This results in a family with one fewer parameter, $(D(n), n \in N) : \mathbb{N}^k \to \mathcal{D}$. We graphically depict a list instantiation as a dashed box in a diagram, as follows.



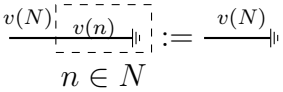
The definition of the list instantiation operator is given recursively on the construction of D below. On the diagram wires we use $v(N)$ to denote the wire cardinality $\sum_{n \in N} v(n)$, $\overrightarrow{\alpha}(N)$ for the concatenation of phase vectors $\overrightarrow{\alpha}(n_1) :: \cdots :: \overrightarrow{\alpha}(n_m)$, and $\sigma(N)$ for the composition of permutations $\bigotimes_{n \in N} \sigma(n)$. In general, a permutation arrow $\sigma(N, v, w)$ instantiated in concrete values can be replaced by a reordering of wires between two gather gates using the rewrite rule (**p**).

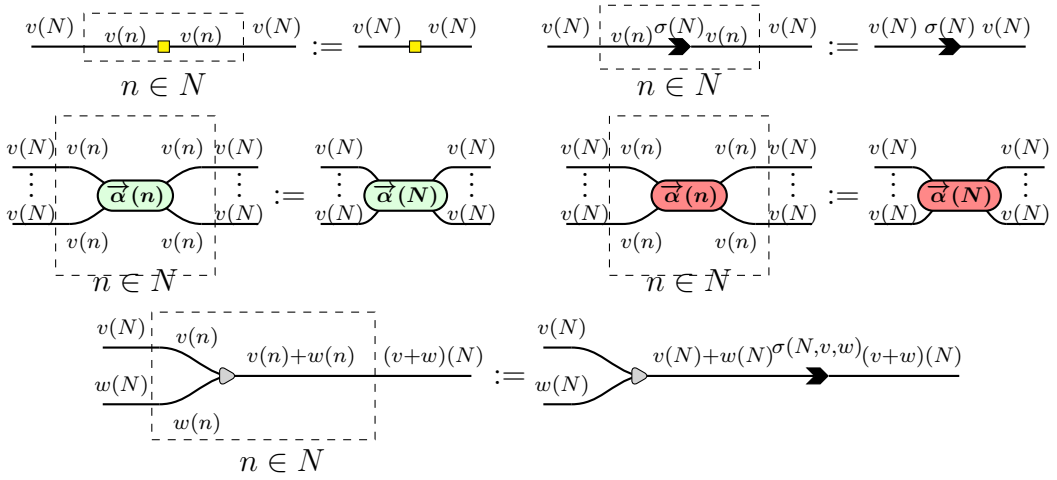
Given $D : \mathbb{N}^{k+1} \to \mathcal{D}$, $N = [n_1, \ldots, n_m] \in \mathbb{N}^m$,

$$((D_1 \otimes D_2)(n), n \in N) := (D_1(n), n \in N) \otimes (D_2(n), n \in N)$$



$$((D_2 \circ D_1)(n), n \in N) := (D_2(n), n \in N) \circ (D_1(n), n \in N)$$





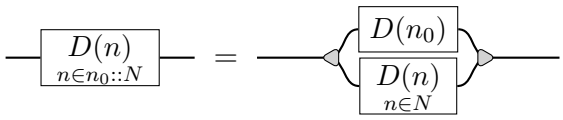
Where $\sigma(N, v, w) \in \mathbb{F}_2^{v(N)+w(N) \times v(N)+w(N)}$ is the permutation defined as the matrix

$$\sigma(N, v, w) = \left(\sigma_f^N | \sigma_g^N\right), \quad \sigma_f^N \in \mathbb{F}_2^{v(N)+w(N) \times v(N)}, \ \sigma_g^N \in \mathbb{F}_2^{v(N)+w(N) \times w(N)}$$

$$\sigma_f^{[]} = Id_0 \qquad \sigma_f^{n::N'} = \begin{pmatrix} Id_{v(n)} & 0 \\ 0 & 0 \\ 0 & \sigma_f^{N'} \end{pmatrix} \qquad \sigma_g^{[]} = Id_0 \qquad \sigma_g^{n::N'} = \begin{pmatrix} 0 & 0 \\ Id_{w(n)} & 0 \\ 0 & \sigma_g^{N'} \end{pmatrix}$$

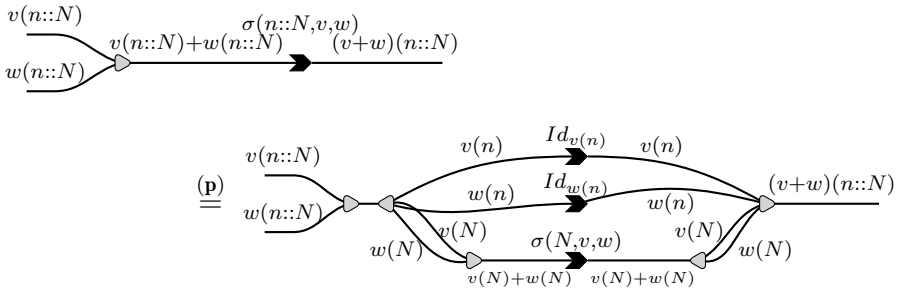
A list instantiation of a family of diagrams is equivalent to composing all the individual family instantiations in parallel, as proven by Lemmas 2.2 and 2.3. But in contrast to doing a parallel composition, we show in Lemma 2.4 that such operation adds a small number of additional nodes independently of the size of the list.

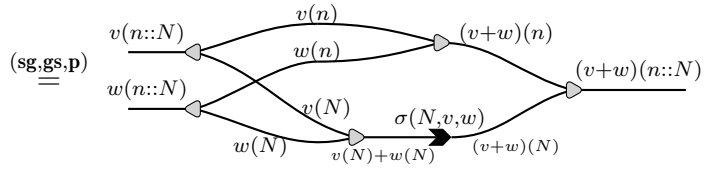
Lemma 2.2 For any diagram family D, $n_0 : \mathbb{N}$, $N : \mathbb{N}^k$,



Proof By induction on the term construction

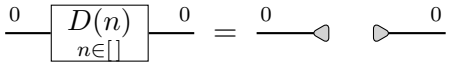
- If $\mathcal{D}$ is a gather,





- The other cases can be directly derived from the commutation properties of the gather generator via rules $(z1), (z2), (z3), (w)$, and $(p4)$. □

Lemma 2.3 A diagram family initialized with the empty list corresponds to the empty map. For any diagram family D,



Proof Notice that any wire in the initialized diagrams has cardinality zero. By rules (∅**1**), (∅**2**), (∅**3**), (∅**4**), and (∅**w**) every internal node can be eliminated from the diagram. □

Lemma 2.4 The list instantiation procedure on an n-node diagram family adds $\mathcal{O}(n)$ nodes to the original diagram.

Proof By induction on the term construction. Notice that the instantiation of any term except the gather does not introduce any new nodes, and the gather introduction creates exactly one extra node. Therefore, the list instantiation adds a number of nodes equal to the number of gather generators in the diagram. □

3

Compiling high-level quantum programs into SZX diagrams

The ZX calculus is a powerful tool when used as an intermediate representation language for quantum compilers. In this line it has successfully been used for the compilation and verification of quantum programs and quantum circuits.

Indeed, starting from a quantum circuit we are able to obtain an equivalent ZX representation directly by translating each gate into a small number of nodes. In Chapter 4 we will see an instance of a translation that generates diagrams with a similar size to the original circuit.

While the ZX calculus proves to be a good representation for specific quantum maps, it suffers the same limitation as the quantum circuits in that it cannot express generic operations with parametric iteration or parallelism. As an example, a function applying a quantum gate over each qubit in a register would be required to define the exact number of qubits before compiling it into a diagram with a proportional number of nodes.

An alternative representation of quantum programs can be obtained using families of SZX diagrams, introduced in Section 2.4. In this chapter we show how this extension can be used as an efficient intermediate representation to encode parametric parallel and iterative operations in constant-sized diagrams.

The Proto-Quipper-D language described in Section 1.4 is a powerful language for describing high-level quantum operations. This instance of the Proto-Quipper-D family includes support for dependently typed functions. Such capability allows encoding a function like the previously mentioned parallel gate application with an explicit natural parameter indicating the number of qubits to be used.

The full language however can describe operations that cannot be encoded as SZX diagrams, such as non-terminating operations and unbounded recursion. As such, we are required to define a less expressive fragment of the language that can be

completely encoded as SZX diagrams.

We start this chapter by introducing a suitable starting language based on Proto-Quipper-D in Section 3.1 and then define a compilation procedure for it in Section 3.3. Finally, in Section 3.4 we show an example encoding of the generic Quantum Fourier Transform (QFT) as a SZX diagram family.

3.1 The λ_D calculus

We begin by defining a starting language from which we can define a complete translation into SZX diagram families. In this section we present the calculus λ_D, as a subset of the strongly normalizing Proto-Quipper-D programs. That is, we consider the subset of terminating Proto-Quipper-D programs which we are able to encode as SZX diagrams.

The calculus includes products, fixed length vectors, natural numbers, and a number of quantum primitives. Terms are inductively defined by:

$$\begin{aligned} M, N, L := {} & x \mid C \mid \mathbb{R} \mid \texttt{U} \mid 0 \mid 1 \mid n \mid \texttt{meas} \mid \texttt{new} \mid \lambda x^S.M \mid M\ N \mid \lambda' x^P.M \mid M\ @\ N \mid \\ & \star \mid M \otimes N \mid \texttt{let}\ x^{S_1} \otimes y^{S_2} = M\ \texttt{in}\ N \mid M; N \mid \\ & \texttt{VNil}^A \mid M :: N \mid \texttt{let}\ x^S :: y^{\texttt{Vec}\ n\ S} = M\ \texttt{in}\ N \\ & M \square N \mid \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N \mid \texttt{for}\ k^P\ \texttt{in}\ M\ \texttt{do}\ N \end{aligned}$$

where C is a set of implicit bounded recursive primitives used for operating over vectors and describing iterating functions, $n \in \mathbb{N}$, $\square \in \{+, -, \times, /, \wedge\}$ and $\texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N$ is a conditional branching with a test-for-zero guard.

Here $\texttt{U}$ denotes a set of unitary operations and $\mathbb{R}$ is a phase shift gate with a parametrized angle. In this article we arbitrarily fix the former to the CNOT and Hadamard (H) gates and the latter to the arbitrary rotation gates $R_{z(\alpha)}$ and $R_{x(\alpha)}$, forming a universal quantum gate set.

For the remaining constants, 0 and 1 represent bits, `new` is used to create a qubit, and `meas` to measure it. $\star$ is the inhabitant of the `Unit` type, and the sequence $M; N$ is used to eliminate it. Qubits can be combined via the tensor product $M \otimes N$ with $\texttt{let}\ x^{S_1} \otimes y^{S_2} = M\ \texttt{in}\ N$ as its corresponding destructor.

The system supports lists; $\texttt{VNil}^A$ represents the empty list, $M :: N$ the constructor and $\texttt{let}\ x^S :: y^{\texttt{Vec}\ n\ S} = M\ \texttt{in}\ N$ acts as the destructor. Finally, the term $\texttt{for}\ k^P\ \texttt{in}\ M\ \texttt{do}\ N$ allows iterating over parameter lists.

The typing system is defined in Figure 3.1. We write $|\Phi|$ for the list of variables in a typing context Φ. The type $\texttt{Vec}\ n\ A$ represents a vector of known length n with elements of type A.

We differentiate between *state contexts* (Denoted with Γ and Δ) and *parameter contexts* (Denoted with Φ). For our case study, parameter contexts consist only of pairs $x : \texttt{Nat}$ or $x : \texttt{Vec}\ (n : \texttt{Nat})\ \texttt{Nat}$, since they are the only non-linear types of variables that we manage. Every other variable falls under the state context. The terms $\lambda x^S.M$ and MN correspond to the abstraction and application which will be used for state-typed terms. The analogous constructions for parameter-typed terms are $\lambda' x^P.M$ and $M\ @N$.

In this sense we deviate from the original Proto-Quipper-D type system, which supports a single context decorated with indices. Instead, we use a linear and non-linear approach similar to the work of Cervesato and Pfenning[19].

A key difference between Quipper (and, by extension, Proto-Quipper-D) and λ_D is the approach to defining circuits. In Quipper, circuits are an intrinsic part of the language and can be operated upon. In our case, the translation into SZX diagrams will be mediated with a function defined outside the language.

Types are divided into two kinds: parameter and state types. Both can depend on terms of type `Nat`. For the scope of this work, this dependence may only influence the size of vector types.

Parameter types represent non-linear variable types which are known at the time of generation of the concrete quantum operations. In the translation into SZX diagrams, these variables may dictate the labels of the wires and spiders. Vectors of `Nat` terms represent their cartesian product. On the other hand, state types correspond to the quantum operations and states to be computed. In the translation, these terms inform the shape and composition of the diagrams. Vectors of state type terms represent their tensor product.

In lieu of unbounded and implicit recursion, we define a series of primitive functions for performing explicit vector manipulation. These primitives can be defined in the original language, with the advantage of them being strongly normalizing. The first four primitives are used to manage state vectors, while the last one is used for generating parameters. For ease of translation some terms are decorated with type annotations. However, we will omit these for clarity when the type is apparent.

$$\begin{aligned}
&\texttt{accuMap}_{A,B,C} : (n : \texttt{Nat}) \to \\
&\qquad \texttt{Vec}\ n\ A \multimap \texttt{Vec}\ n\ (A \multimap C \multimap B \otimes C) \multimap C \multimap (\texttt{Vec}\ n\ B) \otimes C \\
&\texttt{split}_A : (n : \texttt{Nat}) \to (m : \texttt{Nat}) \to \texttt{Vec}\ (n+m)\ A \multimap \texttt{Vec}\ n\ A \otimes \texttt{Vec}\ m\ A \\
&\texttt{append}_A : (n : \texttt{Nat}) \to (m : \texttt{Nat}) \to \texttt{Vec}\ n\ A \multimap \texttt{Vec}\ m\ A \multimap \texttt{Vec}\ (n+m)\ A \\
&\texttt{drop} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ \texttt{Unit} \multimap \texttt{Unit} \\
&\texttt{range}\ : (n : \texttt{Nat}) \to (m : \texttt{Nat}) \to \texttt{Vec}\ (m-n)\ \texttt{Nat}
\end{aligned}$$

These primitives include an accumulating map operation over a list, utilities to separate and merge lists and products of lists, a function to drop empty lists, and a last one to generate a list with a range of integers.

Types: $A := S \mid P \mid (n : \texttt{Nat}) \to A[n]$
State types: $S := \texttt{B} \mid \texttt{Q} \mid \texttt{Unit} \mid S_1 \otimes S_2 \mid S_1 \multimap S_2 \mid \texttt{Vec}\ (n : \texttt{Nat})\ S$
Parameter types: $P := \texttt{Nat} \mid \texttt{Vec}\ (n : \texttt{Nat})\ \texttt{Nat}$
State contexts: $\Gamma, \Delta := \cdot \mid x : S, \Gamma$
Parameter contexts: $\Phi := \cdot \mid x : P, \Phi$

$$\frac{}{\Phi, x : A \vdash x : A}\ \mathsf{ax} \qquad \frac{}{\Phi \vdash 0 : \texttt{B}}\ \mathsf{ax}_0 \qquad \frac{}{\Phi \vdash 1 : \texttt{B}}\ \mathsf{ax}_1$$

$$\frac{n \in \mathbb{N}}{\Phi \vdash n : \texttt{Nat}}\ \mathsf{ax}_n \qquad \frac{\Phi \vdash M : \texttt{Nat} \quad \Phi \vdash N : \texttt{Nat}}{\Phi \vdash M \square N : \texttt{Nat}}\ \square$$

$$\frac{}{\Phi \vdash \texttt{meas} : \texttt{Q} \multimap \texttt{B}}\ \mathsf{meas} \qquad \frac{}{\Phi \vdash \texttt{new} : \texttt{B} \multimap \texttt{Q}}\ \mathsf{new} \qquad \frac{}{\Phi \vdash \star : \texttt{Unit}}\ \mathsf{ax}_{\texttt{Unit}}$$

$$\frac{}{\Phi \vdash \texttt{U} : \texttt{Q}^{\otimes n} \multimap \texttt{Q}^{\otimes n}}\ \mathsf{u} \qquad \frac{}{\Phi \vdash \mathbb{R} : (n : \texttt{Nat}) \to \texttt{Q}^{\otimes n} \multimap \texttt{Q}^{\otimes n}}\ \mathsf{r}$$

$$\frac{\Phi, \Gamma, x : A \vdash M : B}{\Phi, \Gamma \vdash \lambda x^A.M : A \multimap B}\ \multimap_i \qquad \frac{\Phi, n : \texttt{Nat}, \Gamma \vdash M : B[n]}{\Phi, \Gamma \vdash \lambda' n^{\texttt{Nat}}.M : (n : \texttt{Nat}) \to B}\ \to_i$$

$$\frac{\Phi, \Gamma \vdash M : A \multimap B \quad \Phi, \Delta \vdash N : A}{\Phi, \Gamma, \Delta \vdash MN : B}\ \multimap_e$$

$$\frac{\Phi, \Gamma \vdash M : (n : \texttt{Nat}) \to B \quad \Phi \vdash N : \texttt{Nat}}{\Phi, \Gamma \vdash M@N : B[n/N]}\ \to_e$$

$$\frac{\Phi, \Gamma \vdash M : \texttt{Unit} \quad \Phi, \Delta \vdash N : B}{\Phi, \Gamma, \Delta \vdash M; N : B}\ ; \qquad \frac{\Phi, \Gamma \vdash M : \texttt{Vec}\ 0\ A \quad \Phi, \Delta \vdash N : B}{\Phi, \Gamma, \Delta \vdash M;_v N : B}\ ;_{vec}$$

$$\frac{\Phi, \Gamma \vdash M : A \quad \Phi, \Delta \vdash N : B}{\Phi, \Gamma, \Delta \vdash M \otimes N : A \otimes B}\ \otimes$$

$$\frac{\Phi, \Gamma \vdash M : A \otimes B \quad \Phi, \Delta, x : A, y : B \vdash N : C}{\Phi, \Gamma, \Delta \vdash \texttt{let}\ x^A \otimes y^B = M\ \texttt{in}\ N : C}\ let_{\otimes}$$

$$\frac{}{\Phi \vdash \texttt{VNil}^A : \texttt{Vec}\ 0\ A}\ \texttt{VNil} \qquad \frac{\Phi, \Gamma \vdash M : A \quad \Phi, \Delta \vdash N : \texttt{Vec}\ n\ A}{\Phi, \Gamma, \Delta \vdash M :: N\ : \texttt{Vec}\ (n+1)\ A}\ \texttt{Vec}$$

$$\frac{\Phi, \Gamma \vdash M : \texttt{Vec}\ (n+1)\ A \quad \Phi, \Delta, x : A, y : \texttt{Vec}\ n\ A \vdash N : C}{\Phi, \Gamma, \Delta \vdash \texttt{let}\ x^A : y^{\texttt{Vec}\ n\ A} = M\ \texttt{in}\ N : C}\ let_{vec}$$

$$\frac{n : \texttt{Nat} \quad \Phi \vdash V : \texttt{Vec}\ n\ \texttt{Nat} \quad k : \texttt{Nat}, \Phi, \Gamma \vdash M : A[k]}{\Phi, \Gamma^n \vdash \texttt{for}\ k\ \texttt{in}\ V\ \texttt{do}\ M : \texttt{Vec}\ n\ A[k]}\ for$$

$$\frac{\Phi \vdash L : \texttt{Nat} \quad \Phi, \Gamma \vdash M : A \quad \Phi, \Gamma \vdash N : A}{\Phi, \Gamma \vdash \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N : A}\ ifz$$

Figure 3.1: Type system.

We additionally define the following helpful terms based on the previous primitives to aid in the manipulation of vectors.

$$\begin{aligned}
&\texttt{map}_{A,B} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ A \multimap\ \texttt{Vec}\ n\ (A \multimap B) \multimap \texttt{Vec}\ n\ B \\
&\texttt{fold}_{A,C} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ A \multimap\ \texttt{Vec}\ n\ (A \multimap C \multimap C) \multimap C \multimap C \\
&\texttt{compose}_{A} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ (A \multimap A) \multimap A \multimap A
\end{aligned}$$

The map and fold functions are directly derived from the accumap primitive, and the third one allows for composing multiple functions sequentially. They can be defined directly in the calculus as follows,

$$\begin{aligned}
\texttt{map} \ :=\ & \lambda' n^{\texttt{Nat}}.\ \lambda xs^{\texttt{Vec}\ n\ A}.\ \lambda fs^{\texttt{Vec}\ n\ (A \multimap B)}. \\
& \texttt{let}\ fs' \otimes u_1 = \texttt{accuMap}\ @n\ fs \\
& \qquad (\texttt{for}\ k\ \texttt{in}\ (0..n)\ \texttt{do}\ \lambda f.\lambda u.(\lambda x.\lambda u.fx \otimes u) \otimes u) \star \\
& \texttt{in let}\ xs' \otimes u_2 = \texttt{accuMap}\ @n\ xs\ fs' \star \\
& \texttt{in}\ u_1\ ;\ u_2\ ;\ xs' \\
\texttt{fold} \ :=\ & \lambda' n^{\texttt{Nat}}.\ \lambda xs^{\texttt{Vec}\ n\ A}.\ \lambda fs^{\texttt{Vec}\ n\ (A \multimap C \multimap C)}.\ \lambda z^{C}. \\
& \texttt{let}\ fs' \otimes u = \texttt{accuMap}\ @n\ fs \\
& \qquad (\texttt{for}\ k\ \texttt{in}\ (0..n)\ \texttt{do}\ \lambda f.\lambda u.(\lambda x.\lambda y.\star \otimes f\ x\ y) \otimes u) \star \\
& \texttt{in let}\ us \otimes r = \texttt{accuMap}\ @n\ xs\ fs'\ z \\
& \texttt{in}\ u\ ;\ \texttt{drop}\ @n\ us\ ;\ r \\
\texttt{compose} \ :=\ & \lambda' n^{\texttt{Nat}}.\ \lambda xs^{\texttt{Vec}\ n\ (A \multimap A)}. \\
& \texttt{fold}\ @n\ xs\ (\texttt{for}\ k\ \texttt{in}\ 0..n\ \texttt{do}\ (\lambda f.\lambda g.\lambda x.f\ (g\ x)))\ (\lambda x.x)
\end{aligned}$$

Notice that the parameter abstraction only takes single integers, and similarly none of the primitives take integer vectors as arguments. This highlights the need for inclusion of the `for` as a construction into the language. Since it acts over both parameter and state types, its function is effectively to bridge the gap between the two of them. This operation closely corresponds to the list instantiation procedure for SZX diagrams presented in the Section 2.4.

Example 3.1 Let xs be a vector of 2 open terms $R\ @k\ (\texttt{new}\ 0)$. The term

$$\texttt{for}\ k\ \texttt{in}\ 2::4::\texttt{VNil}^{\texttt{Nat}}\ \texttt{do}\ xs$$

generates a vector of quantum maps by instantiating the abstractions for each individual parameter in the vector of integers, equivalent to the vector of qubits

$$R\ @2\ (\texttt{new}\ 0) :: R\ @4\ (\texttt{new}\ 0) ::\texttt{VNil}^{\texttt{Q}}$$

The primitives can be encoded as Proto-Quipper-D functions, and programs in λ_D can be written and compiled into circuits using the `dpq` tool, published by Frank Fu [38]. The definitions in Proto-Quipper-D shown below have been fully checked with said tool.

```
module Primitives where
import "/dpq/Prelude.dpq"

foreach : ! forall a b (n : Nat)
  -> (Parameter a) => !(a -> b) -> Vec a n -> Vec b n
foreach f l = map f l

split : ! forall a (n : Nat) (m: Nat) -> Vec a (n+m) -> Vec a n * Vec a m
split n m v =
  case v of
    VNil -> (VNil, VNil)
    VCons x v' ->
      case n of
        Z -> (VNil, v)
        S n' ->
          let (v1, v2) = split n' m v'
          in (VCons x v1, v2)

cons : ! forall a (n : Nat) (m: Nat) -> Vec a n -> Vec a m -> Vec a (n+m)
cons n m vn vm =
  case vn of
    VNil -> VNil
    VCons x vn' -> x : cons n m vn' vm

accuMap : ! forall a b c (n : Nat)
  -> Vec a n -> Vec (a -> c -> (b,c)) n -> c -> (Vec b n, c)
accuMap n v fs z =
  case v of
    VNil -> (VNil, z)
    VCons x v' ->
      case n of
        S n' ->
          let (y, z') = f x z
          in (VCons y accuMap n' v' f z', z')

mapp : ! forall a b (n : Nat) -> Vec a n -> Vec (a -> b) n -> Vec b n
mapp n v f =
  let (v', _) = accuMap n v (\x z -> (f x, z)) VNil
```

```
  in v'

fold : ! forall a b (n : Nat) -> Vec a n -> Vec (a -> b -> b) n -> b -> b
fold n v f z =
  let (_, z') = accuMap n v (\x z -> (VNil, f x z)) z
  in z'

compose : ! (n : Nat) -> Vec (a -> a) n -> a -> a
compose n fs x = fold fs (replicate n (\f x -> f x)) x

range_aux : ! (n : Nat) -> (m : Nat) -> Nat -> Vec Nat (minus m n)
range_aux n m x =
  case m of
    Z -> VNil
    S m' -> case n of
              Z -> let r' = range_aux Z m' (S x)
                   in subst (\x -> Vec Nat x) (minusSZ' m') (VCons x r')
              S n' -> range_aux n' m' (S x)

range : ! (n : Nat) -> (m : Nat) -> Vec Nat (minus m n)
range n m = range_aux n m Z

drop : ! (n : Nat) -> Vec Unit n -> Unit
drop n v = case n of
            Z -> ()
            S n' -> case v of
                      VCons _ v' -> drop n' v'
```

3.2 Operational Semantics of the λ_D calculus

We define a call-by-value small step operational semantics for the λ_D calculus. The set of values is given as:

$$V := x \mid C \mid 0 \mid 1 \mid \texttt{meas} \mid \texttt{new} \mid \texttt{U} \mid \lambda x^S.M \mid \lambda' x^P.M \mid \star \mid M \otimes N \mid \texttt{VNil} \mid M :: N$$

The reduction system is defined in Figure 3.2.

