Encoding High-level Quantum Programs as SZX-diagrams

A. Borgna
CNRS LORIA, Inria-MOCQUA,
Université de Lorraine
CNRS, LMF,
Université Paris-Saclay
agustin.borgna@loria.fr

R. Romero
CONICET, Instituto de Ciencias de la Computación
Universidad de Buenos Aires, Buenos Aires, Argentina
PEDECIBA,
Universidad de la República-MEC. Montevideo, Uruguay
lromero@dc.uba.ar

The Scalable ZX-calculus is a compact graphical language used to reason about linear maps between quantum states. These diagrams have multiple applications, but they frequently have to be constructed in a case-by-case basis. In this work we present a method to encode quantum programs implemented in a fragment of the linear dependently typed Proto-Quipper-D language as families of SZX-diagrams. We define a subset of translatable Proto-Quipper-D programs and show that our procedure is able to encode non-trivial algorithms as diagrams that grow linearly on the size of the program.

1 Introduction

The ZX calculus [18] has been used as intermediary representation language for quantum programs in optimization methods [12, 5, 3] and in the design of error correcting schemes [4]. The highly flexible representation of linear maps as open graphs with a complete formal rewriting system and the multiple extensions adapted to represent different sets of quantum primitives have proven useful in reasoning about the properties of quantum circuits.

Quantum operations are usually represented as quantum circuits composed by primitive gates operating over a fixed number of qubits. The ZX calculus has a close correspondence to this model and is similarly limited to representing operations at a single-qubit level. In this work we will focus on the Scalable ZX extension [7], which generalizes the ZX diagrams to work with arbitrary qubit registers using a compact representation. Previous work [6] has shown that the SZX calculus is capable of encoding nontrivial algorithms via the presentation of multiple hand-written examples. For an efficient usage as an intermediate representation language, we require an automated compilation method from quantum programming languages to SZX diagrams. While ZX diagrams can be directly obtained from a program compiled to a quantum circuit, to the best of our knowledge there is no efficient method leveraging the parametricity of the SZX calculus.

There exist several quantum programming languages capable of encoding high-level parametric programs [1, 11, 17]. Quipper [15] is a language for quantum computing capable or generating families of quantum operations indexed by parameters. These parameters need to be instantiated at compile time to generate concrete quantum circuit representations. Quipper has multiple formal specifications, in this work we focus on the linear dependently typed Proto-Quipper-D formalization [14, 13] to express high-level programs with integer parameters.

The contributions of this article the following. We introduce a *list initialization* notation to represent multiple elements of a SZX diagram family composed in parallel. We formally define a fragment of Proto-Quipper-D programs that can be described as families of diagrams. Then we present a novel compilation method that encodes quantum programs as families of SZX diagrams and demonstrate the codification and translation of a nontrivial algorithm using our procedure.

Submitted to:
QPL 2022

In Section 2 we outline both languages and introduce the list initialization notation. In Section 3 we define the restricted Proto-Quipper-D fragment. In Section 4 we introduce the translation into SZX diagrams. Finally, in Section 5 we demonstrate an encoding of the Quantum Fourier Transform algorithm using our method. The proofs of the lemmas stated in this work can be found in Appendix E.

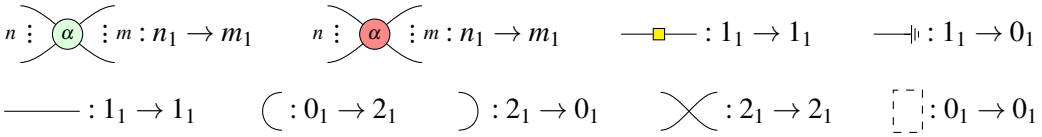
2 Background

We describe a quantum state as a system of n qubits corresponding to a vector in the $\mathbb{C}^{2^n}$ Hilbert space. We may partition the set of qubits into multi-qubit *registers* representing logically related subsets. Quantum computations under the QRAM model correspond to compositions of unitary operators between these quantum states, called quantum gates. Additionally, the qubits may be initialized on a set state and measured.