A key point to note here is that the reduction is merely syntactical. The rewriting rules produce new λ_D terms without modifying the quantum states, the measurements and unitary operations are not reduced. This system is powerful enough for our goal of translating into SZX-diagrams. We include the rewrite rules for the primitives on Table 3.1.

$$(\lambda x.M)V \to M[V/x]$$
$$(\lambda' x.M)@V \to M[V/x]$$
$$\texttt{let } x \otimes y = M_1 \otimes M_2 \texttt{ in } N \to N[x/M_1][y/M_2]$$
$$\texttt{let } x :: y = M_1 :: M_2 \texttt{ in } N \to N[x/M_1][y/M_2]$$
$$\texttt{ifz } V \texttt{ then } M \texttt{ else } N \to \begin{cases} M & \text{If } V = 0 \\ N & \text{Otherwise} \end{cases}$$
$$\star \ ; \ M \to M$$
$$\texttt{VNil} \ ;_v \ M \to M$$
$$V_1 \square V_2 \to V \qquad \text{Where } V_i = n_i \text{ and } V = n_1 \square n_2 \text{ for any integer operation } \square$$
$$\texttt{for } k \texttt{ in } M_1 \ :: M_2 \texttt{ do } N \to N[k/M_1] \ :: \texttt{for } k \texttt{ in } M_2 \texttt{ do } N$$
$$\texttt{for } k \texttt{ in VNil do } N \to \texttt{VNil}$$

$$\frac{M \to N}{MV \to NV} \qquad \frac{M \to N}{LM \to LN} \qquad \frac{M \to N}{M@V \to N@V} \qquad \frac{M \to N}{L@M \to L@N}$$

$$\frac{M \to N}{\texttt{let } x \otimes y = M \texttt{ in } L \to \texttt{let } x \otimes y = N \texttt{ in } L}$$

$$\frac{M \to N}{\texttt{let } x :: y = M \texttt{ in } L \to \texttt{let } x :: y = N \texttt{ in } L}$$

$$\frac{M \to N}{L \square M \to L \square N} \qquad \frac{M \to N}{M \square V \to M \square V}$$

$$\frac{M \to N}{M \ ; \ L \to N \ ; \ L} \qquad \frac{M \to N}{\texttt{let } x : y = M \texttt{ in } L \to \texttt{let } x : y = N \texttt{ in } L}$$

$$\frac{M \to N}{\texttt{ifz } M \texttt{ then } L_1 \texttt{ else } L_2 \to \texttt{ifz } N \texttt{ then } L_1 \texttt{ else } L_2}$$

$$\frac{M \to N}{\texttt{for } k \texttt{ in } M \texttt{ do } L \to \texttt{for } k \texttt{ in } N \texttt{ do } L}$$

Figure 3.2: Reduction system of λ_D.

$$
\begin{aligned}
\texttt{accuMap}\ @n\ xs\ fs\ z \rightarrow & \texttt{ifz}\ n\ \texttt{then}\ xs\ ;_v\ fs\ ;_v\ \texttt{VNil} \otimes z\ \texttt{else} \\
& \texttt{let}\ x :: xs' = xs\ \texttt{in}\ \texttt{let}\ f :: fs' = fs\ \texttt{in} \\
& \texttt{let}\ y \otimes z' = f\ x\ z\ \texttt{in} \\
& \texttt{let}\ ys \otimes z'' = \texttt{accuMap}\ @(n-1)\ xs'\ fs'\ z'\ \texttt{in} \\
& (y :: ys) \otimes z'' \\
\texttt{split}\ @n\ @m\ xs \rightarrow & \texttt{ifz}\ n\ \texttt{then}\ \texttt{VNil} \otimes xs\ \texttt{else}\ \texttt{let}\ y :: xs' = xs\ \texttt{in} \\
& \texttt{let}\ ys_1 \otimes ys_2 = \texttt{split}@(n-1)\ @m\ xs'\ \texttt{in} \\
& (y :: ys_1) \otimes ys_2 \\
\texttt{append}\ @n\ @m\ xs\ ys \rightarrow & \texttt{ifz}\ n\ \texttt{then}\ xs\ ;_v\ ys\ \texttt{else} \\
& \texttt{let}\ x :: xs' = xs\ \texttt{in}\ x :: (\texttt{append}\ @(n-1)\ @m\ xs'\ ys) \\
\texttt{drop}\ @n\ xs \rightarrow & \texttt{ifz}\ n\ \texttt{then}\ xs\ ;_v\ \star\ \texttt{else} \\
& \texttt{let}\ x :: xs' = xs\ \texttt{in}\ x\ ;\ \texttt{drop}\ @(n-1)\ xs' \\
\texttt{range}\ @n\ @m \rightarrow & \texttt{ifz}\ m - n\ \texttt{then}\ \texttt{VNil}\ \texttt{else}\ n\ ::\texttt{range}\ @(n+1)\ @m
\end{aligned}
$$

Table 3.1: Reductions pertaining to the primitives.

Proto-Quipper-D has been given a categorical semantics in [41] based on a fibration of a certain monoidal category. We should then be able to define a categorical semantics for the λ_D calculus based on this construction. However, since our calculus is a reduced fraction of the full language there is interest in finding a restricted codomain that would allow for a simpler definition of the semantics. We have left this development as future work.

3.3 Encoding programs as diagram families

In this section we introduce an encoding of the lambda calculus presented in Section 3.1 into families of SZX diagrams with context variables as inputs and term values as outputs. We split the lambda-terms into those that represent linear mappings between quantum states and can be encoded as families of SZX diagrams, and parameter terms that can be completely evaluated at compile-time.

3.3.1 Parameter evaluation

We say a type is *evaluable* if it has the form $A = (n_1 : \texttt{Nat}) \to \cdots \to (n_k : \texttt{Nat}) \to P[n_1, \ldots, n_k]$ with P a parameter type. Since A does not encode a quantum operation, we interpret it directly into functions over vectors of natural numbers. The translation of an evaluable type, $\lfloor A \rfloor$, is defined recursively as follows:

$$\lfloor (n : \texttt{Nat}) \to B[n] \rfloor = \mathbb{N} \to \bigcup_{n \in \mathbb{N}} \lfloor B[n] \rfloor \qquad \lfloor \texttt{Nat} \rfloor = \mathbb{N} \qquad \lfloor \texttt{Vec}\ (n : \texttt{Nat})\ \texttt{Nat} \rfloor = \mathbb{N}^n$$

Given a type judgement $\Phi \vdash L : P$ where P is an evaluable type, we define $\lfloor L \rfloor_\Phi$ as the evaluation of the term into a function from parameters into products of natural numbers. Since the typing is syntax directed, the evaluation is defined directly over the terms as follows:

$$\lfloor x \rfloor_{x:\texttt{Nat},\Phi} = x, |\Phi| \mapsto x \qquad \lfloor n \rfloor_\Phi = |\Phi| \mapsto n$$

$$\lfloor M \square N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \square \lfloor N \rfloor_\Phi (|\Phi|)$$

$$\lfloor M\ ::\ N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \times \lfloor N \rfloor_\Phi (|\Phi|) \qquad \left\lfloor \texttt{VNil}^{\texttt{Nat}} \right\rfloor_\Phi = |\Phi| \mapsto []$$

$$\left\lfloor \lambda' x^P . M \right\rfloor_\Phi = x, |\Phi| \mapsto \lfloor M \rfloor_\Phi (x, |\Phi|) \qquad \lfloor M @ N \rfloor_\Phi = \lfloor M \rfloor_\Phi (\lfloor N \rfloor_\Phi (|\Phi|), \Phi)$$

$$\lfloor \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N \rfloor_\Phi = |\Phi| \mapsto \begin{cases} \lfloor M \rfloor_\Phi (|\Phi|) \text{ if } \lfloor L \rfloor_\Phi (|\Phi|) = 0 \\ \lfloor N \rfloor_\Phi (|\Phi|) \text{ otherwise} \end{cases}$$

$$\lfloor \texttt{range} \rfloor_\Phi = n, m, |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i$$

$$\lfloor \texttt{for } k \texttt{ in } V \texttt{ do } M \rfloor_\Phi = |\Phi| \mapsto \bigtimes_{k \in \lfloor V \rfloor_\Phi (|\Phi|)} \lfloor M \rfloor_{k:\texttt{Nat}\Phi} (k, |\Phi|)$$

$$\left\lfloor \texttt{let } x^P :: y^{\texttt{Vec } n\ P} = M \texttt{ in } N \right\rfloor_\Phi = |\Phi| \mapsto \lfloor N \rfloor_{x:P, y:\texttt{Vec } n\ P, \Phi} (y_1, [y_2, \ldots, y_n], |\Phi|)$$

Where $[y_1, \ldots, y_n] = \lfloor M \rfloor_\Phi (|\Phi|)$.

We can prove that the evaluation of terms behaves as expected, and that it is correct in respect to the operational semantics of the language.

Lemma 3.1 Given an evaluable type A and a type judgement $\Phi \vdash L : A$, $\lfloor L \rfloor_\Phi \in \left(\bigtimes_{x:P \in \Phi} \lfloor P \rfloor\right) \to \lfloor A \rfloor$. That is, $\lfloor L \rfloor_\Phi$ is a function from parameters to products of natural numbers in $\lfloor A \rfloor$.

Proof By induction on the typing judgement $\Phi \vdash M : A$:

- If $\Phi \vdash n : \texttt{Nat}$, then $\lfloor n \rfloor_\Phi = |\Phi| \mapsto n \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \mathbb{N}$.
- If $\Phi, x : A \vdash x : A$, then $\lfloor x \rfloor_{x:A,\Phi} = x, |\Phi| \mapsto x \in \bigtimes_{y:P \in x:A,\Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.
- If $\Phi \vdash M \square N : \texttt{Nat}$ and $\Phi, \Gamma, \Delta \vdash M :: N \ : \texttt{Vec } (n+1)\ A$, then $\lfloor M \square N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \square \lfloor N \rfloor_\Phi (\ |\Phi|\ )$.

  By inductive hypothesis, $\lfloor M \rfloor_\Phi (|\Phi|), \lfloor M \rfloor_\Phi (|\Phi|) \in \mathbb{N}$.

  Then, $|\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \square \lfloor N \rfloor_\Phi (|\Phi|) \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \mathbb{N}$.
- If $\Phi \vdash \lambda' x.M : (x : P) \to B$ then $\left\lfloor \lambda' x^P.M \right\rfloor_\Phi = x, |\Phi| \mapsto \lfloor M \rfloor_\Phi (x, |\Phi|)$. By inductive hypothesis, $\lfloor M \rfloor_\Phi (x, |\Phi|) \in \lfloor B \rfloor$. Then, $|\Phi| \mapsto \lfloor M \rfloor_\Phi (x, |\Phi|) \in \bigtimes_{y:P \in x:P\Phi} \lfloor P \rfloor \to \lfloor B \rfloor$.
- If $\Phi, \Gamma \vdash M @ N : B[x/r]$, then $\lfloor M @ N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (\lfloor N \rfloor_\Phi (|\Phi|), \Phi)$. By inductive hypothesis, $\lfloor N \rfloor_\Phi (|\Phi|) \in \mathbb{N}$ and $x \mapsto \lfloor M \rfloor_\Phi (x, \Phi) \in \lfloor x : \texttt{Nat} \to B[x] \rfloor$. Then, $|\Phi| \mapsto \lfloor M \rfloor_\Phi (\lfloor N \rfloor_\Phi (|\Phi|), \Phi) \in \bigtimes_{y:P \in \Phi} \lfloor P \rfloor \to \lfloor B[A/x] \rfloor$.
- If $\Phi \vdash \texttt{VNil}^A : \texttt{Vec } 0\ A$, then $\left\lfloor \texttt{VNil}^P \right\rfloor_\Phi = |\Phi| \mapsto [] \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \mathbb{N}^0$.
- If $\Phi, \Gamma, \Delta \vdash M :: N \ : \texttt{Vec } (n+1)\ A$, then $\lfloor M \ :: \ N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \times \lfloor N \rfloor_\Phi (|\Phi|)$. By inductive hypothesis $M_\Phi(|\Phi|) \in \lfloor A \rfloor$ and $\lfloor N \rfloor_\Phi (|\Phi|) \in \lfloor A \rfloor^n$. Then, $|\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \times \lfloor N \rfloor_\Phi (|\Phi|) \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \lfloor A \rfloor^{n+1}$.

- If $\Phi, \Gamma \vdash \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N : A$, then
  $\lfloor \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N \rfloor_\Phi = |\Phi| \mapsto \begin{cases} \lfloor M \rfloor_\Phi (|\Phi|) \text{ if } \lfloor L \rfloor_\Phi (|\Phi|) = 0 \\ \lfloor N \rfloor_\Phi (|\Phi|) \text{ otherwise} \end{cases}$.
  By inductive hypothesis $\lfloor m \rfloor_\Phi (|\Phi|), \lfloor N \rfloor_\Phi (|\Phi|) \in \lfloor A \rfloor$ and $\lfloor L \rfloor_\Phi (|\Phi|) \in \mathbb{N}$. Then $\lfloor \texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N \rfloor_\Phi \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.

- If $\Phi \vdash \texttt{for}\ k\ \texttt{in}\ N\ \texttt{do}\ M : \texttt{Vec}\ n\ A$, then $\lfloor \texttt{for}\ k\ \texttt{in}\ V\ \texttt{do}\ M \rfloor_\Phi = |\Phi| \mapsto \bigtimes_{k \in \lfloor N \rfloor_\Phi (|\Phi|)} \lfloor M \rfloor_{k:\texttt{Nat}\Phi} (k, |\Phi|)$. By induction hypothesis, $\lfloor N \rfloor_\Phi (|\Phi|) \in \texttt{Nat}^n$ and $x \mapsto \lfloor M \rfloor_\Phi (x, \Phi) \in \lfloor k : \texttt{Nat} \to A[k] \rfloor$.
  Then $|\Phi| \mapsto \bigtimes_{k \in \lfloor N \rfloor_\Phi (|\Phi|)} \lfloor M \rfloor_{k:\texttt{Nat}\Phi} (k, |\Phi|) \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \lfloor A[k/N] \rfloor^n$.

- If $\Phi \vdash \texttt{let}\ x^B : y^{\texttt{Vec}\ n\ B} = M\ \texttt{in}\ N : A$, then $\left\lfloor \texttt{let}\ x^P :: y^{\texttt{Vec}\ n\ P} = M\ \texttt{in}\ N \right\rfloor_\Phi = |\Phi| \mapsto \lfloor N \rfloor_{x:P,y:\texttt{Vec}\ n\ P\Phi} (y_1, [y_2, \ldots, y_n], |\Phi|)$ where $[y_1, \ldots, y_n] = \lfloor M \rfloor_\Phi (|\Phi|)$.
  By inductive hypothesis $\lfloor M \rfloor_\Phi (|\Phi|) \in \lfloor B \rfloor^n$ and $\lfloor N \rfloor_{x:P,y:\texttt{Vec}\ n\ P\Phi} (y_1, [y_2, \ldots, y_n\ ], |\Phi|) \in \lfloor A \rfloor$. Then $|\Phi| \mapsto \lfloor N \rfloor_{x:P,y:\texttt{Vec}\ n\ P\Phi} (y_1, [y_2, \ldots, y_n], |\Phi|) \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.

- If $\Phi \vdash \texttt{range}\ : (n : \texttt{Nat}) \to (m : \texttt{Nat}) \to \texttt{Vec}\ (m - n)\ \texttt{Nat}$, then $\lfloor \texttt{range}\ \rfloor_\Phi = n, m, |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i \in \bigtimes_{z:P \in x:\texttt{Nat}, y:\texttt{Nat}, \Phi} \lfloor P \rfloor \to \mathbb{N}^{m-n}$ □

To prove Lemma 3.3 we first require the following auxiliary Lemma.

Lemma 3.2 Given the typing judgements $\Phi, x : A \vdash M : B$, and $\Phi \vdash N : A$, then $\lfloor M \rfloor_{x:A,\Phi} (\lfloor N \rfloor_\Phi, |\Phi|) = \lfloor M[N/x] \rfloor_\Phi (\ |\Phi|\ )$. That is, passing a parameter to the evaluation of a term is equivalent to substituting the corresponding variable in the term.

Proof Proof by straightforward induction on M. □

Lemma 3.3 Given an evaluable type A, a type judgement $\Phi \vdash L : A$, and $M \to N$, then $\lfloor M \rfloor_\Phi = \lfloor N \rfloor_\Phi$.

Proof By induction on the evaluation function $\lfloor M \rfloor_\Phi$:

- If $M = x$, $M = n$, $M = \texttt{VNil}^{\texttt{Nat}}$, $M = \lambda' x.M'$, $M = M_1\ ::\ M_2$ or $M = \texttt{range}$ then M is in normal form and it does not reduce.

- If $M = M_1 \square M_2$ we have three cases:

– If $M \to M_1 \Box N$ with $M_2 \to N$, then:

$$\begin{aligned}\lfloor M_1 \Box M_2 \rfloor_\Phi &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (|\Phi|) \Box \lfloor M_2 \rfloor_\Phi (|\Phi|) \\ &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (|\Phi|) \Box \lfloor N \rfloor_\Phi (|\Phi|) \\ &= \lfloor M_1 \Box N \rfloor_\Phi\end{aligned}$$

– If $M \to N \Box V$ with $M_1 \to N$, then:

$$\begin{aligned}\lfloor M_1 \Box V \rfloor_\Phi &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (|\Phi|) \Box \lfloor V \rfloor_\Phi (|\Phi|) \\ &= |\Phi| \mapsto \lfloor N \rfloor_\Phi (|\Phi|) \Box \lfloor V \rfloor_\Phi (|\Phi|) \\ &= \lfloor N \Box V \rfloor_\Phi\end{aligned}$$

– If $M \to n$ with $M_i = n_i \in \mathbb{N}$ and $n = n_1 \Box n_2$, then:

$$\begin{aligned}\lfloor n_1 \Box n_2 \rfloor_\Phi &= |\Phi| \mapsto \lfloor n_1 \rfloor_\Phi (|\Phi|) \Box \lfloor n_2 \rfloor_\Phi (|\Phi|) \\ &= |\Phi| \mapsto n_1 \Box n_2 \\ &= |\Phi| \mapsto n \\ &= \lfloor n \rfloor_\Phi\end{aligned}$$

- If $M = M_1 \ @M_2$ we have three cases:

– If $M \to M_1 \ @N$ with $M_2 \to N$, then:

$$\begin{aligned}\lfloor M_1 \ @M_2 \rfloor_\Phi &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (\lfloor M_2 \rfloor_\Phi (|\Phi|), \Phi) \\ &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (\lfloor N \rfloor_\Phi (|\Phi|), \Phi) \\ &= \lfloor M_1 \ @N \rfloor_\Phi\end{aligned}$$

– If $M \to N@ \ V$ with $M_1 \to N$, then:

$$\begin{aligned}\lfloor M_1 \ @V \rfloor_\Phi &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (\lfloor V \rfloor_\Phi (|\Phi|), \Phi) \\ &= |\Phi| \mapsto \lfloor M_1 \rfloor_\Phi (\lfloor V \rfloor_\Phi (|\Phi|), \Phi) \\ &= \lfloor N \ @V \rfloor_\Phi\end{aligned}$$

– If $M \to M'[V/x]$ with $M_1 = \lambda' x.M'$ and $M_2 = V$, then:

$$\begin{aligned}\lfloor (\lambda' x.M)@V \rfloor_\Phi &= |\Phi| \mapsto \lfloor \lambda' x.M \rfloor_\Phi (\lfloor V \rfloor_\Phi (|\Phi|), \Phi) \\ &= |\Phi| \mapsto (x, |\Phi| \mapsto \lfloor M \rfloor_{x,\Phi} (x, |\Phi|))(\lfloor V \rfloor_\Phi (|\Phi|), \Phi) \\ &\overset{Lemma\ 3.3}{=} |\Phi| \mapsto \lfloor M[V/X] \rfloor_\Phi (|\Phi|) \\ &= \lfloor M[V/X] \rfloor_\Phi\end{aligned}$$

- If $M = \texttt{ifz}\ M'\ \texttt{then}\ L\ \texttt{else}\ R$ we have three cases:
  - If $M \to \texttt{ifz}\ N\ \texttt{then}\ L\ \texttt{else}\ R$ with $M' \to N$, then:
$$\begin{aligned}\llfloor \texttt{ifz}\ M'\ \texttt{then}\ L\ \texttt{else}\ R\rrfloor_\Phi &= |\Phi| \mapsto \begin{cases}\llfloor M\rrfloor_\Phi(|\Phi|) \text{ if } \llfloor M'\rrfloor_\Phi(|\Phi|) = 0\\ \llfloor N\rrfloor_\Phi(|\Phi|) \text{ otherwise}\end{cases}\\ &= |\Phi| \mapsto \begin{cases}\llfloor M\rrfloor_\Phi(|\Phi|) \text{ if } \llfloor M'\rrfloor_\Phi(|\Phi|) = 0\\ \llfloor N\rrfloor_\Phi(|\Phi|) \text{ otherwise}\end{cases}\\ &= \llfloor \texttt{ifz}\ N\ \texttt{then}\ L\ \texttt{else}\ R\rrfloor_\Phi\end{aligned}$$
  - If $M \to L$ with $M' = 0$, then:
$$\begin{aligned}\llfloor \texttt{ifz}\ M'\ \texttt{then}\ L\ \texttt{else}\ R\rrfloor_\Phi &= |\Phi| \mapsto \begin{cases}\llfloor M\rrfloor_\Phi(|\Phi|) \text{ if } \llfloor M'\rrfloor_\Phi(|\Phi|) = 0\\ \llfloor N\rrfloor_\Phi(|\Phi|) \text{ otherwise}\end{cases}\\ &= |\Phi| \mapsto \llfloor L\rrfloor_\Phi(|\Phi|)\\ &= \llfloor L\rrfloor_\Phi\end{aligned}$$
  - The symmetric case for the else branch is similar to the previous one.
- If $M = \texttt{for}\ k\ \texttt{in}\ M'\ \texttt{do}\ R$ we have three cases:
  - If $M \to \texttt{for}\ k\ \texttt{in}\ N\ \texttt{do}\ R$ with $M' \to N$, then:
$$\begin{aligned}\llfloor \texttt{for}\ k\ \texttt{in}\ M'\ \texttt{do}\ R\rrfloor_\Phi &= |\Phi| \mapsto \bigtimes_{k \in \llfloor M'\rrfloor_\Phi(|\Phi|)} \llfloor R\rrfloor_\Phi(k, |\Phi|)\\ &= |\Phi| \mapsto \bigtimes_{k \in \llfloor N\rrfloor_\Phi(|\Phi|)} \llfloor R\rrfloor_\Phi(k, |\Phi|)\\ &= \llfloor \texttt{for}\ k\ \texttt{in}\ N\ \texttt{do}\ R\rrfloor_\Phi\end{aligned}$$
  - If $M \to R[k/M_1]\ :: \texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R$ with $M' = V :: L$, then:
$$\begin{aligned}&\llfloor \texttt{for}\ k\ \texttt{in}\ M_1\ ::\ M_2\ \texttt{do}\ R\rrfloor_\Phi\\ &= |\Phi| \mapsto \bigtimes_{k \in \llfloor M_1\ ::\ M_2\rrfloor_\Phi(|\Phi|)} \llfloor R\rrfloor_\Phi(k, |\Phi|)\\ &= |\Phi| \mapsto \bigtimes_{k \in \llfloor M_1\rrfloor_\Phi(|\Phi|) \times \llfloor M_2\rrfloor_\Phi(|\Phi|)} \llfloor R\rrfloor_\Phi(k, |\Phi|)\\ &= |\Phi| \mapsto \llfloor R\rrfloor_{k,\Phi}(\llfloor M_1\rrfloor_\Phi(|\Phi|), |\Phi|) \times \bigtimes_{k \in \llfloor M_2\rrfloor_\Phi(|\Phi|} \llfloor R\rrfloor_{k,\Phi}(k, |\Phi|)\\ &\overset{\text{Lemma 3.3}}{=} \llfloor R[M_1/k]\rrfloor_\Phi \times \llfloor \texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R\rrfloor_\Phi\\ &= \llfloor R[M_1/k]\ ::\ \texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R\rrfloor_\Phi\end{aligned}$$

– If $M \to \texttt{VNil}$ with $M' = \texttt{VNil}^{\texttt{Nat}}$, then:

$$\begin{aligned}\left\lfloor \texttt{for } k \texttt{ in } \texttt{VNil}^{\texttt{Nat}} \texttt{ do } R \right\rfloor_\Phi &= |\Phi| \mapsto \bigtimes_{k \in \lfloor \texttt{VNil}^{\texttt{Nat}} \rfloor_\Phi(|\Phi|)} \lfloor R \rfloor_\Phi (k, |\Phi|) \\ &= |\Phi| \mapsto \bigtimes_{k \in []} \lfloor R \rfloor_\Phi (k, |\Phi|) \\ &= |\Phi| \mapsto [] \\ &= \left\lfloor \texttt{VNil}^{\texttt{Nat}} \right\rfloor_\Phi\end{aligned}$$

- If $M = \texttt{let } x :: y = M_1 \texttt{ in } M_2$ we have two cases:

  – If $M \to \texttt{let } x :: y = N \texttt{ in } M_2$ with $M_1 \to N$, then:
  $\lfloor \texttt{let } x :: y = M_1 \texttt{ in } M_2 \rfloor_\Phi$

$$\begin{aligned}&= |\Phi| \mapsto \lfloor M_2 \rfloor_\Phi (y_1, [y_2, \ldots, y_n], |\Phi|) \text{ where } [y_1,\ldots,y_n] = \lfloor M_1 \rfloor_\Phi(|\Phi|) \\ &= |\Phi| \mapsto \lfloor M_2 \rfloor_\Phi (y_1, [y_2, \ldots, y_n], |\Phi|) \text{ where } [y_1,\ldots,y_n] = \lfloor N \rfloor_\Phi(|\Phi|) \\ &= \lfloor \texttt{let } x :: y = N \texttt{ in } M_2 \rfloor_\Phi\end{aligned}$$

  – If $M \to N[x/M_1][y/M_2]$ with $M' = M_1 :: M_2$, then:

$$\begin{aligned}\lfloor \texttt{for } k \texttt{ in } M_1 \ :: \ M_2 \texttt{ do } N \rfloor_\Phi &= |\Phi| \mapsto \lfloor N \rfloor_{x,y,\Phi} (y_1, [y_2, \ldots, y_n], |\Phi|) \\ &= |\Phi| \mapsto \lfloor N \rfloor_{x,y,\Phi} (M_1, M_2, |\Phi|) \\ &\overset{\textit{Lemma } 3.3}{=} |\Phi| \mapsto \lfloor N[M_1/x][M_2/y] \rfloor_\Phi (|\Phi|) \\ &= \lfloor N[M_1/x][M_2/y] \rfloor_\Phi\end{aligned}$$

  where $[y_1, \ldots, y_n] = \lfloor M_1 :: M_2 \rfloor_\Phi (|\Phi|)$.