High-level programs can be encoded in Quipper [15], a Haskell-like programming language for describing quantum computations. In this work we use a formalization of the language called Proto-Quipper-D[13] with support for linear and dependent types. Concrete quantum operations correspond to linear functions between quantum states, generated as a composition of primitive operations that can be described directly as a quantum circuit. Generic circuits may have additional parameters that must fixed at compilation time to produce the corresponding quantum circuit.

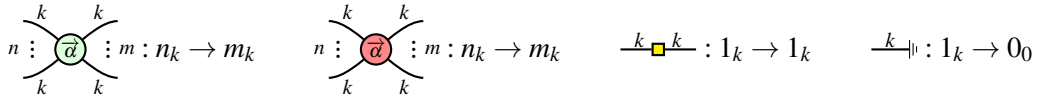
In Section 3 we describe a restricted fragment of the Proto-Quipper-D language containing the relevant operations for the work presented in this paper.

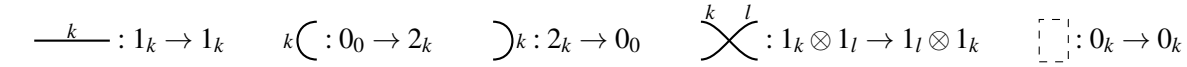
2.1 The Scalable ZX-calculus. The ZX calculus [18] is a formal graphical language that encodes linear maps between quantum states. Multiple extensions to the calculus have been proposed. We first present the base calculus with the grounded-ZX extension, denoted $\text{ZX}_{\perp}$ [10], to allow us to encode quantum state measurement operations. A $\text{ZX}_{\perp}$ diagram is generated by the following primitives, in addition to parallel and serial composition:



where n_k represents the n-tensor of k-qubit registers, the green and red nodes are called Z and X spiders, $\alpha \in [0, 2\pi)$ is the phase of the spiders, and the yellow square is called the Hadamard node. These primitives allow us to encode any quantum operation, but they can become impractical when working with multiple qubit registers.

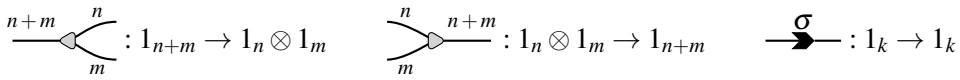
The SZX calculus [7, 6] is a *Scalable* extension to the ZX-calculus that generalizes the primitives to work with arbitrarily sized qubit registers. This facilitates the representation of diagrams with repeated structure in a compact manner. Carette et al. [6] show that the scalable and grounded extensions can be directly composed. Will refer to the resulting $\text{SZX}_{\perp}$-calculus 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 $\overline{\alpha} = \alpha_1 :: \cdots :: \alpha_k$. The natural extension of the ZX generators correspond to the following primitives:





Wires of multiplicity zero are equivalent to the empty mapping. 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. Using the split node and the wire primitives can derive the rotated version, which we call a *gather*.

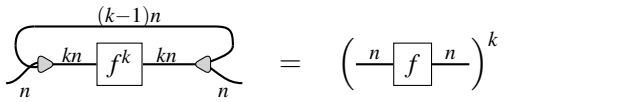


The rewriting rules of the calculus imply that a SZX diagrams can be considered as an open graph where only the topology of its nodes and edges matters. In the translation process we will make repeated use of the following reductions rules to simplify the diagrams:

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.

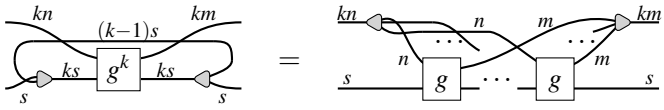
Refer to Appendix A for a complete definition of the rewriting rules and the interpretation of the SZX calculus. Cf. [6] for a description of the calculus including the generalized arrow generators.