- If $M = \texttt{range } @n\ @M_2$ then:

$$\begin{aligned}\lfloor \texttt{range } @n @m \rfloor_\Phi &= |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i \\ &= |\Phi| \mapsto \begin{cases} [] \text{ if } n - m = 0 \\ n \times \bigtimes_{i=n+1}^{m-1} i \text{ otherwise} \end{cases} \\ &= |\Phi| \mapsto \begin{cases} [] \text{ if } \lfloor n - m \rfloor_\Phi = 0 \\ \lfloor n \rfloor_\Phi \times \lfloor \texttt{range } @(n+1) @m) \rfloor \text{ otherwise} \end{cases} \\ &= \lfloor \texttt{ifz } m - n \texttt{ then VNil else } n \ :: \texttt{range } \ @(n+1)\ @m \rfloor_\Phi \square\end{aligned}$$

3.3.2 Diagram encoding

A non-evaluable type has necessarily the form $A = (n_1 : \texttt{Nat}) \to \cdots \to (n_k : \texttt{Nat}) \to S$, with S any state type. We call such types *translatable* since they correspond to terms encoding quantum operations that can be described as families of diagrams.

We first define a translation $[\![\cdot]\!]$ from state types into wire multiplicities as follows. Due to the symmetries of the SZX diagrams, the linear functions can be represented in a similar way to products.

$$[\![\texttt{B}]\!] = 1 \qquad [\![\texttt{Q}]\!] = 1 \qquad [\![\texttt{Vec}\ (n : \texttt{Nat})\ A]\!] = [\![A]\!]^{\otimes n}$$

$$[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!] \qquad [\![A \multimap B]\!] = [\![A]\!] \otimes [\![B]\!]$$

Given a translatable type judgement

$$\Phi, \Gamma \vdash M : (n_1 : \texttt{Nat}) \to \cdots \to (n_k : \texttt{Nat}) \to S$$

we can encode it as a family of SZX diagrams

$$n_1, \ldots, n_k, |\Phi| \mapsto \overset{[\![\Gamma]\!]}{\boxed{M(|\Phi|)}} \overset{[\![S[|\Phi|]]\!]}{}$$

In our diagrams we will omit the brackets for clarity. In a similar manner to the evaluation, we define the translation $[\![M]\!]_{\Phi,\Gamma}$ recursively on the terms as follows:

$$[\![x]\!]_{\Phi,x:A} = |\Phi| \mapsto \overset{\text{A}}{} \qquad [\![0]\!]_\Phi = |\Phi| \mapsto \overset{\texttt{Q}}{\bullet} \qquad [\![1]\!]_\Phi = |\Phi| \mapsto \overset{\texttt{Q}}{\boxed{\pi}}$$

$$[\![\texttt{meas}]\!]_\Phi = |\Phi| \mapsto \overset{\texttt{Q}\multimap\texttt{Q}}{} \qquad [\![\texttt{new}]\!]_\Phi = |\Phi| \mapsto \overset{1\multimap 1}{}$$

$$[\![U]\!]_\Phi = |\Phi| \mapsto \overset{n\texttt{Q}\multimap n\texttt{Q}}{\boxed{U}} \qquad [\![R]\!]_\Phi = n, |\Phi| \mapsto \overset{\texttt{Q}\multimap\texttt{Q}}{\boxed{R(n)}}$$

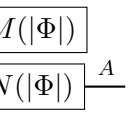
$$[\![\lambda' x^A.M]\!]_{\Phi,\Gamma} = x, |\Phi| \mapsto \overset{\Gamma}{\boxed{M(x,|\Phi|)}}\overset{A}{}$$

$$[\![M\ @N]\!]_{\Phi,\Gamma} = |\Phi| \mapsto \overset{\Gamma}{\boxed{M(\lfloor N \rfloor_\Phi(|\Phi|),|\Phi|)}}\overset{B}{}$$

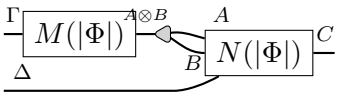
$$[\![\lambda x^A.M]\!]_{\Phi,\Gamma} = |\Phi| \mapsto \overset{\Gamma}{\boxed{M(|\Phi|)}}\overset{A}{}\overset{A\multimap B}{} \qquad [\![M\ N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto \overset{\Delta}{\boxed{N(|\Phi|)}}\overset{A}{}\ \overset{\Gamma}{\boxed{M(|\Phi|)}}\overset{A\multimap B}{}\overset{B}{}$$

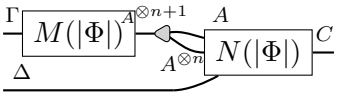
$$[\![M; N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto \overset{\Gamma}{\boxed{M(|\Phi|)}}\ \overset{\Delta}{\boxed{N(|\Phi|)}}\overset{A}{} \qquad [\![\star]\!]_\Phi = |\Phi| \mapsto \boxed{\ }$$

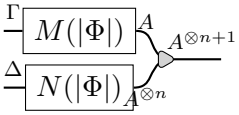
$$[\![M \otimes N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto \overset{\Gamma}{\boxed{M(|\Phi|)}}\overset{A}{}\ \overset{A\otimes B}{}\ \overset{\Delta}{\boxed{N(|\Phi|)}}\overset{B}{}$$

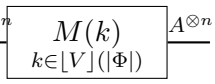
$$[\![M ;_v N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$$ 

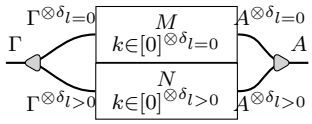
$$[\![\texttt{VNil}]\!]_{\Phi} = |\Phi| \mapsto$$ 

$$[\![let\ x^A \otimes y^B = M\ in\ N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$$ 

$$[\![let\ x^A : y^{\texttt{Vec}\ n\ A} = M\ in\ N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$$ 

$$[\![M :: N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$$ 

$$[\![\texttt{for}\ k\ \texttt{in}\ V\ \texttt{do}\ M]\!]_{\Phi,\Gamma^n} = |\Phi| \mapsto$$ 

$$[\![\texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N]\!]_{\Phi,\Gamma} = |\Phi| \mapsto$$ 

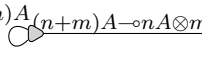
where δ is the Kronecker delta and $l = \lfloor L \rfloor (|\Phi|)$. Notice that the `new` and `meas` operations share the same translation. Although `new` can be encoded as a simple wire, we keep the additional $\fallingdotseq$ node to maintain the symmetry with the measurement.

The unitary operators U and rotations R correspond to a predefined set of primitives, and their translation is defined on a by case basis. The following table shows the encoding of the operators used in this procedure.

Name	$R_z(n)$	$R_z^{-1}(n)$	$R_x(n)$	$R_x^{-1}(n)$	H	CNOT
Encoding						

Notice that the spiders and Hadamard nodes from the ZX calculus are only required for the quantum primitives. Our choice of quantum operators is an arbitrary universal set that is easily representable in ZX. If required we could straightforwardly replace this definition with an alternative calculus supporting the scalable extension such as ZH [5], and use a different set of quantum primitives, without altering the rest of our translation.

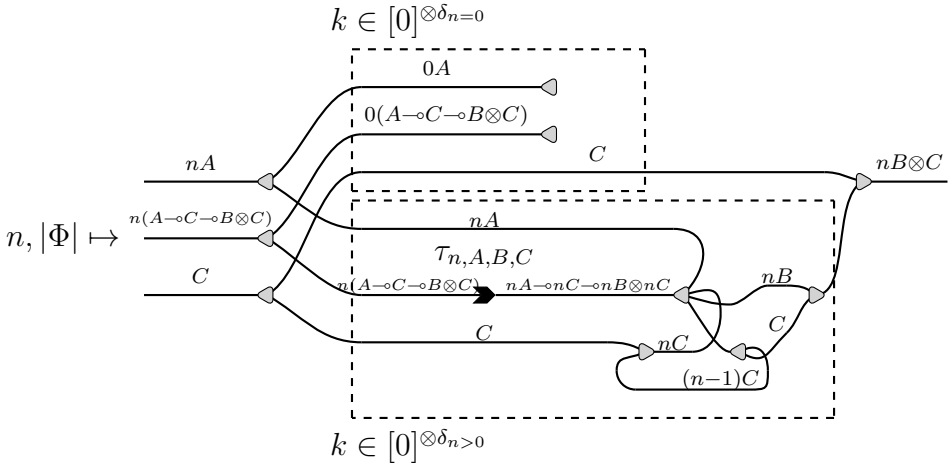
The primitives `split`, `append`, `drop` and `accuMap` are translated below. Since vectors are isomorphic to products in the wire encoding, the first three primitives do not perform any operation. For the accumulating map we utilize the construction presented in Lemma 2.1, replacing the function box with a function vector input. In the latter we omit the wires and gathers connecting the inputs and outputs of the function to a single wire on the right of the diagram for clarity.

$$[\![\texttt{split}_A]\!]_{\Phi} = n, m, |\Phi| \mapsto$$ 

$$[\![\texttt{append}_A]\!]_\Phi = n, m, |\Phi| \mapsto \overset{(n+m)A}{\bigcirc} \quad \overset{nA \multimap mA \multimap (n+m)A}{\triangleright}$$

$$[\![\texttt{drop}]\!]_\Phi = n, |\Phi| \mapsto \triangleright \overset{n0 \multimap 0}{}$$

$[\![\texttt{accuMap}_{A,B,C}]\!]_\Phi =$



where $\tau_{n,A,B,C}$ is a permutation that rearranges the vectors of functions into tensors of vectors for each parameter and return value. That is, $\tau_{n,A,B,C}$ reorders a sequence of registers $(A, C, B, C) \dots (A, C, B, C)$ into the sequence $(A \dots A)(C \dots C)(B \dots B)$ $(C \dots C)$. It is defined as follows,

$\tau_{n,A,B,C}(i) =$

$$\begin{cases} i \bmod k + a * (i \operatorname{div} k) & \text{if } i \bmod k < a \\ i \bmod k + c * (i \operatorname{div} k) + a * (n-1) & \text{if } a \leq i \bmod k < (a+c) \\ i \bmod k + b * (i \operatorname{div} k) + (a+c) * (n-1) & \text{if } (a+c) \leq i \bmod k < (a+c+b) \\ i \bmod k + c * (i \operatorname{div} k) + (a+c+b) * (n-1) & \text{if } (a+c+b) \leq i \bmod k \end{cases}$$

for $i \in [0, (a+c+b+c) * n)$, where mod and div are the integer modulo and division operators, $a = [\![A]\!]$, $b = [\![B]\!]$, $c = [\![C]\!]$, and $k = a + c + b + c$.

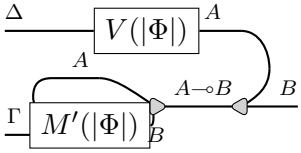
As a consequence of Lemma 2.4, the number of nodes in the produced diagrams grows linearly with the size of the input.

Lemma 3.4 The translation procedure is correct in respect to the operational semantics of λ_D. If A is a translatable type, $\Phi, \Gamma \vdash M : A$, and $M \to N$, then $[\![M]\!]_{\Phi,\Gamma} = [\![N]\!]_{\Phi,\Gamma}$.

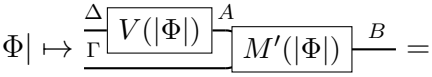
Proof By case analysis on the reductions of translatable terms.

- If $M = (\lambda x^A.M')V$ and $N = M'[V/x]$,

$$[\![(\lambda x^A.M')V]\!]_{\Phi,\Delta,\Gamma} = |\Phi| \mapsto$$



$$\overset{(\mathbf{gs})}{=} |\Phi| \mapsto$$



$$= [\![M'[V/x]]\!]_{\Phi,\Delta,\Gamma}$$

- If $M = (\lambda' x^A.M')\ @V$ and $N = M'[V/x]$,

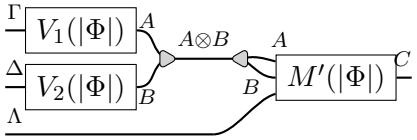
$$[\![(\lambda' x^A.M')\ @V]\!]_{\Phi,\Gamma} = |\Phi| \mapsto \overset{\Gamma}{\vdash}\boxed{\lfloor(\lambda' x^A.M')\rfloor_\Phi\,(\lfloor V\rfloor_\Phi\,(|\Phi|),|\Phi|)}\overset{B}{\dashv}$$

$$= |\Phi| \mapsto \overset{\Gamma}{\vdash}\boxed{\lfloor M'\rfloor_\Phi\,(\lfloor V\rfloor_\Phi\,(|\Phi|),|\Phi|)}\overset{B}{\dashv} = [\![M'[V/x]]\!]_{\Phi,\Gamma}$$

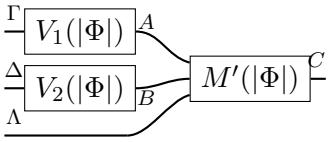
  By Lemma 3.2.

- If $M = \mathtt{let}\ x^A \otimes y^B = V_1 \otimes V_2\ \mathtt{in}\ M'$ and $N = M'[V_1/x][V_2/y]$,

$$[\![\mathtt{let}\ x^A \otimes y^B = V_1 \otimes V_2\ \mathtt{in}\ M']\!]_{\Phi,\Gamma,\Delta,\Lambda} = |\Phi| \mapsto$$



$$\overset{(\mathbf{gs})}{=} |\Phi| \mapsto$$

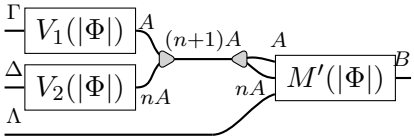


$$= [\![M'[V_1/x][V_2/y]]\!]_{\Phi,\Delta,\Gamma,\Lambda}$$

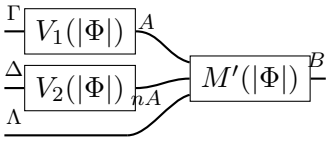
- If $M = \mathtt{let}\ x^A :: y^{\mathtt{Vec}\ n\ A} = V_1 :: V_2\ \mathtt{in}\ M'$ and $N = M'[V_1/x][V_2/y]$,

$$[\![\mathtt{let}\ x^A :: y^{\mathtt{Vec}\ n\ A} = V_1 :: V_2\ \mathtt{in}\ M']\!]_{\Phi,\Gamma,\Delta,\Lambda}$$

$$= |\Phi| \mapsto$$



$$\overset{(\mathbf{gs})}{=} |\Phi| \mapsto$$

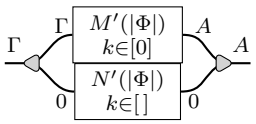


$$= [\![M'[V_1/x][V_2/y]]\!]_{\Phi,\Delta,\Gamma,\Lambda}$$

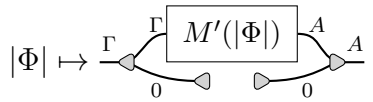
- If $M = \mathtt{ifz}\ L\ \mathtt{then}\ M'\ \mathtt{else}\ N'$,

– if $L = 0$ and $N = M'$, $\lfloor L \rfloor_\Phi = 0$ and

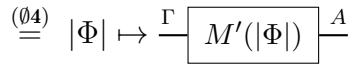
$$[\![\texttt{ifz } L \texttt{ then } M' \texttt{ else } N']\!]_{\Phi,\Gamma} = |\Phi| \mapsto$$



$$\overset{\textit{Lemma } 2.3}{=} |\Phi| \mapsto$$



$$\overset{(\emptyset 4)}{=} |\Phi| \mapsto$$

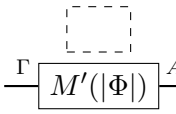


$$= [\![M']\!]_{\Phi,\Gamma}$$

– The case where $L > 0$ and $N = N'$ is symmetric to the case above.

- If $M = \texttt{VNil}; M'$ and $N = M'$,

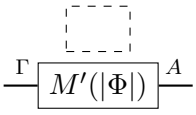
$$[\![\texttt{VNil};_v M']\!]_{\Phi,\Gamma} = |\Phi| \mapsto$$



$$= [\![M']\!]_{\Phi,\Gamma}$$

- If $M = \star; M'$ and $N = M'$,

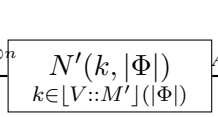
$$[\![\star; M']\!]_{\Phi,\Gamma} = |\Phi| \mapsto$$



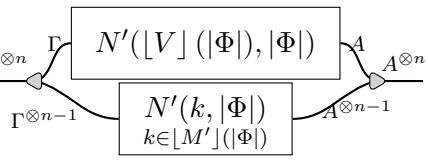
$$= [\![M']\!]_{\Phi,\Gamma}$$

- If $M = \texttt{for } k \texttt{ in } V :: M' \texttt{ do } N'$ and $N = N'[k/V] :: \texttt{for } k \texttt{ in } M' \texttt{ do } N'$,

$$[\![\texttt{for } k \texttt{ in } V :: M' \texttt{ do } N']\!]_{\Phi,\Gamma^{\otimes n}} = |\Phi| \mapsto$$



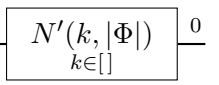
$$\overset{\text{Lemma } 2.2}{=} |\Phi| \mapsto$$



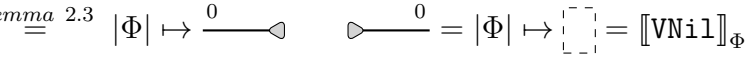
$$= [\![N'[k/V] :: \texttt{for } k \texttt{ in } M' \texttt{ do } N']\!]_{\Phi,\Gamma^{\otimes n}}$$

- If $M = \texttt{for } k \texttt{ in VNil do } N'$ and $N = \texttt{VNil}$,

$$[\![\texttt{for } k \texttt{ in VNil do } N']\!]_{\Phi} = |\Phi| \mapsto$$



$$\overset{\textit{Lemma } 2.3}{=} |\Phi| \mapsto$$

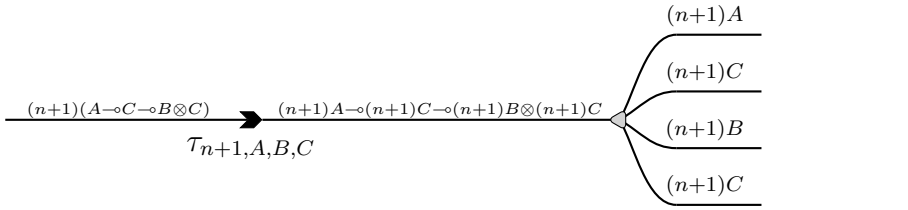


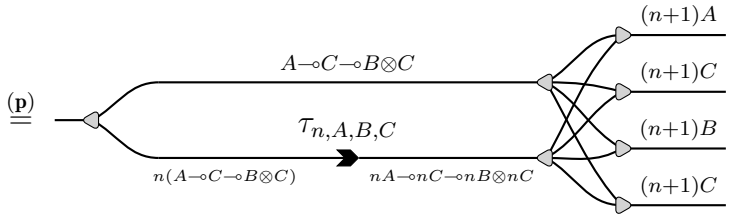
$$= |\Phi| \mapsto \boxed{} = [\![\texttt{VNil}]\!]_{\Phi}$$

- If $M = \texttt{accuMap}_{A,B,C}\ N'\ M_{xs}\ M_{fs}\ M_z$ and

  $N = \texttt{ifz}\ N'\ \texttt{then}\ xs\ ;_v\ fs\ ;_v\ \texttt{VNil} \otimes M_z\ \texttt{else}$
  $\texttt{let}\ x :: xs' = M_{xs}\ \texttt{in let}\ f :: fs' = M_{fs}\ \texttt{in let}\ y \otimes z' = f\ x\ z\ \texttt{in let}\ ys \otimes z'' = \texttt{accuMap}\ @(N'-1)\ xs'\ fs'\ z'\ \texttt{in}\ (y :: ys) \otimes z''$. Let $n = \lfloor N_1 \rfloor_\Phi (|\Phi|)$.

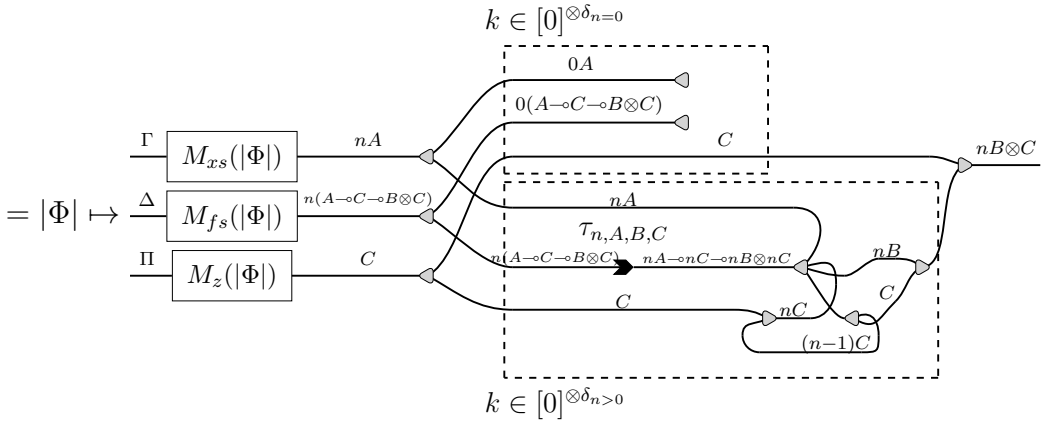
  Notice that, by definition of $\tau_{n,A,B,C}$,



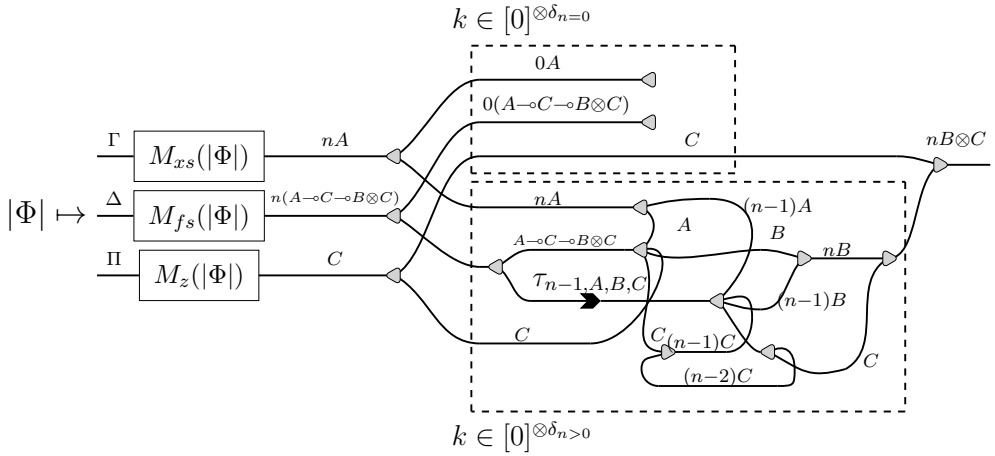


  Therefore,

  $\llbracket \texttt{accuMap}_{A,B,C}\ N'\ M_{xs}\ M_{fs}\ M_z \rrbracket_{\Phi,\Gamma,\Delta,\Pi}$



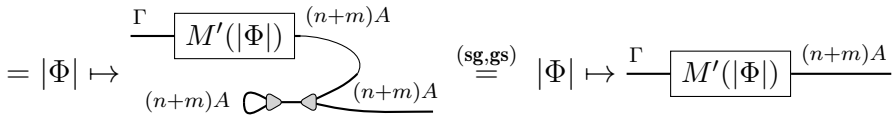
(p,gs,sg)

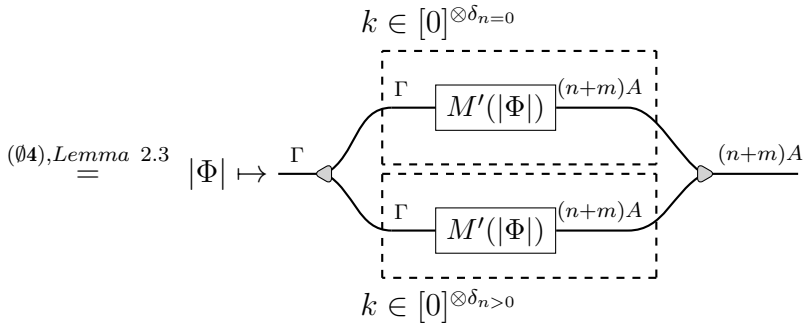


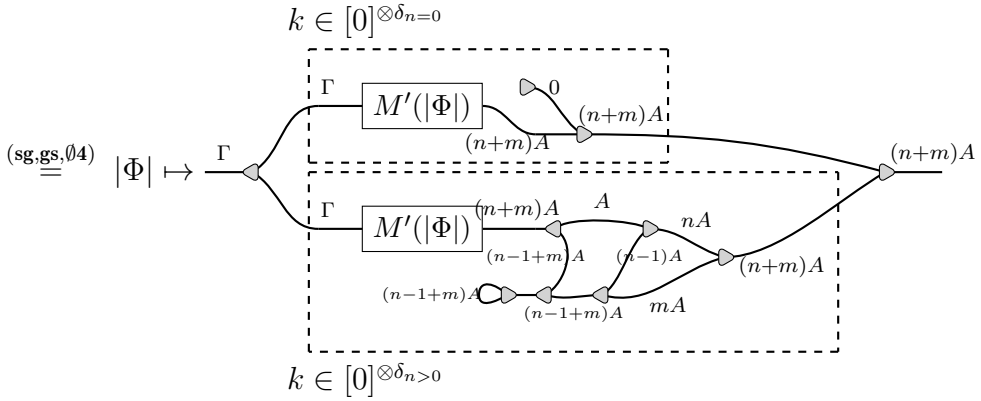
$= [\![\texttt{ifz}\ N'\ \texttt{then}\ xs\ ;_v\ fs\ ;_v\ \texttt{VNil} \otimes M_z\ \texttt{else}\ \texttt{let}\ x :: xs' = M_{xs}\ \texttt{in}\ \texttt{let}\ f :: fs' = M_{fs}\ \texttt{in}$
$\texttt{let}\ y \otimes z' = f\ x\ z\ \texttt{in}\ \texttt{let}\ ys \otimes z'' = \texttt{accuMap}\ @(N'-1)\ xs'\ fs'\ z'\ \texttt{in}\ (y :: ys) \otimes z'']\!]_{\Phi,\Gamma,\Delta,\Pi}$

- If $M = \texttt{split}_A\ @n\ @m\ xs$ and $N = \texttt{ifz}\ n\ \texttt{then}\ \texttt{VNil} \otimes xs\ \texttt{else}\ \texttt{let}\ y :: xs' = xs\ \texttt{in}\ \texttt{let}\ ys_1 \otimes ys_2 = \texttt{split}@(n-1)\ @m\ xs'\ \texttt{in}\ (y :: ys_1) \otimes ys_2$. Let $n = \lfloor N_1 \rfloor_\Phi (|\Phi|)$ and $m = \lfloor N_2 \rfloor_\Phi (|\Phi|)$.

  $[\![\texttt{split}_A\ @n\ @m\ xs]\!]_{\Phi,\Gamma}$

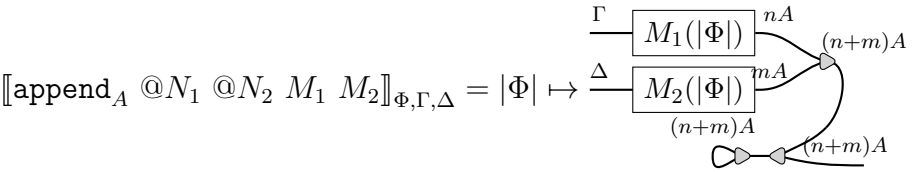


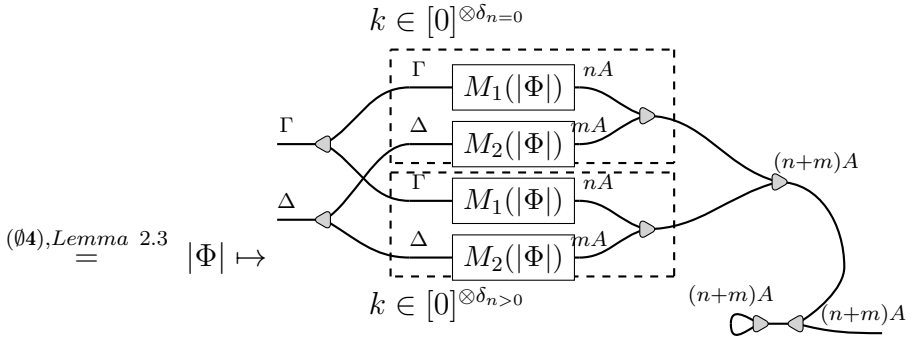


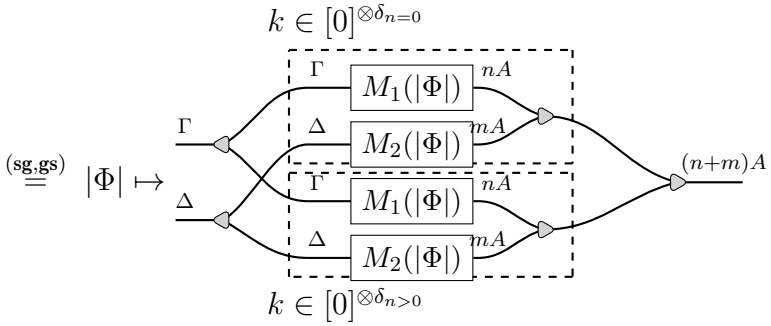


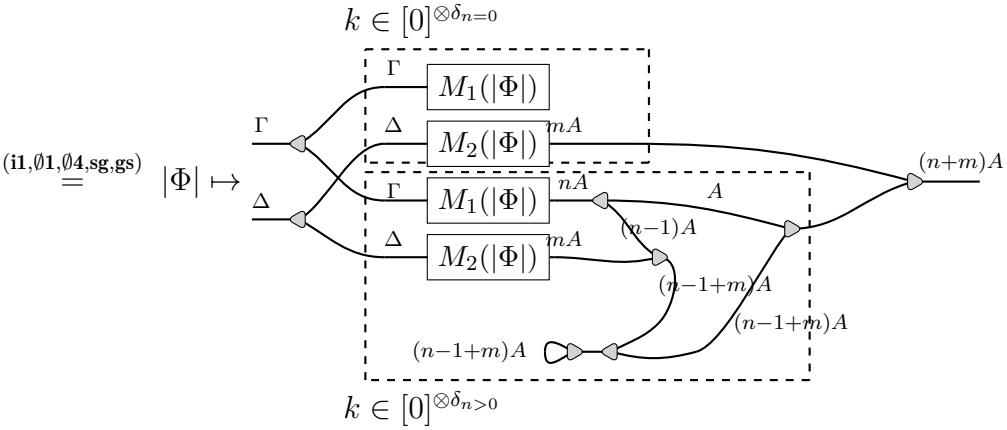
$= [\![\texttt{ifz}\ n\ \texttt{then}\ \texttt{VNil} \otimes xs\ \texttt{else}\ \texttt{let}\ y :: xs' = xs\ \texttt{in}\ \texttt{let}\ ys_1 \otimes ys_2 = \texttt{split}\ @(n-1)\ @m\ xs'\ \texttt{in}\ (y :: ys_1) \otimes ys_2]\!]_{\Phi,\Gamma}$

- If $M = \texttt{append}_A\ @N_1\ @N_2\ M_1\ M_2$ and $N = \texttt{ifz}\ N_1\ \texttt{then}\ M_1\ ;_v\ M_2\ \texttt{else}\ \texttt{let}\ x :: xs' = M_1\ \texttt{in}\ x :: (\texttt{append}\ @(N_1 - 1)\ @N_2\ M_1\ M_2)$. Let $n = \lfloor N_1 \rfloor_\Phi (|\Phi|)$ and $m = \lfloor N_2 \rfloor_\Phi (|\Phi|)$.



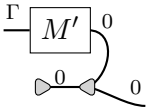


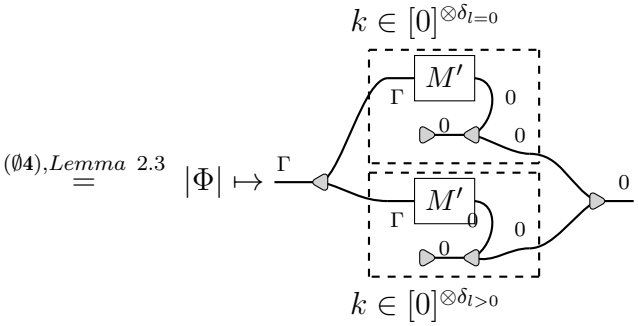


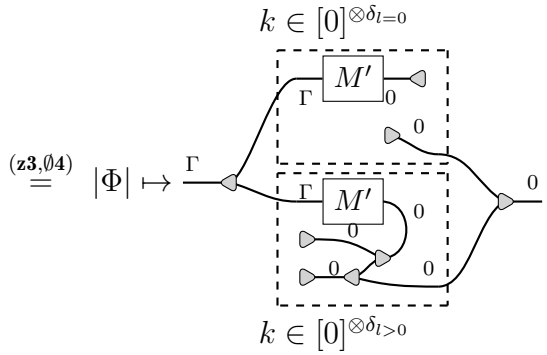


$= [\![\texttt{ifz}\ N_1\ \texttt{then}\ M_1\ ;_v\ M_2\ \texttt{else let}\ x :: xs' = M_1\ \texttt{in}\ x\ ::\ (\texttt{append}\ @(N_1 - 1)\ @N_2\ M_1\ M_2]\!]_{\Phi,\Gamma,\Delta}$

- If $M = \texttt{drop}\ @N'\ M'$ and $N = \texttt{ifz}\ N'\ \texttt{then}\ M'\ ;\ \star\ \texttt{else let}\ x :: xs' = M'\ \texttt{in}\ x\ ;\ \texttt{drop}\ @(N'-1)\ xs'$. Let $l = \lfloor N' \rfloor_\Phi (|\Phi|)$,

$[\![\texttt{drop}\ @N'\ M']\!]_{\Phi,\Gamma} = |\Phi| \mapsto$ 





$= [\![\texttt{ifz}\ N'\ \texttt{then}\ M'\ ;\ \star\ \texttt{else let}\ x :: xs' = M'\ \texttt{in}\ x\ ;\ \texttt{drop}\ @(N'-1)\ xs']\!]_{\Phi,\Gamma}$

- If $M \to N$ is an internal reduction of a translatable term, then the diagrams are equivalent under the inductive hypothesis.
- If $M \to N$ is an internal reduction of an evaluable term, then the diagrams are equivalent under the inductive hypothesis and Lemma 3.3. □

3.4 Application example: QFT

The Quantum Fourier Transform is an algorithm used extensively in quantum computation, notably as part of Shor's algorithm for integer factorization [56]. The QFT function operates generically over n-qubit states and when compiled into a circuit produces operations with $\mathcal{O}(n^2)$ quantum gates. In this section we present an encoding of the algorithm as a λ_D term, followed by the translation into a family of constant-sized diagrams.

The following presentation divides the algorithm into three parts. We present for each part its encoding as a lambda term and as a Quipper program using our library of primitives, followed by the translation into SZX diagram families. In the terms, we use $n \ldots m$ as a shorthand for `range` $@n\ @m$. For the translations we omit the wire connecting the function inputs to the right side of the graphs for clarity and eliminate superfluous gathers and splitters using rules (**sg**) and (**gs**).

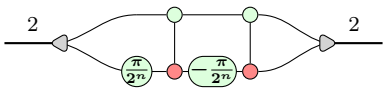
The first function *crot* applies a controlled rotation over a qubit with a parametrized angle. For the Quipper program, we use directly a controlled rotation primitive.

$\text{crot} : (n : \texttt{Nat}) \to (\texttt{Q} \otimes \texttt{Q}) \multimap (\texttt{Q} \otimes \texttt{Q})$

$$\begin{aligned}\text{crot} := \lambda' n^{\texttt{Nat}}.\lambda q s^{\texttt{Q}\otimes\texttt{Q}}.\\ &\texttt{let } c^{\texttt{Q}} \otimes q^{\texttt{Q}} = qs \texttt{ in}\\ &\texttt{let } c^{\texttt{Q}} \otimes q^{\texttt{Q}} = \texttt{CNOT } c\ (\texttt{Rz } @2^n\ q) \texttt{ in}\\ &\texttt{CNOT } c\ (\texttt{Rz}^{-1}\ @2^n\ q)\end{aligned}$$

```
crot : ! (n : Nat) -> Qubit * Qubit -> Qubit * Qubit
crot n q = let (q',c) = q in flip $ R n q' c
```

The term can be directly translated as by encoding each quantum gate.

$[\![\text{crot}]\!] = \ n \mapsto$ 

The second function *apply_crot* operates over the last $n - k$ qubits of an n-qubit state by applying a Hadamard gate to the first one and then using it as target of

successive *crot* applications using the rest of the qubits as controls. Notice that, in contrast to the presented lambda terms, the type checker implementation requires explicit encodings of the Leibniz equalities between parameter types.

apply_crot : $(n : \texttt{Nat}) \to (k : \texttt{Nat}) \to \texttt{Vec}\ n\ \texttt{Q} \multimap \texttt{Vec}\ n\ \texttt{Q}$

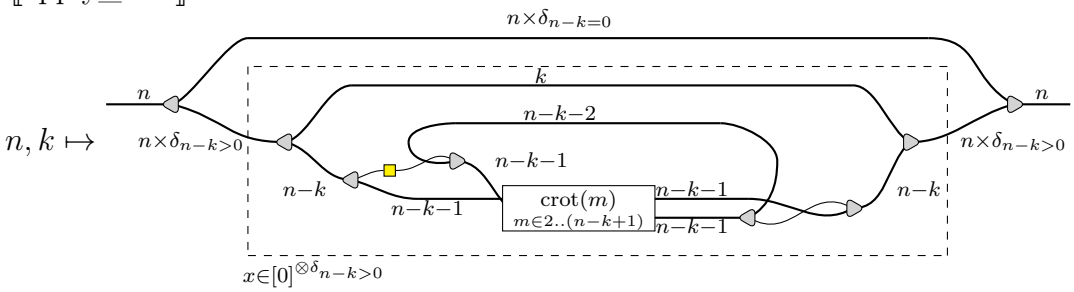
$$
\begin{aligned}
\text{apply\_crot} := \lambda' n^{\texttt{Nat}}.\ \lambda' k^{\texttt{Nat}}.\ \lambda qs^{\texttt{Vec}\ n\ \texttt{Q}}.\\
&\texttt{ifz}\ (n-k)\ \texttt{then}\ qs\ \texttt{else}\\
&\texttt{let}\ h^{\texttt{Vec}\ k\ \texttt{Q}} \otimes qs'^{\texttt{Vec}\ n-k\ \texttt{Q}} = \texttt{split}\ @k\ @(n-k)\ qs\ \texttt{in}\\
&\texttt{let}\ q^{\texttt{Q}} \otimes cs^{\texttt{Vec}\ n-k-1\ \texttt{Q}} = qs'\ \texttt{in}\\
&\texttt{let}\ fs^{\texttt{Vec}\ (n-k-1)\ (\texttt{Q}\otimes\texttt{Q}\multimap\texttt{Q}\otimes\texttt{Q})}\\
&\qquad = (\texttt{for}\ m^{\texttt{Nat}}\ \texttt{in}\ 2..(n-k+1)\ \texttt{do}\ \text{crot}\ @m)\ \texttt{in}\\
&\texttt{let}\ cs'^{\ (\texttt{Vec}\ n-k-1\ \texttt{Q})} \otimes q'^{\ \texttt{Q}} = \texttt{accuMap}\ fs\ (H\ q)\ cs\ \texttt{in}\\
&\text{concat}\ h\ (q' : cs')
\end{aligned}
$$

```
-- Specify types to help the typechecker
applyCrot_aux : ! (n : Nat) -> Qubit -> Qubit -> Qubit * Qubit
applyCrot_aux n ctrl q = crot n (q, ctrl)

-- Apply a CROT sequence to a qubit register, ignoring the first k qubits.
applyCrot : ! (n k : Nat) -> Vec Qubit n -> Vec Qubit n
applyCrot n k qs =
  let WithEq r e = inspect (minus n k)
  in case r of
      Z -> qs
      S n' ->
        let
          -- e : Eq Nat (S n') (minus n k)
          -- e' : Eq Nat (add k (S n')) n
          e' = trans (symAdd k (S n')) $ minusPlus n n' k $ sym (minus n k) e
          -- qs' : Vec Qubit (minus n k)
          qs' = subst (\m -> Vec Qubit m) (sym (add k (S n')) e') qs
          (head, qs') = split k (S n') $ qs'
          (q,ctrls) = chop qs'
          -- fs : Vec (Qubit -> Qubit -> Qubit * Qubit) (minus n' Z)
          fs = foreach (\k -> applyCrot_aux (S(S k))) $ 0..n'
          -- fs : Vec (Qubit -> Qubit -> Qubit * Qubit) Z
          eq = sym n' $ minusZ n'
          fs = subst (\m -> Vec (Qubit -> Qubit -> Qubit * Qubit) m) eq fs
          (ctrls', q') = accumap fs (H q) ctrls
        in subst (\m -> Vec Qubit m) e' $ append head (VCons q' ctrls')
```

The translation of *apply_crot* only includes the diagram for *crot* once, and it iterates it using the accumap construction.

$[\![\text{apply\_crot}]\!] =$



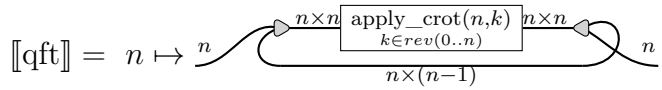
Finally, *qft* repeats *apply_crot* for all values of k.

$\text{qft} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ \texttt{Q} \multimap \texttt{Vec}\ n\ \texttt{Q}$

$\text{qft} := \lambda' n^{\texttt{Nat}}.\lambda qs^{\texttt{Vec}\ n\ \texttt{Q}}.\texttt{compose}$
$\quad (\texttt{for}\ k^{\texttt{Nat}}\ \texttt{in}\ \text{reverse\_vec}\ @(0..n)\ \texttt{do}\ \lambda qs'^{\texttt{Vec}\ n\ \texttt{Q}}.\text{apply\_crot}\ @n\ @k\ qs')\ qs$

```
-- Required for the type checker to derive the second !
qft_aux : ! (n : Nat) -> ! (k : Nat) -> Vec Qubit n -> Vec Qubit n
qft_aux n head_size qs = applyCrot n head_size qs

qft : ! (n : Nat) -> Vec Qubit n -> Vec Qubit n
qft n qs = let f = qft_aux n in compose' (foreach f $ reverse_vec (0..n)) qs
```



Notice that, in contrast to a quantum circuit encoding of QFT which required a quadratic number of gates, the produced SZX diagram's size does not depend on the number of qubits n.

4

Optimizing hybrid quantum-classical circuits with the $\mathrm{ZX}_{\perp}$ calculus

In general, in any resource-constrained system it is important to make an efficient use of the available resources during its operation. In the context of quantum computers and especially for NISQ machines, the execution of a quantum program is heavily constrained by the available number of qubits and the inherent noise of the quantum operation which limit the number of operations that can executed.

As quantum circuits became the standard representation for running operations on quantum machines, the research on optimization procedures has been mostly focused on methods based on gate substitution or optimization of phase polynomials [2, 43]. One notable exception is the one introduced by Duncan et al. in 2019 [33] in which they use the ZX calculus as an alternative intermediate representation to apply granular rewriting rules that ignore the boundaries between gates. In this chapter we will refer to their procedure as the *Clifford optimization* algorithm.

Most common optimization strategies for quantum circuits focus exclusively on pure operations, without contemplating any hybrid quantum-classical structure. Although pure quantum circuits can be effectively optimized in isolation from the classical segments, it is natural to look at the more general problem of optimization considering the interaction with the environment in addition to the pure quantum fragments. In this chapter we present an optimization procedure for hybrid quantum-classical circuits using the $\mathrm{ZX}_{\perp}$ calculus.

We first introduce in Section 4.1 the subset of *graph-like* $\mathrm{ZX}_{\perp}$ diagrams that we will require during the process and describe a translation from hybrid circuits into such diagrams in Section 4.2. In Section 4.3 we describe the original ZX based optimization

procedure for pure quantum circuits by Duncan et. al. which serves as a basis for our definition.

Then, in Section 4.4 we begin the discussion of our main contribution in the form of an extension of the pure optimization procedure to hybrid quantum-classical circuits using the $ZX_{\Downarrow}$ calculus. Section 4.5 further improves the optimization by introducing a methodology to find additional rewriting opportunities. Section 4.6 describes an efficient algorithm to apply our optimization and finally Section 4.7 formalises an extraction back into quantum circuits based on the Clifford procedure.

Section 4.8 includes an example execution of the extraction algorithm, and finally Section 4.9 discusses the implementation and benchmarking of our algorithm.

The results described in this chapter have been presented at the APLAS 2021 conference [11].

4.1 Graph-like diagrams and focused gFlow

Our plan for the optimization involves transforming the quantum circuits into $ZX_{\Downarrow}$ diagrams and extracting them back into circuits. The translation into $ZX_{\Downarrow}$ diagrams is a straightforward operation; the usual set of Clifford+T quantum gates has a direct representation as compact ZX diagrams and the measurements and qubit initializations can be directly represented with the ground operator. On the other hand, extracting back the circuits can be a complex task.

Recently, de Beaudrap et al. [29] proved that the problem of extracting a quantum circuit from an arbitrary ZX diagrams is #P-hard. However, some efficient extraction algorithms have been described for $ZX_{\Downarrow}$ diagrams which admit a special partial order between spiders called *focused gFlow*.

To define the gFlow property, we must first introduce the special form of *graph-like* diagrams [33]. We say that a pure ZX diagram is graph-like if it only contains Z-spiders internally connected by Hadamard wires, it has no parallel edges nor self-loops, and no spider is connected to more than one input or output. We define the graph-like form for $ZX_{\Downarrow}$ diagrams in a similar manner, additionally restricting the spiders to be connected to at most one $\Downarrow$, and restricting the connections between $\Downarrow$ -spiders and the inputs and outputs.

When translating a quantum circuit into a $ZX_{\Downarrow}$ diagram, we will first generate diagrams in a weaker form allowing a node to connect to an input, a $\Downarrow$, and any number of outputs simultaneously. We will then rewrite these *weakly graph-like* diagrams to transform the final translation result into the strict version afterwards.

The string and weak graph-like forms are formally defined as follows:

Definition 4.1 A $ZX_{\Downarrow}$ diagram is *graph-like* (respectively *weakly graph-like*) when:

1. All spiders are Z-spiders.
2. Z-spiders are only connected via Hadamard edges.
3. There are no parallel Hadamard edges or self-loops.
4. There is no pair of connected $\fatslash$ -spiders.
5. Every input, output, or $\fatslash$ is connected to a Z-spider.
6. Every Z-spider connected to a $\fatslash$ has phase 0.
7. Every Z-spider is connected to at most one input, one output, or one $\fatslash$ (at most one input and at most one $\fatslash$).

Lemma 4.2 Every $\mathrm{ZX}_{\fatslash}$ diagram is equivalent to a weakly graph-like $\mathrm{ZX}_{\fatslash}$ diagram.

Proof Duncan et al. [33, Lemma 3.2] proved that any pure-ZX diagram is equivalent to a graph-like one. When applied to a $\mathrm{ZX}_{\fatslash}$ diagram, their procedure results in a diagram that satisfies the conditions 1, 2, and 3 of Definition 4.1, and where every input and output is connected to a different Z-spider.

From this point, we can apply rule $(\boldsymbol{l})$ to eliminate Hadamards connected to $\fatslash$ generators, satisfying condition 5. We then apply rule $(\boldsymbol{n})$ to disconnect wires between $\fatslash$ -spiders, rule $(\boldsymbol{gg})$ to eliminate duplicated $\fatslash$ connected to the same spider, and rules $(\boldsymbol{m})$ and $(\boldsymbol{f})$ to remove the phase of $\fatslash$ -spiders.