Carette et al. [6] 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 this construction can be used 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

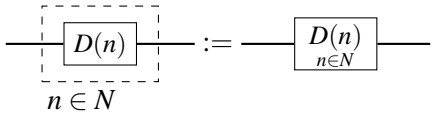


As an example, given a list $N = [n_1, n_2, n_3]$ and a starting accumulator value x_0, this construction would produce the mapping $([n_1, n_2, n_3], x_0) \mapsto ([m_1, m_2, m_3], x_3)$ where $(m_i, x_i) = g(n_i, x_{i-1})$ for $i \in [1, 3]$.

2.2 SZX diagram families and list instantiation. We introduce the definition of a family of SZX diagrams $D : \mathbb{N}^k \to \mathscr{D}$ 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.

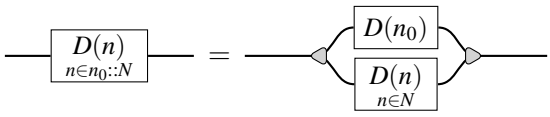
Since instantiations of a family share the same structure, we can compose them in parallel by merging the different values of wire tags and spider phases. We introduce a shorthand for instantiating a family of diagrams on multiple values and combining the resulting diagrams in parallel. This definition is strictly

more general than the *thickening endofunctor* presented by Carette et al. [6], which replicates a concrete diagram in parallel. A *list instantiation* of a family of diagrams $D : \mathbb{N}^{k+1} \to \mathscr{D}$ over a list N of integers is written as $(D(n), n \in N)$. This results in a family with one fewer parameter, $(D(n), n \in N) : \mathbb{N}^k \to \mathscr{D}$. We graphically depict a list instantiation as a dashed box in a diagram, as follows.

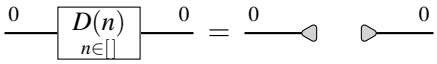


The definition of the list instantiation operator is given recursively on the construction of D in Figure 1. On the diagram wires we use $v(N)$ to denote the wire cardinality $\sum_{n\in N} v(n)$, $\overrightarrow{\alpha}(N)$ for the concatenation of phase vectors $\overrightarrow{\alpha}(n_1) :: \cdots :: \overrightarrow{\alpha}(n_m)$, 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**).

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



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



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

3 The λ_D calculus

We first define a base language from which to build our translation. In this section we present the calculus λ_D, as a subset of the strongly normalizing Proto-Quipper-D programs. Terms are inductively defined by:

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

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

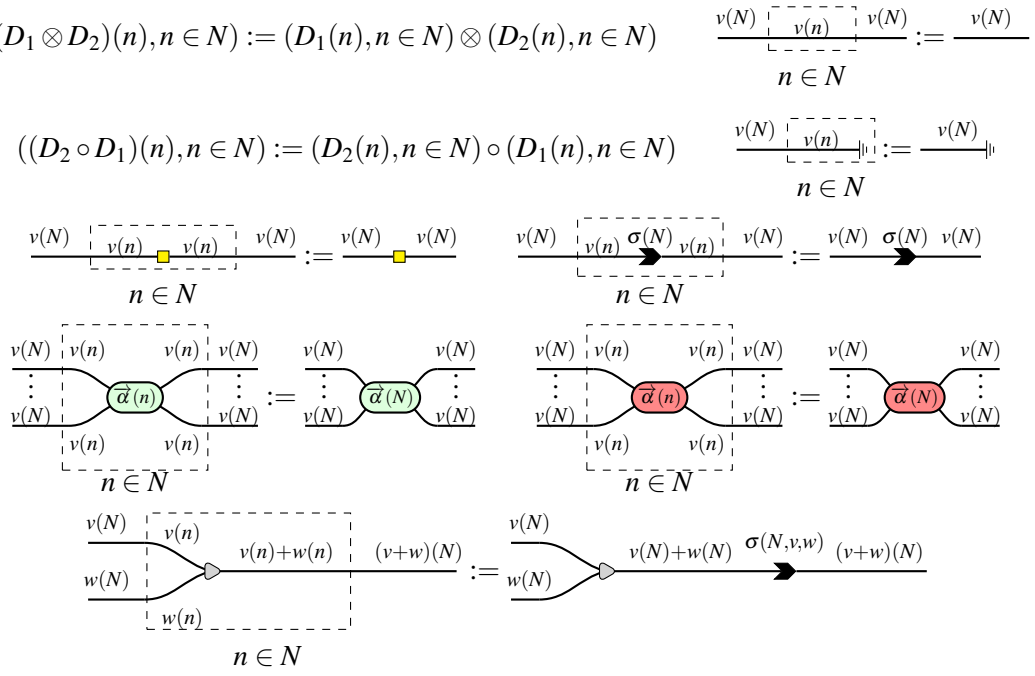
Here `U` denotes a set of unitary operations and `R` is a phase shift gate with a parametrized angle. In this article we fix the former to the CNOT and Hadamard (H) gates, and the latter to the arbitrary rotation gates $R_{z(\alpha)}$ and $R_{x(\alpha)}$.