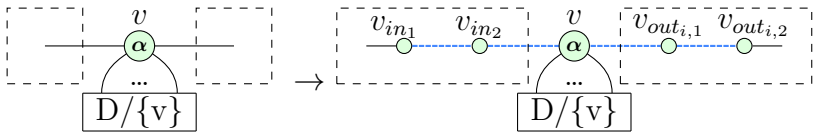
The resulting diagram satisfies all conditions of the weakly graph-like form. □

Lemma 4.3 There exists an algorithm to transform an arbitrary $\mathrm{ZX}_{\fatslash}$ diagram into an equivalent strictly graph-like diagram.

Proof By adding identity spiders to the inputs and outputs.

The strict graph-like condition limits to 1 the number of inputs, outputs, and ground generators connected to each spider, in addition to the weakly graph-like restrictions. By Lemma 4.2, the diagram can be rewritten into an equivalent weakly graph-like diagram D. We describe an algorithm which modifies D to comply with the additional restriction.

For each spider v in D connected to at least two inputs, outputs or $\fatslash$ generators, add two Z-spiders and Hadamard wires to each connected input and output as follows.



(4.4)

Notice that v may have at most one connected input. The introduced spiders correspond to identity operations, and therefore the procedure generates a strictly graph-like diagram equivalent to D. □

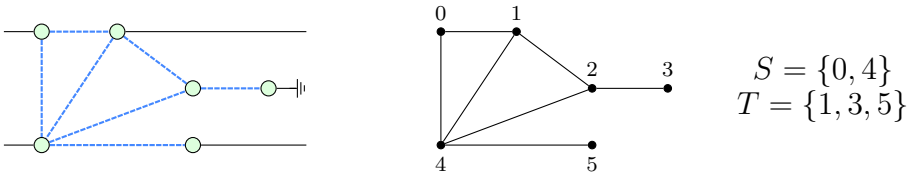
Once a diagram is in a weakly graph-like form, all its spiders as well as all its internal connections are of the same kind. We can refer to its underlying structure as a simple undirected graph, marking the nodes connected to inputs and outputs. In addition, $\frac{1}{\pi}$ generators or the $\frac{1}{\pi}$ -spiders connected to them can be seen as outputs discarding information into the environment. This is known as the underlying open graph of a diagram.

Definition 4.5 An *open graph* is a triple (G, S, T) where $G = (V, E)$ is an undirected graph, and $S \subseteq V$ is a set of *sources* and $T \subseteq V$ is a set of *sinks*. For a weakly graph-like ZX$_{\frac{1}{\pi}}$diagram D, the *underlying open graph* $G(D)$ is the open graph whose vertices are spiders D, whose edges correspond to Hadamard edges, whose set S is the subset of spiders connected to the inputs of D, and whose set T is the subset of spiders connected to the outputs of D or to ground generators.

The underlying open graph of a ZX diagram produced from our translation of quantum circuits satisfies a graph-theoretic invariant called *focused gFlow* [53]. This structure —originally conceived for graph states in measurement-based quantum computation— gives a notion of flow of information and time on the diagram. It will be required to guide the extraction strategy in Section 4.7.

Definition 4.6 Given an open graph G, a *focused gFlow* $(g, \prec)$ on G consists of a function $g : \overline{T} \to 2^{\overline{S}}$ and a partial order $\prec$ on the vertices V of G such that for all $u \in \overline{T}$, $\mathsf{Odd}_G(g(u)) \cap \overline{T} = \{u\}$ and $\forall v \in g(u), u \prec v$ where $2^{\overline{S}}$ is the powerset of $\overline{S}$ and $\mathsf{Odd}_G(A) \coloneqq \{v \in V(G) \mid |N(v) \cap A| \equiv 1 \bmod 2\}$ is the *odd neighbourhood* of A.

Example 4.7 Consider the following graph-like diagram and associated open graph, where each node has been labelled with an index.



The natural label ordering $i < j$ and the function

$$g(0) := \{1, 2, 3\} \qquad g(2) := \{3\} \qquad g(4) := \{5\}$$

form a valid focused gFlow on this open graph.

4.2 Translation of hybrid circuits

We describe our translation from hybrid quantum-classical circuits into strictly graph-like $\mathrm{ZX}_{\Cup}$ diagrams by steps. First, we translate each individual gate directly into a weakly graph-like diagram and connect them with regular (non-Hadamard) wires. We define this translation $T(\cdot)$ form circuits into $\mathrm{ZX}_{\Cup}$diagrams by inductively translating the gates as described in Table 4.1 immediately followed by the application of the spider fusion rule (***f***) and rules (***gg***) and (***fh***) to remove all regular wires, duplicated $\Cup$ generators, and parallel Hadamard wires, ensuring that the final combined diagram is in a weakly graph-like form.

Notice that the translation maps both classical and quantum wires to regular $\mathrm{ZX}_{\Cup}$ diagram edges. We keep track of which inputs and outputs of the diagram were connected to classical wires and introduce $\Cup$ generators for the operations that interact with the environment. In Chapter 5 we present a method to detect the sections of the final circuit that can be implemented as classical operations by looking at the classical inputs/outputs and the $\Cup$ generators, independently of which wires were originally classical.

In this work we restrict the inputs to circuits with classical parity logic, choosing not to include AND gates. The $\mathrm{ZX}_{\Cup}$ calculus is able to represent these operations in what equates to the Clifford+T decomposition of the Toffoli gate, in an approach that introduces multiple T-gates and CNOT gates to the circuit [63]. The multiple spiders would be dispersed around the diagram during the optimization step, potentially breaking the pattern formed by the AND gate and replacing it with multiple quantum operations. This could produce the unexpected result of introducing expensive quantum operations in an originally pure classical circuit. Therefore, we keep the restriction during the rest of this chapter.

Lemma 4.1 The $\mathrm{ZX}_{\Cup}$ diagram resulting from the translation $T(\cdot)$ is weakly graph-like.

Proof By induction on the circuit construction.

Notice that all the translation rules aside from the serial composition generate weakly graph-like diagrams trivially.

For the serial composition, notice that both $T(C)$ and $T(C')$ are weakly graph-like by the induction hypothesis and therefore all spiders in the resulting diagram are Z-spiders, all inputs and outputs are connected to Z-spiders, no two inputs are connected to the same spider, and all spiders connected to $\Cup$ -generators have phase 0. The internal edges added by the composition will therefore connect two green spiders, which will be merged by the fusion rule application.

The fusion step may create spiders connected to two ground generators, one of which is removed by the application of rule (***gg***). It may also generate parallel

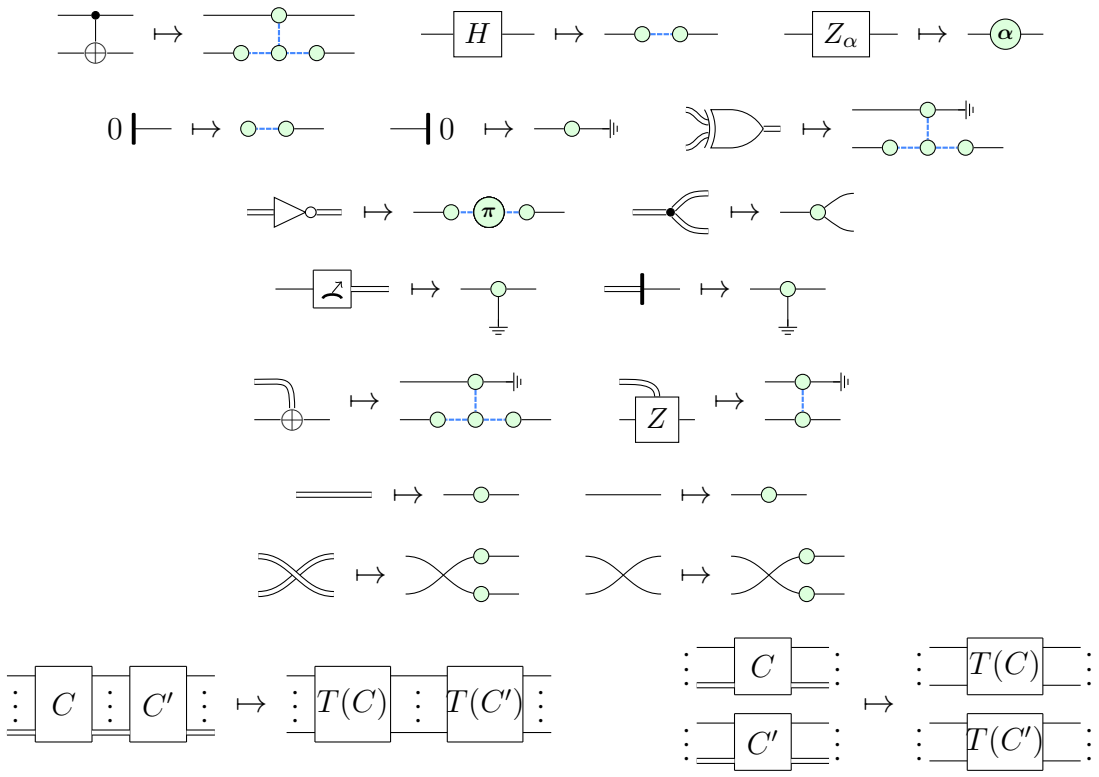


Table 4.1: The $T(\cdot)$ translation from hybrid quantum-classical circuits into $ZX_{\perp}$ diagrams.

Hadamard wires, which are removed by the application of rule (***fh***).

Therefore, the translation generates weakly graph-like $ZX_{\perp}$-diagrams. □

After the translation, we can apply Lemma 4.3 to obtain a strictly graph-like diagram. This step essentially separates the $\perp$ generators from the inputs and outputs, allowing the optimization procedure to move them around and let them interact with other parts of the diagram.

The underlying graph of a diagram produced by the translation admits a gFlow. To prove this, we first use a weaker version of the focused gFlow invariant called *causal flow* [53].

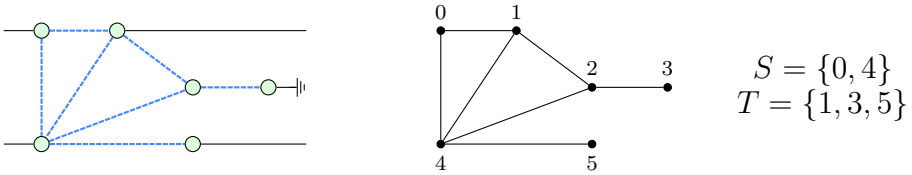
Definition 4.2 Given an open graph G, a *causal flow* $(f, \prec)$ on G consists of a function $f : \overline{T} \to \overline{S}$ and a partial order $\prec$ on the set V satisfying the following properties:

1. $f(v) \sim v$

2. $v \prec f(v)$
3. If $u \sim f(v)$ and $u \neq v$ then $v \prec u$

where $u \sim v$ if u is connected to v in the graph, and $\overline{A} = V(G)/A$.

Example 4.3 Consider the graph-like diagram and associated open graph from Example 4.7.



$$S = \{0, 4\}$$
$$T = \{1, 3, 5\}$$

The natural label ordering $i < j$ and the function

$$f(0) := 1 \qquad f(2) := 3 \qquad f(4) := 5$$

form a valid causal flow on this open graph.

Duncan et al. prove the following lemma relating both flow properties:

Lemma 4.4 ([33, Theorem B.3]) If G admits a causal flow, then there exists a valid focused gFlow for G.

Lemma 4.5 If D is a weakly graph-like $\mathrm{ZX}_{\frac{1}{2}}$ diagram and $G(D)$ admits a causal flow, the strictly graph-like diagram generated by the algorithm described in Lemma 4.3 admits a causal flow.

Proof Let $(f_D, \prec_D)$ be a valid causal flow for D and let D' be the resulting diagram after applying rule 4.4 over a node v. We construct a causal flow for D' by defining a function $f_{D'}$ and relation $\prec_{D'}$ as the minimal objects such that

- $f_{D'} \supseteq f_D$ and $\prec_{D'} \supseteq \prec_D$.
- If v is connected to an input in D, $f_{D'}(v_{in_1}) = v_{in_2}$, $f_{D'}(v_{in_2}) = v$, and (v_{in_1}, v_{in_2}), $(v_{in_2}, v) \in \prec_{D'}$.
- If v is connected to at least one output in D and v is not a $\frac{1}{2}$-spider, $f_{D'}(v) = v_{out_{1,1}}$.
- For each output i connected to v in D, $(v, v_{out_{i,1}}), (v_{out_{i,1}}, v_{out_{i,2}}) \in \prec_{D'}$ and $f_{D'}(v_{out_{i,1}}) = v_{out_{i,2}}$.

Notice that $(f_{D'}, \prec^+_{D'})$ is a valid focused flow for D', where $\prec^+_{D'}$ is the transitive closure of $\prec_{D'}$. Since the rule 4.4 preserves the focused flow, the resulting diagram after successive application admits a causal flow. □

Lemma 4.6 If C is a hybrid quantum-classical circuit and D is the graph-like $ZX_{\overline{=}}$ diagram obtained from the translation $T(C)$ and Lemma 4.3, then $G(D)$ admits a focused gFlow.

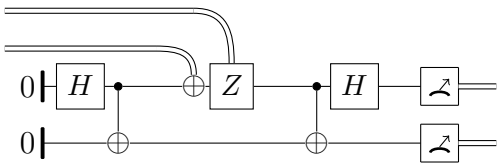
Proof By induction on C.

By Lemma 4.4, it suffices to prove that $G(D)$ admits a causal flow. We proceed by induction on the construction of C.

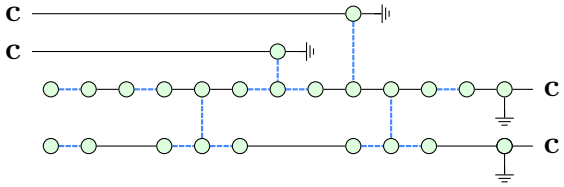
- Notice that the translation of each base constructor cannot be further simplified by rules $(\boldsymbol{f})$, $(\boldsymbol{gg})$, or $(\boldsymbol{fh})$, and the underlying open graph trivially admits a causal flow.
- If $C = C_1 \otimes C_2$, let $D_i = T(C_i)$. Since the two circuits are not connected after the composition, they do not interact via the rules $(\boldsymbol{f})$, $(\boldsymbol{gg})$, or $(\boldsymbol{fh})$, and therefore $D = D_1 \otimes D_2$. By the induction hypothesis $G(D_1)$ and $G(D_2)$ admit some causal flow $(f_1, \prec_1)$ and $(f_2, \prec_2)$ respectively. Then, $(f_1 \cup f_2, \prec_1 \cup \prec_2)$ is a causal flow for $G(D)$.
- If $C = C_1 \otimes C_2$, let $D_i = T(C_i)$. By the induction hypothesis $G(D_1)$ and $G(D_2)$ admit some causal flow $(f_1, \prec_1)$ and $(f_2, \prec_2)$ respectively. Notice that rule $(\boldsymbol{f})$ will be applied between each output of C_1 and the connected inputs in C_2. Let f'_1 be a function and $\prec'_1$ a relation such that for each vertex v in $G(D_1)$ and corresponding non-empty set of inputs $\{u_1, \ldots, u_k\}$ in $G(D_2)$, $\forall v' st.\ f_1(v') = v, f'_1(v') = u_1$ and $\forall v' st.\ v' \prec_1 v, \{(v', u_i)\}_{i=1}^k \subseteq \prec'_1$. Notice that since D_2 is weakly graph-like, u_1 has exactly one corresponding sink node in $G(D_1)$. Additionally, for all v_1, v_2 non-output nodes in $G(D_1)$ let $f'_1(v_1) = v_2$ if $f_1(v_1) = v_2$ and let $v_1 \prec'_1 v_2$ if $v_1 \prec_1 v_2$. Then, $(f'_1 \cup f_2, (\prec'_1 \cup \prec_2)^+)$ is a causal flow for $G(D)$.

Then, by Lemma 4.5, the application of Lemma 4.3 preserves the existence of a causal flow. □

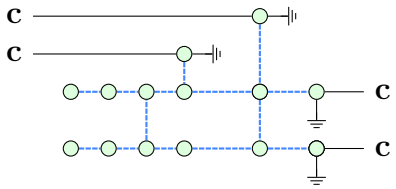
Example 4.7 We demonstrate the translation of the following quantum-classical circuit representing a superdense coding scheme into a $ZX_{\overline{=}}$ diagram.



We start by applying the translation rules from Table 4.1 to obtain the following ZX$_{\frac{}{}}$ diagram with labels corresponding to whether each input or output was a classical or quantum wire.



To obtain the final weakly graph-like diagram, we apply some spider-fusion steps on the simple wires to obtain a weakly graph-like diagram.

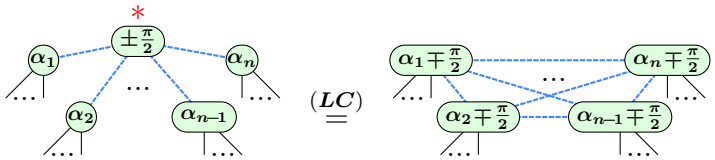


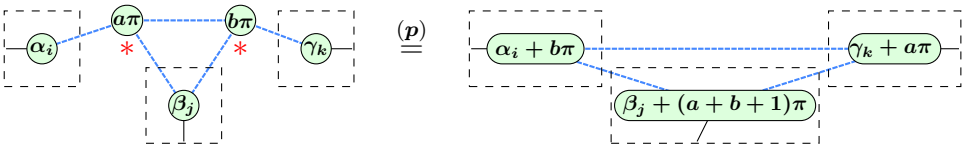
Notice that the $\frac{}{}$ -spiders connected to the inputs and outputs prevent it from being in graph-like form. However, we can easily rewrite the diagram into a strictly graph-like form using Lemma 4.3, adding identity spiders to the inputs and outputs where required.

4.3 Graph-theoretical circuit optimization

We can now describe the Clifford optimization procedure for pure ZX diagrams introduced by Duncan et al. [33]. In their work, they introduce a number of rewriting steps that preserve the gFlow in a graph-like ZX-diagram and define a reduction strategy that decreases the number of nodes until it reaches a local minimum.

The first rewriting rule, called *local complementation* and referred as (***LC***), is able to remove a proper Clifford node (with angle $\pm\frac{1}{2}\pi$) from the diagram by moving its phase to its neighbours and inverting the connectivity between them. The second rule is called *pivoting* (***p***)and performs a similar operation on a pair of connected Pauli nodes while switching the connectivity between the two sets of exclusive neighbours to each node, and the set of common neighbours. The rules have been proven to preserve the gFlow in a graph-like diagram [33]. They are depicted as follows, where $*$ denotes vertices to which the rules are applied,





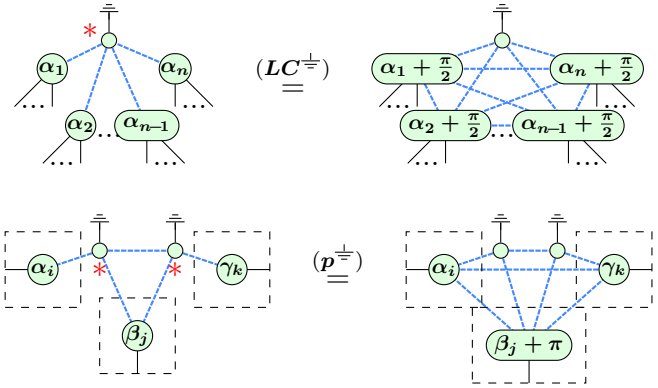
Both rules effectively reduce the size of the diagram by at least one node on each application. The optimization strategy finds matches for both rules and applies them repeatedly until the diagram cannot be further reduced. The resulting diagrams may not necessarily be globally minimal. Due to the interactions between the rules, it is possible for two different orderings of reductions to produce different results. Nevertheless, the optimization procedure has been shown to produce sufficiently minimal circuits when compared to other state of the art algorithms on standardised sets of benchmarks [33].

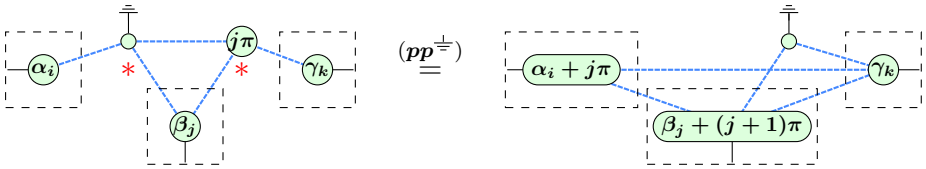
This optimization method has been used as a basis for the construction of additional simplification procedures, as in the case of the T-gate reduction algorithm by Kissinger and van de Wetering [48] and the extended procedure by Backens et al. [6]. In the following section we will present our extension of this method to $ZX_{\doteq}$ diagrams.

4.4 Grounded ZX optimization

Similarly to the Clifford optimization presented in Section 4.3, our simplification strategy for $ZX_{\doteq}$ diagrams is based on eliminating nodes from the diagram by systematically applying a number of rewriting rules while preserving the existence of a focused gFlow.

The pure ZX rules presented in Section 4.3 can be applied directly in $ZX_{\doteq}$ diagrams when the target spiders are not connected to a $\doteq$ generator. For the cases where some of the target spiders are $\doteq$ -spiders we introduce the following additional rules.

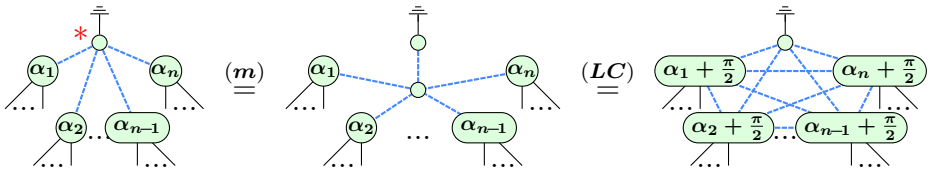




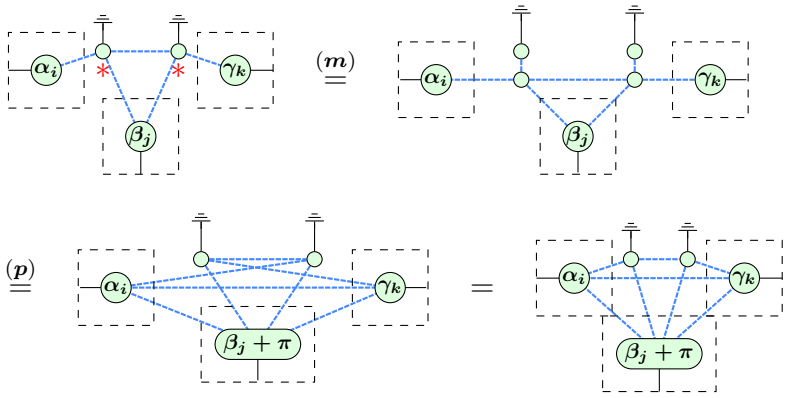
Notice that both rules ($\boldsymbol{LC}^{\doteq}$) and ($\boldsymbol{p}^{\doteq}$) do not decrease the number of spiders in the diagram. As such, we will focus on rule ($\boldsymbol{pp}^{\doteq}$) for our optimization.

These rules can be derived using the standard ZX$_{\doteq}$ rules as follows.

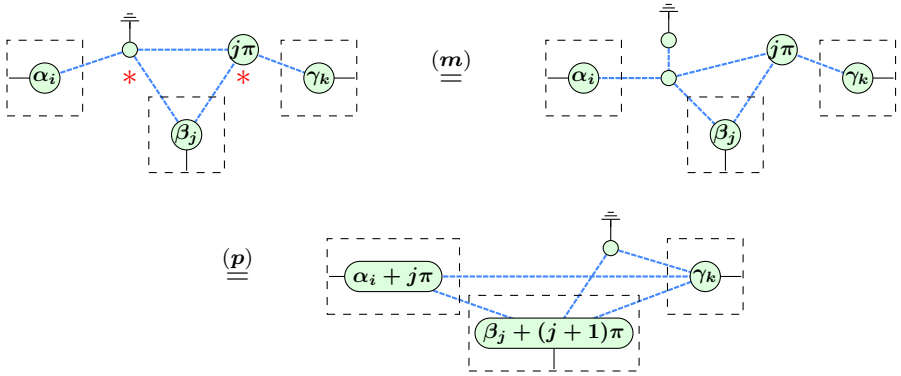
- Rule ($\boldsymbol{LC}^{\doteq}$):



- Rule ($\boldsymbol{p}^{\doteq}$):



- Rule ($\boldsymbol{pp}^{\doteq}$):



If $(\boldsymbol{pp}^{\Downarrow})$ is applied with a non-$\Downarrow$ spider connected to a boundary, the rule produces a $\Downarrow$-spider connected to an input or output thus requires adding an identity operation as described in Lemma 4.3 to preserve the graph-like property. Since in this case we add additional nodes to the graph, we will only apply rule $(\boldsymbol{pp}^{\Downarrow})$ on a boundary spider if it can be followed by another node-removing rule.

Additionally, we use rules $(\boldsymbol{ml})$ and $(\boldsymbol{k})$ directly to remove nodes in the diagram when there are $\Downarrow$-spiders with degree 1 or 0 in the graph, respectively.

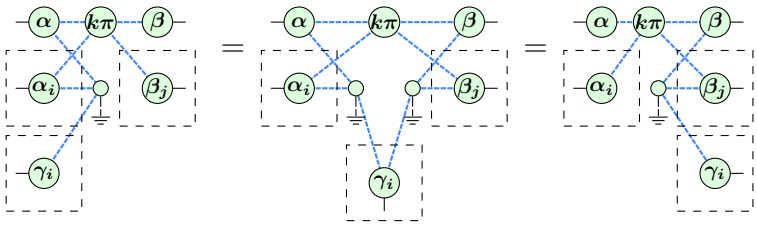
Lemma 4.1 If the non-$\Downarrow$ spider in the left-hand side of the discarding rule $(\boldsymbol{ml})$ is not connected to an output or input, then applying the rule over a graph-like diagram D preserves the existence of a focused gFlow.

Proof If the non-$\Downarrow$ spider in the left-hand side is not connected to an input or output of the diagram, then applying the rule does not break the graph-like property of D. The preservation of the gFlow follows from $\Downarrow$-spiders being sinks of the underlying open graph. □

Lemma 4.2 Rules $(\boldsymbol{LC}^{\Downarrow})$, $(\boldsymbol{p}^{\Downarrow})$, and $(\boldsymbol{pp}^{\Downarrow})$ preserve the existence of a focused gFlow.

Proof Notice that rules $(\boldsymbol{LC}^{\Downarrow})$, $(\boldsymbol{p}^{\Downarrow})$, and $(\boldsymbol{pp}^{\Downarrow})$ are defined as compositions of gFlow-preserving rules. □

In addition to the previous rules, we can derive following *ground teleportation* rule from rule $(\boldsymbol{pp}^{\Downarrow})$,



While this rule does not remove any spider, if there is only one β spider then it can be followed by a use of rules $(\boldsymbol{ml})$ and $(\boldsymbol{pp}^{\Downarrow})$ to delete two nodes from the graph. Experimentally, this case rarely happened on the tested circuits and therefore we do not include this rule in our optimization algorithm.

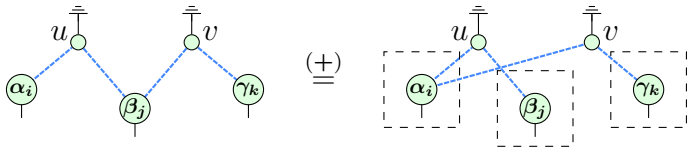
4.5 Ground-cut simplification

The previously introduced rewriting rules require a simplification strategy to apply them. A simple solution used in the Clifford optimization is to try to find a match

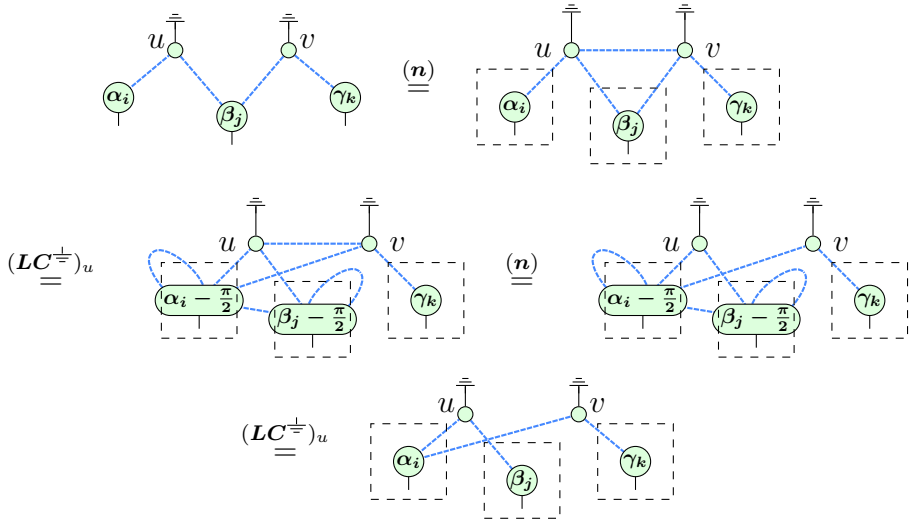
for each rule and apply them iteratively until no more matches are available. Alternatively, we can apply non-reducing rewriting rules aimed at exposing hidden optimization opportunities. To that end, we describe a strategy that can find additional rule matches by operating on the biadjacency matrix between the $\doteq$ -spiders and the non-$\doteq$ spiders.

Definition 4.1 The *ground-cut* of a graph-like $\mathrm{ZX}_{\doteq}$ diagram D is the cut resulting from splitting the $\doteq$ and non-$\doteq$ spiders in $G(D)$.

Since the diagram is graph-like, there are no internal wires in the $\doteq$ partition. Given a $\mathrm{ZX}_{\doteq}$ diagram D, we denote M_D the biadjacency matrix of its ground-cut, where rows correspond to $\doteq$ -spiders and columns correspond to non-$\doteq$ spiders. We can apply all elementary row operations on the matrix by rewriting the diagram. The addition operation between the rows corresponding to the $\doteq$ -spider u and the $\doteq$ -spider v can be implemented via the following rule.



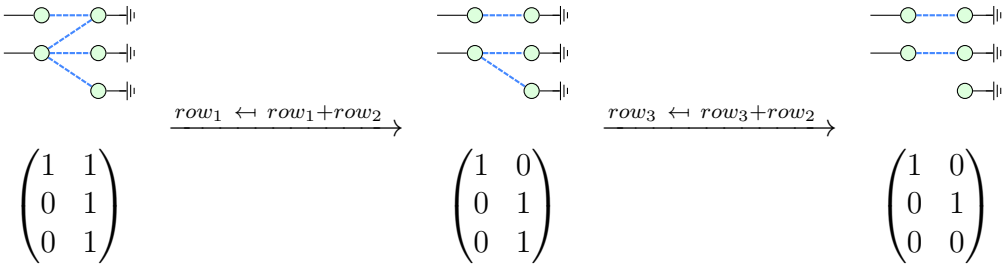
The rule can be derived from the $\mathrm{ZX}_{\doteq}$ equations as follows.



Using the elementary row operations we can apply Gaussian elimination on the ground-cut biadjacency matrix of a graph-like $\mathrm{ZX}_{\doteq}$ diagram, generating in the process an equivalent diagram whose ground-cut biadjacency matrix is in reduced echelon form, where the leading coefficient of each row is strictly to the right of the leading coefficients of the rows above it.

Any row in the ground-cut biadjacency matrix left without non-zero elements after applying Gaussian elimination corresponds to an isolated $\fallingdotseq$ -spider in the diagram that can be eliminated by rule (***k***). If the reduced row echelon form of the biadjacency matrix contains a row with exactly one non-zero element, then that element corresponds to an isolated $\fallingdotseq$ -spider and non-$\fallingdotseq$ spider pair in the diagram and therefore we can apply rule (***ml***) to remove the non-$\fallingdotseq$ spider.

Example 4.2 The following example demonstrates the application of a Gauss elimination over a graph-like diagram. The corresponding ground-cut adjacency matrices are shown below each step.



After the reduction procedure, we can eliminate the isolated $\fallingdotseq$ -spiders using rule (***k***) and use rule (***ml***) on the spiders corresponding to the first two rows.

4.6 The Algorithm

Based on the gFlow-preserving rules presented in Section 4.4 and the strategy from Section 4.5, we define a terminating procedure which turns any graph-like ZX$_{\fallingdotseq}$ diagram into an equivalent *simplified* diagram that cannot be further reduced.

Definition 4.1 A graph-like ZX$_{\fallingdotseq}$ diagram is in simplified form if it does not contain any of the following, up to single-qubit unitaries on the inputs and outputs.

- Interior proper Clifford spiders.
- Adjacent pairs of interior Pauli spiders.
- Interior Pauli spiders adjacent to boundary spiders.
- Interior Pauli spiders adjacent to $\fallingdotseq$ -spiders.
- Degree-1 $\fallingdotseq$ -spiders not connected to input or output spiders.
- Connected components not containing inputs nor outputs.

Example 4.2 The following is a comparison between two equivalent diagrams that satisfy and do not satisfy the simplified form property.

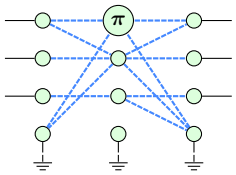


Diagram not in simplified form

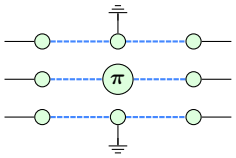


Diagram in simplified form

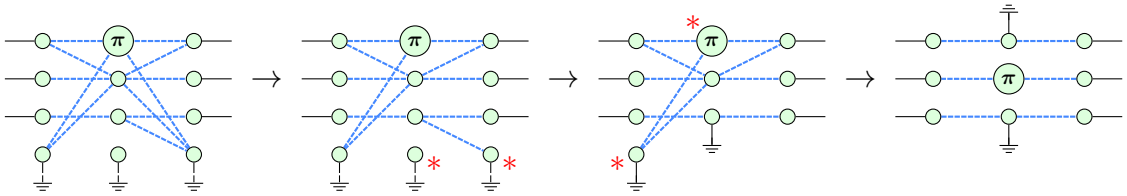
Notice that the first diagram contains interior Pauli spiders adjacent to $\doteq$ -spiders and connected components not containing inputs nor outputs, and therefore it does not comply with the simplified form rules.

We define an optimization algorithm that produces diagrams in simplified form by piggybacking on the pure optimization procedure. This optimization applies the local complementations (***LC***) and pivoting (***p***) rules until there are no interior proper Clifford spiders or adjacent pairs of non-$\doteq$ interior Pauli spiders. After the initial pure simplification, we continue our optimization as follows.

1. Repeat until no rule matches, removing wires between $\doteq$ -spiders and parallel Hadamard connections after each step:
   (a) Run Gaussian elimination on the ground-cut of the diagram as described in Section 4.5.
   (b) Remove the grounds corresponding to null rows with rule (***k***).
   (c) If any row of the biadjacency matrix has a single non-zero element, corresponding to a $\doteq$ -spider connected to a spider v, then:
      i. If v is not a boundary spider, apply rule (***ml***).
      ii. If v is a boundary spider and v is adjacent to a Pauli spider, apply rule (***ml***) immediately followed by the procedure from Lemma 4.3 to make the diagram graph-like again. Then delete the Pauli neighbour using rule ($\boldsymbol{pp}^{\doteq}$), to ensure that the step removes at least one node.
   (d) Apply Pauli spider elimination rule ($\boldsymbol{pp}^{\doteq}$) until there are no Pauli spiders connected to ground spiders.
2. Remove any connected component of the graph without inputs or outputs.

Notice that each cycle the loop reduces the number of nodes in the graph, so this is a terminating procedure. Additionally, since each applied rule preserves the existence of a gFlow the final diagram admits a gFlow.

Example 4.3 We show an an example run of the algorithm on the diagram presented in Example 4.2.



We start by applying Gaussian elimination on the ground-cut of the diagram. This step simplifies the connections of the third $\doteq$ -spider.

Next, after looking at rows of the biadjacency matrix we remove the disconnected $\doteq$ and apply rule (***ml***) on the third $\doteq$ -spider to discard an additional node in the diagram.

Finally, we apply the Pauli elimination rule (***pp***$^{\doteq}$) to produce the final diagram.

4.7 Circuit extraction

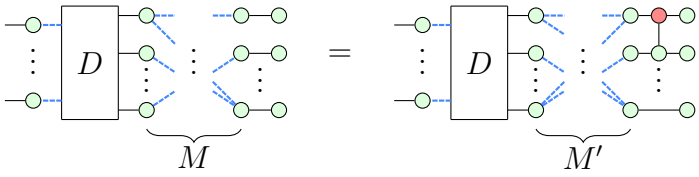
We now describe a general circuit extraction procedure for graph-like $ZX_{\doteq}$ diagrams admitting a focused gFlow into hybrid quantum-classical circuits, by modifying the procedure for pure diagrams from the Clifford optimization. We present the pseudo-code in Algorithm 1 and an example of an execution in Section 4.8.

The algorithm progresses through the diagram from right-to-left, maintaining a set of spiders F, called the *frontier*, which represents the unextracted spiders connected to the extracted segment. Each frontier spider is assigned an output qubit line $\mathsf{Q}(v)$. This set is initially populated by the nodes connected to outputs of the diagram. The strategy is to proceed backwards by steps, adding unextracted spiders into the frontier and deleting some of them to extract operations on the output circuit, in back-to-front order. In the pseudocode, we refer to the set of *input-* and *output-spiders* connected to outputs and inputs of the diagram respectively as O and I.

To find candidate spiders to add to the frontier we apply Gaussian elimination on the biadjacency matrix of the frontier and non-frontier spiders, similarly to the optimization method described in Section 4.5. The gFlow property of the graph ensures that we can always progress by extracting a node; it suffices to look at the set of non-frontier vertices maximal in the order and notice that, after the Gaussian elimination, either we can choose a $\doteq$ -spider from the set, or a non-$\doteq$ spider that has a single connection to the frontier.

We require the following proposition by Duncan et al. to apply the row additions on the graph.

Proposition 4.1 ([33, Proposition 7.1]) For any $\mathrm{ZX}_{\fallingdotseq}$ diagram D, the following equation holds:



where M describes the biadjacency matrix of the relevant vertices, and M' is the matrix produced by adding row 2 to row 1 in M. Furthermore, if the diagram on the left-hand side has a focused gFlow, then so does the right-hand side.

In our pseudocode, the call to CleanFrontier ensures that F only contains phaseless spiders without internal wires. Notice that it preserves the gFlow since it only modifies edges between sink nodes and removes spiders with no other connections. After the while loop terminates, all outputs of the circuit will have been extracted. If there are inputs left unextracted, and since the diagram had a gFlow, we can discard them directly via measurement operations.

Finally, we add any necessary swap operations to map the inputs to the correct lines and insert qubit initializations and measurements at inputs and outputs marked as classical. In Chapter 5 we detail a method to better detect the internal parts of the circuit that can be implemented classically.

In any case, each step of the while loop in Algorithm 1 Line 5 preserves the gFlow of the diagram and we can show that it terminates in at most $|V|$ steps: Indeed, if there are no non-frontier spiders, then a call to CleanFrontier will remove all nodes from the frontier. Moreover, each step of the while loop in Line 5 moves one non-frontier spider to F.

Lemma 4.2 Each step of the while loop in Algorithm 1, Line 5, preserves the gFlow of the diagram.

Proof By Proposition 4.1, the application of Gaussian elimination preserves the gFlow. Then, consider the set of non-frontier spiders maximal in the gFlow order.

If the set contains a non-$\fallingdotseq$ spider u then by definition $g(u) \in \mathsf{F}$. If $\mathsf{Odd}_G(g(u))$ does not contain $\fallingdotseq$ nodes, after the Gaussian elimination there must be a frontier spider v such that it is only connected to u. Therefore, removing v and making u a sink of the diagram does not break the gFlow.

In the other case, since $\fallingdotseq$ -spiders are always sinks of the diagram, promoting them to the frontier does not modify the gFlow of the diagram.

Finally, the call to CleanFrontier does not modify the gFlow. □

Lemma 4.3 The algorithm terminates.

Proof Notice that if there are no non-frontier spiders, then a call to CleanFrontier will remove all nodes from the frontier. Notice that each step of the while loop in Line 5 moves one non-frontier spider to F. Therefore, the loop terminates in at most $|V|$ steps. □

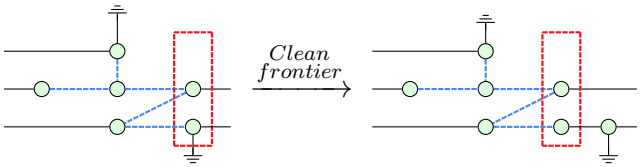
Algorithm 1 Circuit extraction

```
1: function Extraction
2:     F : Set⟨Node⟩ ← O, Q : Map⟨Node, int⟩ ← ∅
3:     for all v ∈ F do Q(v) ← Output connected to v
4:     CleanFrontier(F, Q)
5:     while F ≠ ∅ do
6:         Run Gauss elimination on the frontier biadjacency matrix M (Prop. 4.1)
                                                    ▷ M is in row echelon form
7:         if a row of M has a single non-zero element then
8:             Let u and v be the corresponding non-frontier and frontier node
9:             Q(u) ← Q(v)
10:            Remove v from the diagram and add u to F
11:        else
12:            v ← Arbitrary ⩲ -spider in the neighbourhood of F
13:            Q(v) ← New qubit line id
14:            Extract a classical bit termination on Q(v) and add u to F
15:        CleanFrontier(F, Q)
16:    for all Unextracted v ∈ I do
17:        Q(v) ← Input connected to v
18:        Extract a measurement gate and a classical bit termination on Q(v)
19:        Assign the corresponding input to Q(v)
20: function CleanFrontier(F, Q)
21:    for all v ∈ F do
22:        if v is a ⩲ -spider then Remove the ⩲ , extract a measurement on Q(v)
23:        if v has a phase α ≠ 0 then Set α = 0, extract a Z_α gate on Q(v)
24:        for all u ∈ F, v ∼ u do Remove the wire, extract a CZ gate on Q(v), Q(u)
25:        if v is not connected to any other node then
26:            Remove v
27:            if v ∈ I then assign the input to qubit Q
28:            else extract a |+⟩ qubit initialization on Q(v)
```

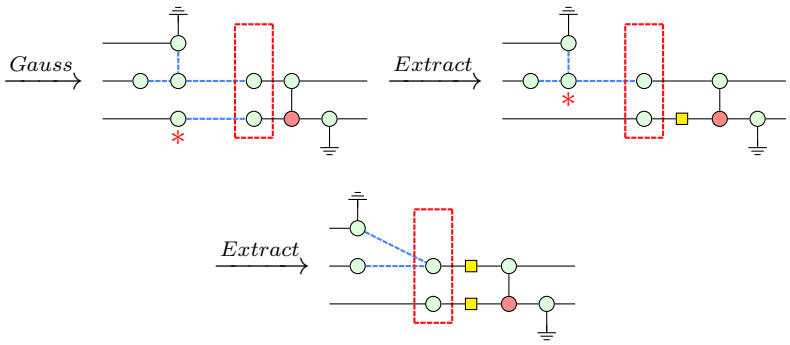
4.8 Extraction examples

We show here an example of running the extraction procedure on a diagram. The frontier set is demarcated with a dashed red box, and the extracted circuit is represented with its ZX equivalent directly connected to the right of the frontier.

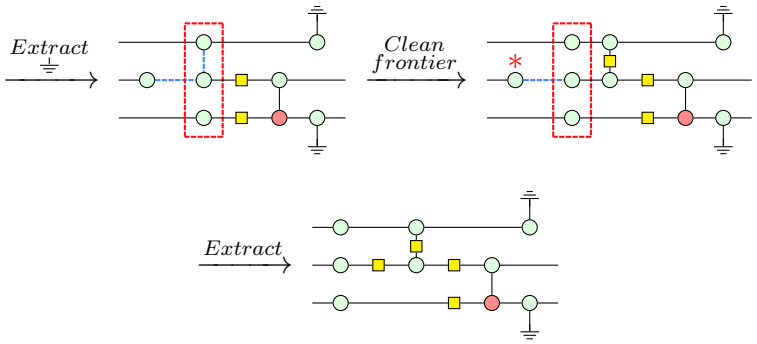
We start with the frontier initialized as the output vertices, and directly extract any ⏚ .



During each step of the algorithm, a maximal non-extracted element in the gFlow order is chosen. Candidates can be chosen efficiently without calculating the gFlow by running Gaussian elimination on the biadjacency matrix between the border and the non-extracted spiders, and mirroring the row-sum operations using a gFlow preserving rewrite rule on the diagram.

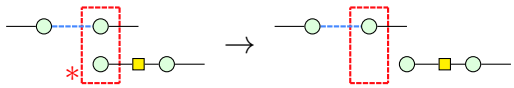


Now, there are no candidate frontier spiders connected to a single unextracted spider. We must therefore extract one of the connected ⏚ -spiders as a qubit termination.



If after any step there are nodes in the frontier that are not connected to any internal spider then they can be removed from the frontier and extracted as a qubit

initialization, as shown in the following example.



4.9 Implementation and benchmarks

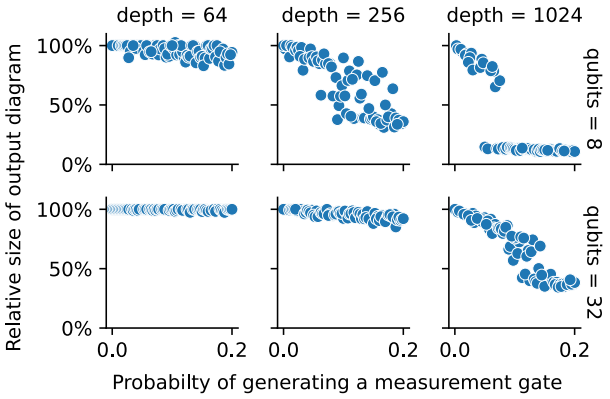
We have implemented each of the algorithms presented in this work as an extension to the open-source Python library *PyZX* [47] by modifying its implementation of ZX diagrams to admit $ZX_{\perp}$ primitives. A repository with the complete code is available on GitHub [10], and parts of the implementation have been upstreamed to the main PyZX repository. We additionally implemented a naive $ZX_{\perp}$ extension of the pure Clifford optimization for comparison purposes, which doesn't use any of our $\perp$ rewriting rules. When applied to pure quantum circuits, our algorithm does not perform additional optimizations after the Clifford procedure and therefore achieves the same benchmark results recorded by Duncan et al. on the circuit set described by Amy et al. [2].

We tested the procedure over two classes of randomly generated hybrid circuits, and measured the size of the resulting diagram as the number of spiders left after the optimization. This metric correlates with the size of the final circuit, although the algorithmic noise caused by the arbitrary choices in the extraction procedure may result in some cases in bigger extracted circuits after a reduction step.

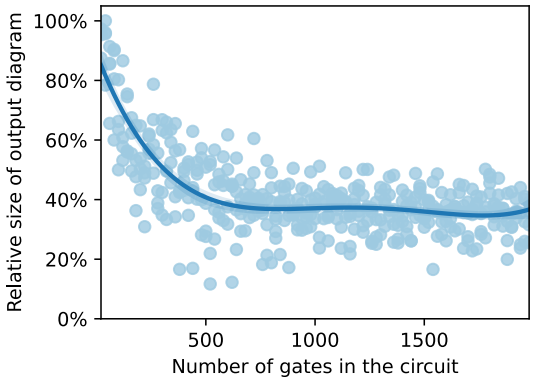
The first test generates Clifford+T circuits with measurements by applying randomly chosen gates from the set {CNOT, S, HSH, T, Meas} over a fixed number of qubits, where Meas are measurement gates on a qubit immediately followed by a qubit initialization. We fix the probability of choosing a CNOT, S, or HSH gate to 0.2 each and vary the probabilities for T and Meas in the remaining 0.4. These circuits present a general worst case, where there is no additional classical structure to exploit during the hybrid circuit optimization.

The second type of generated operations are classical parity-logic circuits. These consist of a number of classical inputs, fixed at 10, where we apply randomly chosen operations from the set {NOT, XOR, Fanout} with probabilities 0.3, 0.3, and 0.4 respectively.

In Figure 4.1 we compare the results of our optimization using the Clifford optimization as baseline. Figure 4.1a shows the reduction of diagram size when running the algorithm on randomly generated Clifford+T circuits with measurement. We vary the probability of generating a measurement gate between 0 and 0.2 while correspondingly changing the probability of generating a T-gate between 0.4 and 0.2 and show the results for different combinations of qubit and gate quantities. We remark that the optimization produces noticeably smaller diagrams once enough $\perp$ generators



(a) Diagram size reduction on Clifford+T circuits with measurements.



(b) Diagram size reduction on parity-logic circuits.

Figure 4.1: Benchmark results on randomly generated diagrams.

start interacting with one another. There is a critical threshold of measurement gate probability, especially visible in the cases with 8 qubits and 1024 gates, where with high probability the outputs of the diagram become disconnected from the inputs due to the $\fatslash$ interactions. This results in our algorithm optimizing the circuits to produce a constant result while discarding their input.

Figure 4.1b shows the comparison of diagram size between our procedure and the Clifford optimization when run over classical parity circuits. The optimization produces consistently smaller diagrams, generally achieving the theoretical minimal number of $\fatslash$ generators, equal to the number of inputs. We further remark that in all of the tested cases the classicalisation procedure was able to detect that all the extracted operations on the optimized parity-logic circuit were classically realisable.

The runtime of our algorithm implementation is polynomial in the size of the circuit. As with the Clifford optimization, the cost of our optimization and extraction

processes is dominated by the Gaussian elimination steps. For the ground-node rewriting rules, our unoptimised implementation is roughly $\mathcal{O}(n^2k^2)$ in the worst case with k being the number of measurement gates and n the number of gates, but in practice it behaves cubically on the number of gates due to the sparseness of the diagrams. The implementation was not developed with a focus on the runtime cost, and some possible optimizations may reduce this bound.

5

Classicality detection using ZX$_{\perp}$ diagrams

In Chapter 4 we described an optimization procedure for quantum circuits that uses the ZX$_{\perp}$ calculus as an alternative representation to perform rewriting operations. During this simplification process we forget the difference between quantum and classical wires and use a single type of edge to represent interconnections in the diagram. This step allows the algorithm to optimize the complete hybrid system as a homogeneous diagram, where operations that can be done either quantumly or classically have a common representation. While this is a necessary step for the optimization, it carries the loss of information on which parts of the circuit can be implemented classically. The extraction procedure from ZX$_{\perp}$ diagrams into hybrid quantum-classical circuits presented in Section 4.7 defaults into always favouring the use of quantum gates. While the circuits generated by the method are correct representations of the quantum operations, they are almost completely composed by quantum operands and wires, and do not include any classical operation beyond qubit preparations and measurements on some inputs and outputs.

Generally, in a physical quantum computer the implementation of classical operations is more efficient than their quantum counterparts, in addition to being cheaper and less error prone. Quantum simulators are commonly able to exploit the knowledge of which wires carry classical data to speed up their operation. As such, in this chapter we explore a method to detect parts of a hybrid quantum circuit that can be turned into classical circuits. These techniques can be directly applied after the previously presented extraction procedure.

Notice, however, that while we aim to recognize all classically realizable operations in the circuit, the characteristics of each quantum computer may dictate the final choice between quantum and classical operators by considering the costs of exchanging data between both realms.

5.1 The classicalisation problem

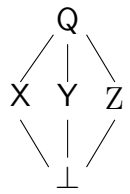
In this section we define the classicalisation problem as a labelling problem on the wires of a ZX$_{\perp}$ diagram that indicates the flow of classical information. Given a ZX$_{\perp}$ diagram corresponding to a quantum circuit, a valid labelling can be used to guide the extraction back into hybrid gates (Cf. Section 5.2). This formulation can be directly combined with the extraction method from ZX$_{\perp}$ diagrams admitting an extended gFlow into hybrid quantum-classical circuits presented in Chapter 4.

Given a ZX$_{\perp}$ diagram, we decorate its wires using labels from the set $\mathcal{L} = \{\mathsf{Q}, \mathsf{X}, \mathsf{Y}, \mathsf{Z}, \perp\}$. In a circuit-like diagram, the label Z indicates that this particular wire in the corresponding circuit carries the same information as a classical wire with data in the computational-basis and could be potentially replaced by one, possibly adding a standard basis measurement and a qubit initialisation in the standard basis depending on whether the connected wires are also classical or not. Similarly, the X and Y labels represent classical information carriers in the diagonal and Y basis respectively. A Q label means that the wire is a quantum wire carrying any (potentially entangled) qubit value, and finally the $\perp$ label marks a wire carrying no information, which could be replaced by discarding the input and generating a maximally mixed state as output. That is, a $\perp$ can always be replaced by two $\perp$ generators.

The wire labels can be interpreted as density matrix subspaces of $\mathbb{C}^{2\times 2}$, representing all possible mixed states allowed by a particular kind of wire. This interpretation is defined as follows.

$$\mathsf{Q} = \mathbb{C}^{2\times 2} \qquad \mathsf{Z} = \{\alpha\,|0\rangle\langle 0| + \beta\,|1\rangle\langle 1| \mid \alpha, \beta \in \mathbb{R}_{\geq 0}, \alpha + \beta = 1\}$$
$$\perp = \{\tfrac{1}{2}\,|0\rangle\langle 0| + \tfrac{1}{2}\,|1\rangle\langle 1|\} \qquad \mathsf{X} = \{\alpha\,|+\rangle\langle +| + \beta\,|-\rangle\langle -| \mid \alpha, \beta \in \mathbb{R}_{\geq 0}, \alpha + \beta = 1\}$$
$$\mathsf{Y} = \{\alpha\,|\circlearrowleft\rangle\langle\circlearrowleft| + \beta\,|\circlearrowright\rangle\langle\circlearrowright| \mid \alpha, \beta \in \mathbb{R}_{\geq 0}, \alpha + \beta = 1\}$$

The set of labels $\mathcal{L}$ therefore forms a partial order given by $\subseteq$ which can be depicted as follows.

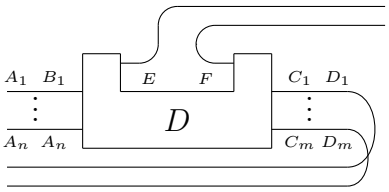


Notice that the greatest common ancestor of two labels $A \sqcup B$ corresponds to the intersection of the sets.

A labelling L of a diagram D is a map from its edges into a pair of labels. The two labels, drawn at each end of the wire, indicate the origin of the constraint. Intuitively, $\overset{\mathsf{Z}}{\underline{\quad}}\overset{A}{\underline{\quad}}$ means that the wire is produced in such a way that guarantees that the qubit carries classical information encoded in the computational basis, whereas $\overset{A}{\underline{\quad}}\overset{\mathsf{Z}}{\underline{\quad}}$ means that the wire can be replaced by a classical wire because some future process will force

this qubit to be measured in that basis —for instance, it is going to be measured in the standard basis and thus one can pre-emptively measure this qubit in the standard basis and use a classical wire instead. We define a partial order between labellings of a diagram as the natural lift from the partial order of the labels.

A labelling is valid if we can cut any wire in the diagram and, after forcing a valid state in the inputs and outputs, we get a valid state in the cut terminals. That is, we rearrange the diagram to transform all outputs into inputs and connect the cut terminals as outputs as shown below. Then, any arbitrary input $\rho \in (\bigotimes_i^n A_i) \otimes (\bigotimes_j^m D_j)$ applied to the diagram produces an output in $E \otimes F$.



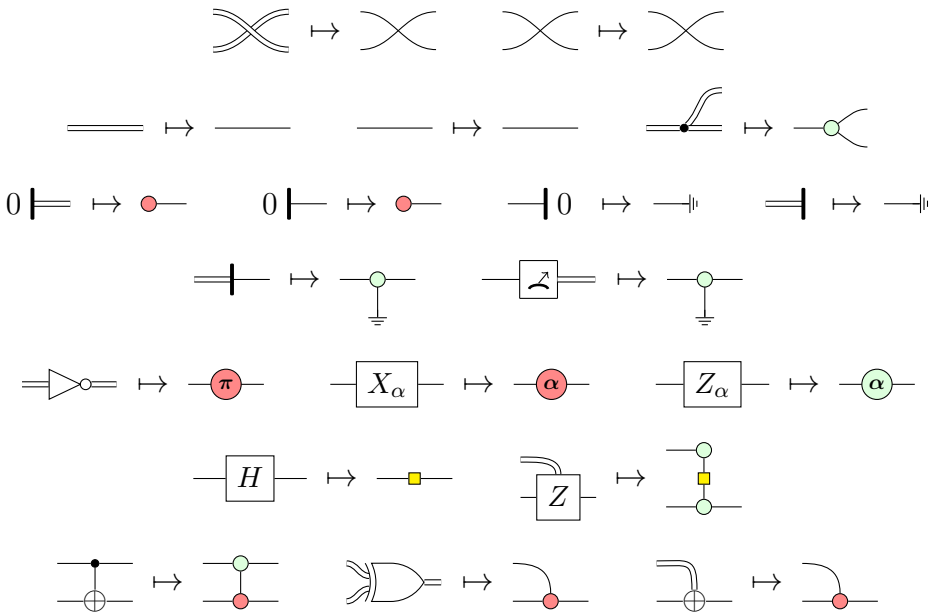
Notice that if A is a valid label for a wire then any $B \geq A$ is also valid, and in particular Q is always a valid label. We can therefore omit unnecessary labels in the diagrams, marking them implicitly as Q.

Given a $\mathrm{ZX}_{\frac{1}{2}}$ diagram D with marked classical inputs and outputs, we define the classicalisation problem as finding a minimal valid labelling where the inputs and outputs are labelled as Q or Z accordingly.

5.2 Labelled diagrams as hybrid circuits

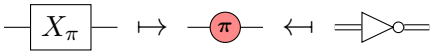
Consider now the diagrams resulting from the translation of a hybrid quantum-classical circuit with parity classical logic. As the classicalisation problem can be defined on arbitrary diagrams we are not required to produce graph-like diagrams as is the case with the compilation presented in Chapter 4. We present instead a simpler

direct translation which produces diagrams with both colours of spider.



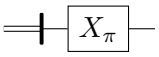
Since the labelling process does not alter the diagram corresponding to a circuit, we will now consider the significance of a labelling on a circuit.

Notice that due to the collapse of both quantum and classical wires into a single type of diagram edge, some quantum and classical gates may have the same diagrammatic representation, as show below for the case of a quantum X rotation and a classical NOT gate.

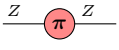


In the extraction procedure presented in Chapter 4 we took the most general strategy consisting of always extracting such patterns as quantum gates, and only introducing classical wires where strictly necessary. However, given a valid labelling on the diagram we may be able to correctly introduce classical gates in the circuit.

Consider a circuit applying an X rotation gate over a classical input,



and its corresponding $ZX_{\perp}$ diagram labelled,



The local label information around the spider further signals that both wires carry data in the computational basis. Therefore, we can replace the X gate in the circuit

with a classical NOT gate while preserving the semantics of the operation. We can verify this using the semantics,

$$\forall \rho \in \mathrm{Z} \qquad [\![\; \text{—} X_\pi \text{—} \;]\!] (\rho) = [\![\; \text{—} \triangleright\!\circ \text{—} \;]\!] (\rho)$$

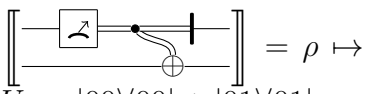
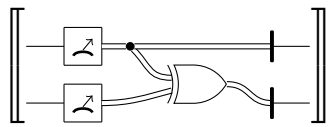
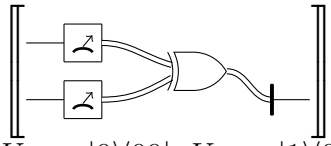
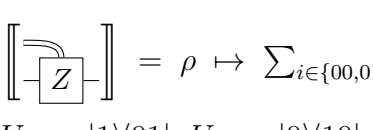
where $U(\rho) = U^\dagger \rho U$.

Following this idea, given a valid labelling of the diagram corresponding to a circuit we present below a set of replacement rules for each quantum gate. This replacement operation introduces qubit preparations and measurements that can be cancelled where applicable.

Gate	Labelled diagram	Replacement gates
0 —	• —	0 — | —
— | 0	— ⫴	— ↗ —
— X_π —	A π B where $A \sqcup B \leq \mathrm{Z}$	— ↗ —▷○—
— Z_α —	A α B where $A \sqcup B \leq \mathrm{Z}$	— ↗ —
• ⊕	A B • where $A \sqcup B \leq \mathrm{Z}$	↗ • ⊕ ↗
	A B C D where $A \sqcup B \leq \mathrm{Z}$ and $C \sqcup D \leq \mathrm{Z}$	↗ ↗
⊕	Z Z	↗ ↗
Z	Z Z Z	↗

Lemma 5.1 The replacement operations preserve the semantics of the circuit.

Proof By case analysis on the labelled diagrams.

- $[\![0 \text{ — }]\!] = |0\rangle\langle 0| = [\![\text{ —●— }]\!]$
- $[\![\text{ —↗═| }]\!] = \rho \mapsto \langle 0|\,\rho\,|0\rangle + \langle 1|\,\rho\,|1\rangle = [\![\text{ —⊣⊩ }]\!]$
- $[\![\text{ —↗═▷○═| }]\!] = \rho \mapsto |1\rangle\langle 0|\,\rho\,|0\rangle\langle 1| + |0\rangle\langle 1|\,\rho\,|1\rangle\langle 0| = [\![\text{ —}^{A}\pi^{B}\text{— }]\!]$ where $A \sqcup B \leq \mathrm{Z}$.
- $[\![\text{ —↗═| }]\!] = \rho \mapsto |0\rangle\langle 0|\,\rho\,|0\rangle\langle 0| + |1\rangle\langle 1|\,\rho\,|1\rangle\langle 1| = [\![\text{ —}^{A}\alpha^{B}\text{— }]\!]$ where $A \sqcup B \leq \mathrm{Z}$.
- $[\![$  $]\!] = \rho \mapsto U_0^\dagger \rho U_0 + U_1^\dagger \rho U_1 = [\![$ $]\!]$ where $A \sqcup B \leq \mathrm{Z}$, $U_0 = |00\rangle\langle 00| + |01\rangle\langle 01|$, and $U_1 = |11\rangle\langle 10| + |10\rangle\langle 11|$.
- $[\![$  $]\!] = \rho \mapsto \sum_{i\in\{00,01,10,11\}} U_i^\dagger \rho U_i = [\![$ $]\!]$ where $A \sqcup B \leq \mathrm{Z}$, $C \sqcup D \leq \mathrm{Z}$, $U_{00} = |00\rangle\langle 00|$, $U_{01} = |01\rangle\langle 01|$, $U_{10} = |11\rangle\langle 10|$, and $U_{11} = |10\rangle\langle 11|$.
- $[\![$  $]\!] = \rho \mapsto \sum_{i\in\{00,01,10,11\}} U_i^\dagger \rho U_i = [\![$ $]\!]$ where $U_{00} = |0\rangle\langle 00|$, $U_{01} = |1\rangle\langle 01|$, $U_{10} = |1\rangle\langle 10|$, and $U_{11} = |0\rangle\langle 11|$.
- $[\![$  $]\!] = \rho \mapsto \sum_{i\in\{00,01,10,11\}} U_i^\dagger \rho U_i = [\![$ $]\!]$ where $U_{00} = |0\rangle\langle 00|$, $U_{01} = |1\rangle\langle 01|$, $U_{10} = |0\rangle\langle 10|$, and $U_{11} = |1\rangle\langle 11|$. □

5.3 Push-relabel algorithm

We present a local search labelling heuristic for $\mathrm{ZX}_{\perp}$ diagrams with explicit Hadamard gates —replacing the Hadamard wires— and only green spiders, that produces locally minimal labellings by propagating the labels over individual spiders.

We introduce three operations over the labels. First, a binary function representing the result of combining two wires via a phaseless green spider, $\star : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$.

$$\begin{array}{lllll} \mathsf{Z} \star A = Z & \mathsf{X} \star A = A & \mathsf{Y} \star \mathsf{Y} = \mathsf{X} & \mathsf{Q} \star \mathsf{Y} = \mathsf{Q} & \perp \star \mathsf{Y} = \perp \\ A \star \mathsf{Z} = Z & A \star \mathsf{X} = A & \mathsf{Y} \star \mathsf{Q} = \mathsf{Q} & \mathsf{Q} \star \mathsf{Q} = \mathsf{Q} & \perp \star \mathsf{Q} = \mathsf{Z} \\ & & \mathsf{Y} \star \perp = \perp & \mathsf{Q} \star \perp = \mathsf{Z} & \perp \star \perp = \perp \end{array}$$

Notice that $(\mathcal{L}, \star)$ is a commutative monoid with X as neutral element. We also define a "Z rotation" operation for $\alpha \in [0, 2\pi)$, $\mathrm{rot}_\alpha : \mathcal{L} \to \mathcal{L}$.

$$\mathrm{rot}_\alpha(\mathsf{Z}) = \mathsf{Z} \qquad \mathrm{rot}_\alpha(\mathsf{Q}) = \mathsf{Q} \qquad \mathrm{rot}_\alpha(\bot) = \bot$$

$$\mathrm{rot}_\alpha(\mathsf{X}) = \begin{cases} \mathsf{X} & \text{if } \alpha \in \{0, \pi\} \\ \mathsf{Y} & \text{if } \alpha \in \{\frac{1}{2}\pi, \frac{3}{4}\pi\} \\ \mathsf{Q} & \text{otherwise} \end{cases} \qquad \mathrm{rot}_\alpha(\mathsf{Y}) = \begin{cases} \mathsf{Y} & \text{if } \alpha \in \{0, \pi\} \\ \mathsf{X} & \text{if } \alpha \in \{\frac{1}{2}\pi, \frac{3}{4}\pi\} \\ \mathsf{Q} & \text{otherwise} \end{cases}$$

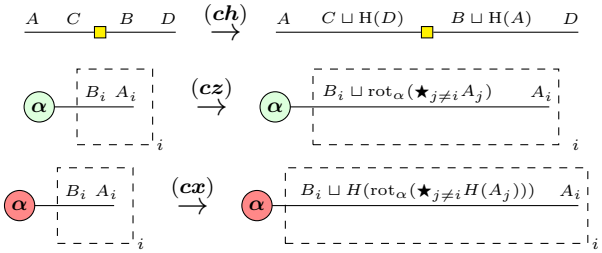
This corresponds to the identity if $\alpha \in \{0, \pi\}$, and in general $\mathrm{rot}_\alpha(A) \star B \geq \mathrm{rot}_\alpha(A \star B)$.

Finally, we define a function H representing the application of the Hadamard operation over a label, $\mathrm{H} : \mathcal{L} \to \mathcal{L}$.

$$\mathrm{H}(\mathsf{Q}) = \mathsf{Q} \qquad \mathrm{H}(\mathsf{X}) = \mathsf{Z} \qquad \mathrm{H}(\mathsf{Z}) = \mathsf{X} \qquad \mathrm{H}(\mathsf{Y}) = \mathsf{Y} \qquad \mathrm{H}(\bot) = \bot$$

Our classical detection procedure starts by labelling any classical input or output with a Z label, and any $\fatslash$ with a $\bot$ label, and the rest of the diagram wires with Q.

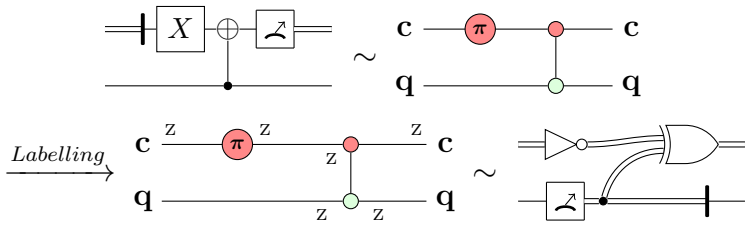
It then proceeds by propagating the labels using the following rules:



For any labels $A, B, C, D, E, F \in \mathcal{L}$.

We apply these rules until there are no more labels to change. Since each time we replace labels with lesser ones in the order, the procedure terminates. Finally, we can interpret wires with a classical label in any direction as classically realisable.

Example 5.1 This example shows the classicalisation of a hybrid quantum-classical circuit using the push-relabel algorithm. Q labels in the diagrams are omitted.

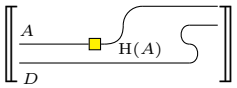


We now proceed to show that our push-relabel algorithm always produces valid labellings. To that end we first prove that both rules (***ch***) and (***cz***) produce valid labellings.

Lemma 5.2 The labelling rule (***ch***) preserves the validity of the labelling.

Proof We prove the validity of the replacement of label B. The replacement of label C is the symmetric case.

Since the starting diagram has a valid labelling, $B \sqcup \mathrm{H}(A)$ is a valid label if $\mathrm{H}(A)$ is a valid label. Therefore, the label is valid if $\forall \rho \in A \otimes B$,



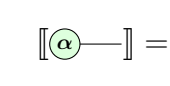
$\circ\, \rho \in \mathrm{H}(A) \otimes D$

Notice that this is equivalent to requiring $\forall a \in A,\ (H\, a\, H^\dagger) \in \mathrm{H}(A)$, which follows from the definition of $\mathrm{H}(\cdot)$. □

Lemma 5.3 The labelling rule (***cz***) preserves the validity of the labelling.

Proof By induction on the number of wires.

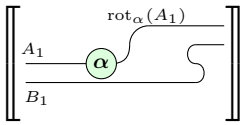
- If $n = 0$, there are no labelled wires.
- If $n = 1$, since the starting diagram has a valid labelling, $B_1 \sqcup \mathrm{rot}_\alpha(\mathsf{X})$ is a valid labelling if $\mathrm{rot}_\alpha(\mathsf{X})$ is valid. Notice that



$$= \begin{pmatrix} 1 & e^{-i\alpha} \\ e^{i\alpha} & 1 \end{pmatrix} \in \mathrm{rot}_\alpha(\mathsf{X})$$

- If $n = 2$, we prove the validity of the replacement of label B_2. The replacement of label B_1 is the symmetric case.

  Since the starting diagram has a valid labelling, $B_2 \sqcup \mathrm{rot}_\alpha(A_1)$ is a valid label if $\mathrm{rot}_\alpha(A_1)$ is a valid label. Therefore, the label is valid if $\forall \rho \in A_1 \otimes B_1$,

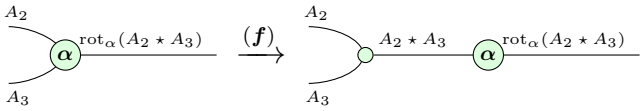


  $\circ\, \rho \in \mathrm{rot}_\alpha(A_1) \otimes B_1$

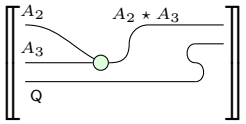
  Notice that this is equivalent to requiring $\forall a \in A_1, (U\, a\, U^\dagger) \in \mathrm{rot}_\alpha(A_1)$ for $U = |0\rangle\langle 0| + e^{i\alpha}\, |1\rangle\langle 1|$, which follows from the definition of $\mathrm{rot}_\alpha(\cdot)$.

- If $n = 3$, we prove the validity of the replacement of label B_1. The replacement of labels B_2 and B_3 are the symmetric case.

Since the starting diagram has a valid labelling, $B_1 \sqcup \mathrm{rot}_\alpha(A_2 \star A_3)$ is a valid label if $\mathrm{rot}_\alpha(A_2 \star A_3)$ is a valid label. We can split the diagram as follows, adding an intermediary label.



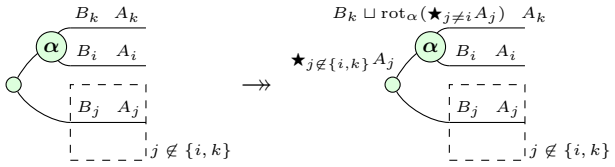
Notice that by the inductive hypothesis, the labelling step for the degree-2 spider is correct. Therefore, it suffices to prove that the intermediary label $A_2 \star A_3$ is valid, that is $\forall \rho \in A_2 \star A_3 \otimes \mathsf{Q}$,



$\circ\, \rho \in (A_2 \star A_3) \otimes \mathsf{Q}$

Notice that this is equivalent to requiring $\forall a_2 \in A_2, a_3 \in A_3$, $(U\,(a_2 \otimes a_3)\,U^\dagger) \in A_2 \star A_3$ for $U = |0\rangle\langle 00| + |1\rangle\langle 11|$, which follows from the definition of $\star$.

- If $n > 3$, for each i and for some $k \neq i$ we can rewrite the diagram as follows, and apply $(\boldsymbol{cz})$ twice to produce the target $B_i \sqcup \mathrm{rot}_\alpha(\bigstar_{j \neq i} A_j)$ label.



By inductive hypothesis, both rule applications produce valid labellings. □

Lemma 5.4 The labelling rule $(\boldsymbol{cx})$ preserves the validity of the labelling.

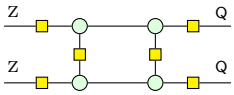
Proof By applying the colour changing rule $(\boldsymbol{h})$ and Lemmas 5.2 and Lemmas 5.3.□

Lemma 5.5 The local-search labelling algorithm produces a valid labelling according to the standard interpretation of the $\mathrm{ZX}_{\frac{1}{2}}$ calculus.

Proof Notice that labelling every wire as Q is always valid and hence the algorithm starts with a valid labelling. By Lemmas 5.2, 5.3, and 5.4, each step applying rules $(\boldsymbol{cz})$, $(\boldsymbol{cx})$ and $(\boldsymbol{ch})$ preserve the validity and therefore the final labelling is valid. □

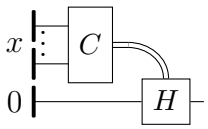
5.4 Global minima computation

The push-relabel algorithm presented in Section 5.3 efficiently finds a locally minimum labelling of the diagram. However, the procedure does not always reach a globally minimum labelling. Consider for example the following input diagram,



Here, although the diagram is equivalent to the identity and the outputs could theoretically be labelled as Z, the entangled wires in the middle do not allow for local propagation of the classical labels past the double controlled Z gates.

In general, finding the global minimum in the classicalisation problem has the same complexity as fully simulating a quantum circuit. Indeed, consider a circuit C solving an Exact Quantum Polynomial-Time (EQP) decision problem precomposed with a constant input x and postcomposed with a controlled Hadamard operation,



Simulating the output of $C(x)$ is equivalent to deciding if the output can be labelled as either Z or X.

Conclusion

We are still a long way from having fully functional quantum computers that can be used for real-world applications. However, the progress made in the last few years is very promising. Part of this progress requires the development of tools that help with compiling human-readable descriptions of quantum algorithms into efficient programs that can be executed on a quantum computer. In this thesis we have presented a series of contributions to the different stages of the quantum compilation process, from the representation of high-level primitives in an optimizable intermediate representation, to gate-level optimizations of quantum circuits.

Throughout this work we have focused on methods that employ multiple extensions of the ZX calculus to manipulate the quantum programs during each step of the compilation. We found that it is a very powerful tool for this kind of reasoning, since it splits the circuit description into simpler elements that lend themselves well to optimization techniques. Furthermore, the formal nature of the calculus have allowed us to easily prove the correctness of each of the transformations we have developed.

For our first contribution we leveraged a scalable extension of the ZX calculus to define a new intermediate representation of quantum programs that is able to compactly represent the repeated structures generated during iterations and parallel operations. This representation was used to compile a reduced fragment of the Proto-Quipper-D language, where we showed examples of the encoding of non-trivial algorithms. As future work, we plan to give some well-suited formal semantics to the intermediate representation, and to use it as the compilation target of other high-level quantum programming languages.

For our second contribution we developed a new method for the optimization of hybrid quantum-classical circuits by first transforming them into a $\mathrm{ZX}_{\Perp}$ diagrams and applying a series of formally proven transformations using an eager algorithm. We implemented these methods on `pyzx`, a Python library for manipulating ZX diagrams and optimizing circuits with it. Part of this work has been upstreamed into the main project.

Lastly, we defined a problem for detecting classically implementable logic in quantum circuits. We described a heuristic algorithm for solving the problem, and

discussed why an exact solver would be intractable. This development was aimed for the artifacts produced by our optimizing compilation program, but it can be used for processing arbitrary quantum circuits.

Bibliography

[1] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial String Diagrams for Reverse-Mode Automatic Differentiation. *arXiv:2107.13433 [cs]*, July 2021. arXiv: 2107.13433.

[2] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time T-depth Optimization of Clifford+T circuits via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, October 2014. arXiv: 1303.2042.

[3] Pablo Arrighi, Alejandro Díaz-Caro, and Benoît Valiron. The vectorial -calculus. *Information and Computation*, 254:105–139, 2017.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[5] Miriam Backens and Aleks Kissinger. ZH: A complete graphical calculus for quantum computations involving classical non-linearity. *Electronic Proceedings in Theoretical Computer Science*, 287:23–42, jan 2019.

[6] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. There and back again: A circuit extraction tale, 2020.

[7] HP Barendregt. The lambda calculus. number 103 in studies in logic and the foundations of mathematics, 1991.

[8] Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories, 2020.

[9] Agustín Borgna and Rafael Romero. Encoding High-level Quantum Programs as SZX-diagrams. In *QPL2022*, Oxford, United Kingdom, June 2022.

[10] Agustín Borgna. Ground optimization extensions to `pyzx`. `https://github.com/aborgna/pyzx/tree/zxgnd`, 2021.

[11] Agustín Borgna, Simon Perdrix, and Benoît Valiron. Hybrid Quantum-Classical Circuit Simplification with the ZX-Calculus. In Hakjoo Oh, editor, *Programming Languages and Systems*, pages 121–139, Cham, 2021. Springer International Publishing.

[12] Titouan Carette. A note on diagonal gates in SZX-calculus. *arXiv:2012.09540 [quant-ph]*, December 2020. arXiv: 2012.09540.

[13] Titouan Carette, Yohann D'Anello, and Simon Perdrix. Quantum Algorithms and Oracles with the Scalable ZX-calculus. *Electronic Proceedings in Theoretical Computer Science*, 343:193–209, September 2021.

[14] Titouan Carette, Marc de Visme, and Simon Perdrix. Graphical Language with Delayed Trace: Picturing Quantum Computing with Finite Memory. *arXiv:2102.03133 [quant-ph]*, April 2021. arXiv: 2102.03133.

[15] Titouan Carette, Dominic Horsman, and Simon Perdrix. SZX-calculus: Scalable Graphical Quantum Reasoning. *arXiv:1905.00041 [quant-ph]*, page 15 pages, 2019. arXiv: 1905.00041.

[16] Titouan Carette, Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of graphical languages for mixed states quantum mechanics, 2019.

[17] Titouan Carette and Simon Perdrix. Colored props for large scale graphical reasoning. *arXiv:2007.03564 [quant-ph]*, July 2020. arXiv: 2007.03564.

[18] David Cattanéo and Simon Perdrix. Minimum degree up to local complementation: Bounds, parameterized complexity, and exact algorithms. *Lecture Notes in Computer Science*, pages 259–270, 2015.

[19] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 264–275, 1996.

[20] Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren, and Dominic Horsman. Graphical Structures for Design and Verification of Quantum Error Correction. *arXiv preprint arXiv:1611.08012*, 2016.

[21] Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren, and Dominic Horsman. Graphical Structures for Design and Verification of Quantum Error Correction. *arXiv:1611.08012 [quant-ph]*, January 2018. arXiv: 1611.08012.

[22] Bob Coecke and Ross Duncan. Interacting Quantum Observables. In *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 298–310, Berlin, Heidelberg, 2008. Springer.

[23] Bob Coecke and Simon Perdrix. Environment and classical channels in categorical quantum mechanics. *Logical Methods in Computer Science*, Volume 8, Issue 4, November 2012.

[24] Bob Coeke and Aleks Kissinger. *Picturing Quantum processes - A diagrammatic approach*. Cambridge University Press, 2017.

[25] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017.

[26] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017.

[27] Niel de Beaudrap. Finding flows in the one-way measurement model. *Physical Review A*, 77(2), Feb 2008.

[28] Niel de Beaudrap and Dominic Horsman. The ZX calculus is a language for surface code lattice surgery. *Quantum*, 4:218, January 2020. tex.ids: debeaudrapZXCalculusLanguage2020a arXiv: 1704.08670.

[29] Niel de Beaudrap, Aleks Kissinger, and John van de Wetering. Circuit Extraction for ZX-diagrams can be #P-hard. *arXiv:2202.09194 [quant-ph]*, February 2022. arXiv: 2202.09194.

[30] Niel de Beaudrap and Martin Pei. An extremal result for geometries in the one-way measurement model, 2007.

[31] Cirq Developers. Cirq, August 2021. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

[32] Alejandro Dí az-Caro. A lambda calculus for density matrices with classical and probabilistic controls. In *Programming Languages and Systems*, pages 448–467. Springer International Publishing, 2017.

[33] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic simplification of quantum circuits with the zx-calculus, 2019.

[34] Ross Duncan and Maxime Lucas. Verifying the Steane code with Quantomatic. *Electronic Proceedings in Theoretical Computer Science*, 171:33–49, December 2014. arXiv: 1306.4532.

[35] Ross Duncan and Simon Perdrix. Rewriting measurement-based quantum computations with generalised flow. In *International Colloquium on Automata, Languages, and Programming*, pages 285–296. Springer, 2010.

[36] R Kent Dybvig. *The SCHEME programming language*. Mit Press, 2009.

[37] MD SAJID ANIS et al. Qiskit: An open-source framework for quantum computing, 2021.

[38] Peng Fu. `dpq`, proto-quipper-d implementation. `https://gitlab.com/frank-peng-fu/dpq-remake`, 2022.

[39] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. *arXiv:2005.08396 [quant-ph]*, December 2020. arXiv: 2005.08396.

[40] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. Proto-quipper with dynamic lifting, 2022.

[41] Peng Fu, Kohei Kishida, and Peter Selinger. Linear Dependent Type Theory for Quantum Programming Languages. *arXiv:2004.13472 [quant-ph]*, December 2021. arXiv: 2004.13472.

[42] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper. *ACM SIGPLAN Notices*, 48(6):333, Jun 2013.

[43] Luke Heyfron and Earl T. Campbell. An Efficient Quantum Compiler that reduces T count. *arXiv:1712.01557 [quant-ph]*, June 2018. arXiv: 1712.01557.

[44] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of the ZX-Calculus. *arXiv:1903.06035 [quant-ph]*, October 2019. arXiv: 1903.06035.

[45] Richard Jozsa. An introduction to measurement based quantum computation. *Quantum Information Processing*, 199, 09 2005.

[46] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. cqasm v1.0: Towards a common quantum assembly language, 2018.

[47] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science*, 318:229–241, May 2020. arXiv: 1904.04735.

[48] Aleks Kissinger and John van de Wetering. Reducing T-count with the ZX-calculus. *arXiv:1903.10477 [quant-ph]*, January 2020. arXiv: 1903.10477.

[49] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system: Documentation and user's manual. *INRIA*, 3:42.

[50] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[51] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[52] Paul-André Melliès. Functorial Boxes in String Diagrams. In *Computer Science Logic*, volume 4207, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

[53] Mehdi Mhalla, Mio Murao, Simon Perdrix, Masato Someya, and Peter S. Turner. Which Graph States are Useful for Quantum Information Processing? In *Theory of Quantum Computation, Communication, and Cryptography*, volume 6745, pages 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[54] Hector Miller-Bakewell. Finite Verification of Infinite Families of Diagram Equations. *Electronic Proceedings in Theoretical Computer Science*, 318:27–52, May 2020. arXiv: 1904.00706 version: 2.

[55] Huy-Tung Nguyen and Sang il Oum. The average cut-rank of graphs, 2019.

[56] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.

[57] Francisco Noriega and Alejandro Díaz-Caro. The vectorial lambda calculus revisited, 2020.

[58] Michele Pagani. Some advances in linear logic. *Mémoire pour l'obtention de l'habilitation à diriger les recherches*, December 2013.

[59] Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. *arXiv:1311.2290 [quant-ph]*, November 2013. tex.ids: paganiApplyingQuantitativeSemantics2014 arXiv: 1311.2290 tex.publisher: ACM.

[60] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[61] PETER SELINGER. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[62] Peter Selinger. Dagger compact closed categories and completely positive maps: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 170:139 – 163, 2007. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005).

[63] Peter Selinger. Quantum circuits of t-depth one. *Physical Review A*, 87(4), Apr 2013.

[64] Peter Selinger, Benoıt Valiron, et al. Quantum lambda calculus. *Semantic techniques in quantum computation*, pages 135–172, 2009.

[65] Peter Selinger and Benoît Valiron. *Quantum Lambda Calculus*, page 135–172. Cambridge University Press, 2009.

[66] Damian S. Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, Jan 2018.

[67] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. Q#. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, 2018.

[68] Benoît Valiron. Semantics of quantum programming languages: Classical control, quantum control. *Journal of Logical and Algebraic Methods in Programming*, 128:100790, 2022.

[69] John van de Wetering. ZX-calculus for the working quantum computer scientist. *arXiv preprint arXiv:2012.13966*, 2020.

[70] Renaud Vilmart. A zx-calculus with triangles for toffoli-hadamard, clifford+t, and beyond. *Electronic Proceedings in Theoretical Computer Science*, 287:313–344, Jan 2019.

[71] Huiyang Zhou and Gregory T. Byrd. Quantum circuits for dynamic runtime assertions in quantum computation. *IEEE Computer Architecture Letters*, 18(2):111–114, Jul 2019.

Abstract

The advent of quantum computers capable of solving problems that are untractable on classical computers has motivated the development of new programming languages and tools for quantum computing. However, the current state of the art in quantum programming is still in its infancy. In this thesis, we present a series of novel approaches to different aspects of the quantum compilation process based on the ZX calculus.

First, we introduce a new intermediate representation for quantum programs capable of encoding bounded recursion and repeated circuit structures in a compact way, based on families of the Scalable extension to the ZX calculus. We then present a compilation algorithm for hybrid circuits containing both quantum and classical gates, based on the pure circuit optimization by Duncan et al. Finally, we define the problem of detecting sections of a quantum circuit that can translated to classical logic, and introduce an heuristic algorithm to solve it.

Keywords: Quantum computing, Compilation, ZX-Calculus, Optimization.

Résumé

L'avènement des ordinateurs quantiques capables de résoudre des problèmes irréalisables sur des ordinateurs classiques a motivé le développement de nouveaux langages et outils de programmation pour l'informatique quantique. Cependant, l'état de l'art actuel en matière de programmation quantique n'en est qu'à ses débuts. Dans cette thèse, nous présentons une série d'approches novatrices de différents aspects du processus de compilation quantique basé sur le calcul ZX.

Tout d'abord, nous introduisons une nouvelle représentation intermédiaire pour les programmes quantiques capable d'encoder la récursion bornée et les structures de circuits répétés de manière compacte, basée sur les familles de l'extension Scalable du calcul ZX. Nous présentons ensuite un algorithme de compilation pour les circuits hybrides contenant à la fois des portes quantiques et classiques, basé sur l'optimisation de circuits purs de Duncan et al. Enfin, nous définissons le problème de la détection des sections de circuits quantiques qui peuvent être traduites en logique classique, et nous introduisons un algorithme heuristique pour le résoudre.

Mots-clés: Informatique quantique, Compilation, ZX-Calcul, Optimisation.

Résumé étendu

La théorie de la mécanique quantique a été développée il y a plus d'un siècle et a été raffinée depuis lors. Ce modèle a été utilisé pour expliquer avec succès un large éventail de phénomènes, du comportement des atomes et des molécules à la nature de la lumière. Plus récemment, les propriétés de la mécanique quantique ont été utilisées pour développer un nouveau modèle de calcul fondamentalement différent du modèle classique. L'exécution d'un ordinateur quantique repose sur une propriété quantique du système, qu'il s'agisse du spin d'un électron ou de la polarisation d'un photon. Il utilise ce degré de liberté pour coder l'information en tant que superposition d'états qui seraient impossibles à traiter sur un ordinateur classique.

Le développement de l'informatique quantique en est encore à ses débuts. Le premier jalon de la suprématie quantique a été atteint en 2019, où un ordinateur quantique a été capable de résoudre une tâche qui prendrait à un ordinateur classique une quantité de temps impraticable. Cependant, l'état actuel de l'informatique quantique est encore loin de pouvoir résoudre des problèmes du monde réel. L'ère à court terme des ordinateurs quantiques est communément appelée "Noisy Intermediate-Scale Quantum (NISQ)", où les ordinateurs quantiques sont limités à quelques centaines de qubits et sont fortement soumis au bruit qui peut corrompre l'état quantique du système. On s'attend à ce que les futurs ordinateurs quantiques avec un nombre plus élevé de qubits et des opérations moins bruyantes puissent mettre en œuvre des schémas tolérants aux pannes qui permettront l'exécution d'algorithmes plus complexes.

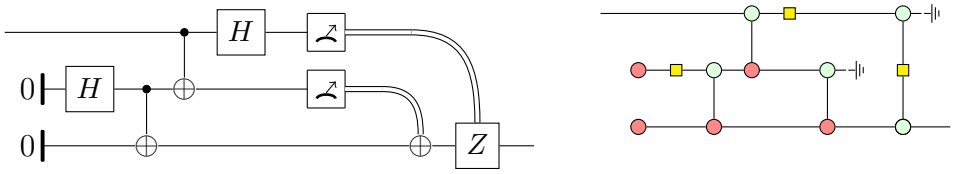
De nombreux groupes de recherche travaillent actuellement à la production d'un ordinateur quantique. Ils cherchent tous à résoudre le même problème mais utilisent pour cela différentes propriétés physiques pour encoder les qubits. Néanmoins, la plupart de ces ordinateurs expérimentaux partagent une interface commune basée sur le modèle de circuit. Dans ce modèle, un ordinateur quantique est composé d'un ensemble de qubits, chacun pouvant être initialisé dans un état particulier, puis manipulé par un ensemble de portes quantiques. Les qubits peuvent également être mesurés, ce qui fera s'effondrer l'état quantique et renvoyer un bit classique.

Cependant, une description d'un algorithme complexe en termes d'opérations de bas niveau peut être fastidieuse et sujette aux erreurs à produire, et elle serait liée à l'ensemble spécifique de portes disponibles sur l'ordinateur quantique. Il est souvent plus pratique d'utiliser un langage de programmation de haut niveau pour abstraire sur les ensembles spécifiques de portes, et exprimer des opérations génériques telles que le travail avec des listes de qubits ou l'exécution de branches conditionnelles.

La mise en œuvre de l'abstraction de programmation de haut niveau nécessite l'optimisation des programmes résultants en représentations de bas niveau, qui peuvent être exécutées efficacement sur un matériel limité. Cela est particulièrement crucial

dans les systèmes de calcul quantique, où la conception d'un langage formel graphique permettant une représentation fine des opérations quantiques est devenue un domaine de recherche actif. C'est là que le calcul ZX entre en jeu. Le calcul ZX est un langage graphique formel qui fournit une représentation fine des opérations quantiques, permettant une conception compacte de structures répétées et une extension des opérations représentables. Une introduction détaillée à la famille des calculs ZX peut être trouvée dans [69].

Example 1 En utilisant le calcul ZX$_{\frac{1}{2}}$, il est possible de coder des expériences hybrides quantique-classique sous forme de diagrammes. Par exemple, considérons la représentation de circuit suivante du protocole de téléportation quantique (gauche). Nous pouvons encoder les mêmes opérations sous forme de diagramme ZX$_{\frac{1}{2}}$(droite).



Le diagramme ZX$_{\frac{1}{2}}$ fournit une vue plus détaillée qui est adaptée à l'optimisation des opérations quantiques.

Ce travail s'articule autour de trois contributions pour une chaîne d'outils de compilation utilisant le calcul ZX :

- Nous présentons l'encodage d'un langage de programmation quantique de haut niveau avec des types dépendants en une nouvelle représentation intermédiaire basée sur l'extension SZX de la famille de calculs ZX.
- Nous présentons la définition d'une nouvelle procédure d'optimisation pour les programmes hybrides quantique-classique basée sur l'optimisation pure de [33].
- Nous présentons notre dernière contribution, la définition d'une heuristique pour détecter les segments classiques dans un circuit quantique hybride-classique.

Compilation de programmes quantiques de haut niveau en diagrammes SZX

Alors que le calcul ZX s'avère être une bonne représentation pour des applications quantiques spécifiques, il souffre de la même limitation que les circuits quantiques, à savoir qu'il ne peut pas exprimer des opérations génériques avec une itération ou un parallélisme paramétrique. Par exemple, une fonction appliquant une porte quantique

à chaque qubit dans un registre devrait définir le nombre exact de qubits avant de la compiler en un diagramme avec un nombre proportionnel de nœuds.

Une représentation alternative des programmes quantiques peut être obtenue en utilisant des familles de diagrammes SZX. Nous montrons comment cette extension peut être utilisée comme représentation intermédiaire efficace pour coder des opérations parallèles et itératives paramétriques dans des diagrammes de taille constante.

Le langage Proto-Quipper-D est un langage puissant pour décrire des opérations quantiques de haut niveau. Cette instance de la famille Proto-Quipper comprend la prise en charge de fonctions avec types dépendants. Cette capacité permet de coder une fonction comme l'application parallèle de portes quantiques avec un paramètre naturel explicite indiquant le nombre de qubits à utiliser. Cependant, le langage complet peut décrire des opérations qui ne peuvent pas être codées en tant que diagrammes SZX, telles que des opérations non-terminales et une récursion non bornée. Nous sommes donc obligés de définir un fragment moins expressif du langage qui peut être entièrement codé en tant que diagrammes SZX.

Nous introduisons λ_D, un sous-ensemble des programmes Proto-Quipper-D fortement normalisants. Autrement dit, nous considérons le sous-ensemble de programmes Proto-Quipper-D terminant que nous sommes en mesure d'encoder en tant que diagrammes SZX. Le calcul inclut des produits, des vecteurs de longueur fixe, des nombres naturels et un certain nombre de primitives quantiques. À partir de ce langage, nous définissons une traduction en familles de diagrammes SZX avec des variables de contexte en entrée et des valeurs de termes en sortie.

Example 2 En utilisant λ_D, nous sommes en mesure de coder une description générique de l'algorithme de la Transformée de Fourier Quantique (TFQ). Nous présentons ici la définition de la méthode principale. L'algorithme complet peut être trouvé dans l'article principal.

$\mathrm{qft} : (n : \mathtt{Nat}) \to \mathtt{Vec}\ n\ \mathtt{Q} \multimap \mathtt{Vec}\ n\ \mathtt{Q}$

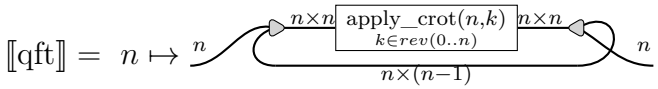
$$\mathrm{qft} := \lambda' n^{\mathtt{Nat}}.\lambda qs^{\mathtt{Vec}\ n\ \mathtt{Q}}.\mathtt{compose}$$
$$(\mathtt{for}\ k^{\mathtt{Nat}}\ \mathtt{in}\ \mathrm{reverse\_vec}\ @(0..n)\ \mathtt{do}\ \lambda qs'^{\mathtt{Vec}\ n\ \mathtt{Q}}.\mathrm{apply\_crot}\ @n\ @k\ qs')\ qs$$

Nous pouvons exprimer un programme équivalent à l'aide d'une syntaxe similaire à Haskell.

```
qft_aux : ! (n : Nat) -> ! (k : Nat) -> Vec Qubit n -> Vec Qubit n
qft_aux n head_size qs = applyCrot n head_size qs

qft : ! (n : Nat) -> Vec Qubit n -> Vec Qubit n
qft n qs = let f = qft_aux n in compose' (foreach f $ reverse_vec (0..n)) qs
```

À l'aide de notre traduction, nous sommes en mesure de coder l'opération en famille de diagrammes SZX comme suit:

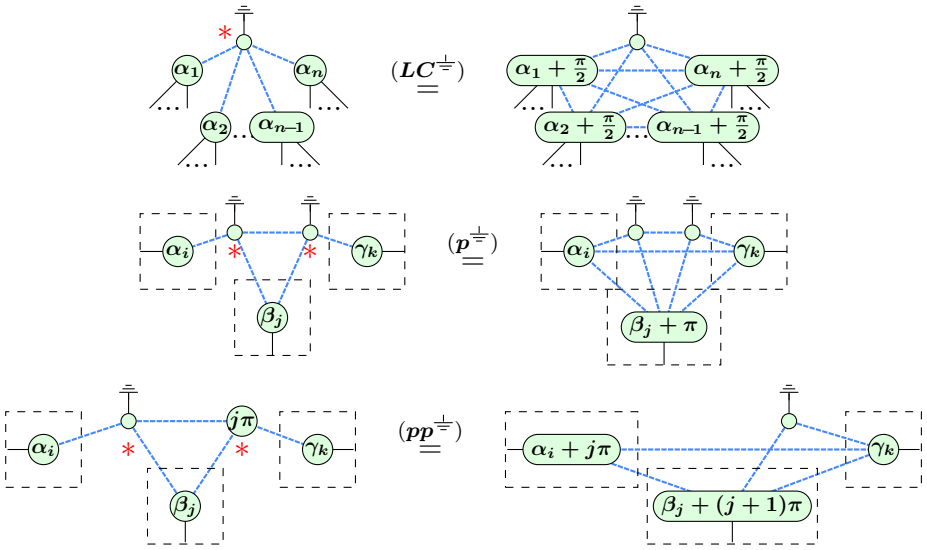


Contrairement à un codage de circuit quantique de QFT qui nécessite un nombre quadratique de portes, la taille du diagramme SZX produit ne dépend pas du nombre de qubits n.

Optimisation de circuits hybrides quantique-classique avec le calcul ZX$_{\overline{=}}^{\perp}$

Les stratégies d'optimisation les plus courantes pour les circuits quantiques se concentrent exclusivement sur les opérations pures, sans prendre en compte les structures hybrides quantiques-classiques. Bien que les circuits quantiques purs puissent être efficacement optimisés indépendamment des segments classiques, il est naturel de considérer le problème plus général de l'optimisation en prenant en compte l'interaction avec l'environnement en plus des fragments quantiques purs. Dans ce travail, nous présentons une procédure d'optimisation pour les circuits hybrides quantiques-classiques en utilisant le calcul ZX en étendant une méthode d'optimisation de circuits quantiques purs de [33].

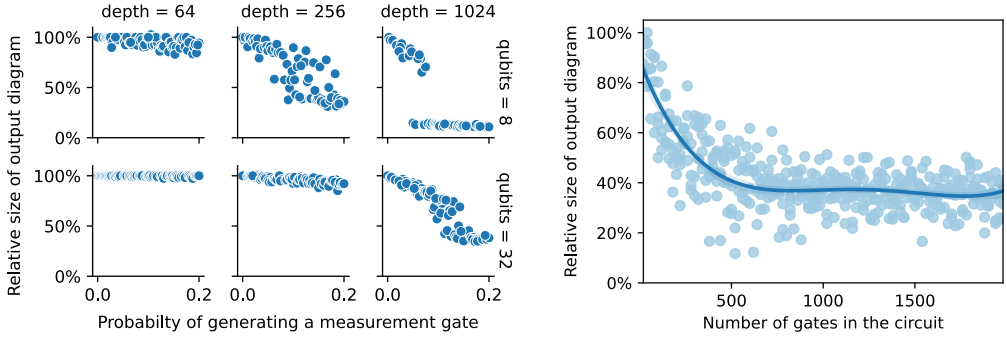
Nous introduisons les trois règles de réécriture suivantes pour les diagrammes ZX$_{\overline{=}}^{\perp}$ qui préservent la structure des diagrammes de telle manière que nous sommes capables d'extraire un circuit par la suite :



Nous utilisons ces nouvelles règles de réécriture en conjonction avec une stratégie de recherche de correspondances de motifs pour définir un algorithme complet qui,

à partir d'un circuit quantique, le traduit en un diagramme ZX$_{\downarrow}$, l'optimise à une forme minimale, et enfin extrait un circuit plus efficace.

Dans la figure 1, nous présentons une série de tests pour notre procédure d'optimisation, qui montrent une réduction cohérente de la taille du circuit lors de l'utilisation de notre algorithme.



(a) Réduction de la taille du diagramme pour les circuits Clifford+T avec mesures.

(b) Réduction de la taille du diagramme pour les circuits de logique de parité.

Figure 1: Résultats des tests de performances sur des diagrammes générés aléatoirement.

Détection de la classique utilisant les diagrammes ZX$_{\downarrow}$

En dernier résultat de ce travail, nous présentons un algorithme de détection des opérations classiques dans un circuit hybride quantique-classique. Notre procédure est basée sur l'ajout d'étiquettes aux nœuds sur le diagramme équivalent ZX$_{\downarrow}$ et l'utilisation d'une stratégie de poussée d'étiquettes pour trouver des étiquetages valides localement minimaux.

Example 3 Cet exemple montre la classique-isation d'un circuit hybride quantique-classique à l'aide de l'algorithme de pousser-étiqueter.