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 discard 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.

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

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

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



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

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

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

Figure 1: Definition of the list instantiation operator.

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 2. 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 of elements of type A.

We differentiate between *state contexts* (Noted with Γ and Δ) and *parameter contexts* (Noted with Φ). For our case of 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[9].

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

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

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

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

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

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

$$\frac{\Phi, \Gamma \vdash M : A \multimap B \quad \Phi, \Delta \vdash N : A}{\Phi, \Gamma, \Delta \vdash MN : B}\ \multimap_e \qquad \frac{\Phi, \Gamma \vdash M : (n : \mathtt{Nat}) \to B \quad \Phi \vdash N : \mathtt{Nat}}{\Phi, \Gamma \vdash M @ N : B[n/N]}\ \to_e$$

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

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

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

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

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

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

Figure 2: Type system.

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

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

Since every diagram represents a linear map between qubits there is no representation equivalent to non-terminating terms, even for weakly normalizing programs. This is the main reason behind the design choice of the primitives set. We include the operational semantics of the calculus and primitives in Appendix B. The encoding of the primitives as Proto-Quipper-D functions is shown in Appendix C.

We additionally define the following helpful terms based on the previous primitives to aid in the manipulation of vectors. Cf. Appendix B for their definition as λ_D-terms.

$$\Phi \vdash \texttt{map}_{A,B} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ A \multimap \texttt{Vec}\ n\ (A \multimap B) \multimap \texttt{Vec}\ n\ B$$
$$\Phi \vdash \texttt{fold}_{A,C} : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ A \multimap \texttt{Vec}\ n\ (A \multimap C \multimap C) \multimap C \multimap C$$
$$\Phi \vdash \texttt{compose}_A : (n : \texttt{Nat}) \to \texttt{Vec}\ n\ (A \multimap A) \multimap A \multimap A$$

The distinction between primitives that deal with state and parameters highlights the inclusion of the `for` as a construction into the language instead of a primitive. 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 presented in the Section 2.1.

For example, if we take *ns* to be a vector of natural numbers, and *xs* a vector of abstractions $R@k(new0)$. The term `for` k `in` *ns* `do` *xs* generates a vector of quantum maps by instantiating the abstractions for each individual parameter in *ns*.

4 Encoding programs as diagram families

In this section we introduce an encoding of the lambda calculus presented in Section 3 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.

4.1 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:

$$\lfloor x \rfloor_{x:\mathtt{Nat},\Phi} = x, |\Phi| \mapsto x \qquad \lfloor n \rfloor_\Phi = |\Phi| \mapsto n \qquad \lfloor M \square N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi (|\Phi|) \square \lfloor N \rfloor_\Phi (|\Phi|)$$

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

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

$$\lfloor \mathtt{ifz}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \rfloor_\Phi = |\Phi| \mapsto \begin{cases} \lfloor M \rfloor_\Phi (|\Phi|) \text{ if } \lfloor L \rfloor_\Phi (|\Phi|) = 0 \\ \lfloor N \rfloor_\Phi (|\Phi|) \text{ otherwise} \end{cases} \qquad \lfloor \mathtt{range}\ \rfloor_\Phi = n, m, |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i$$

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

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

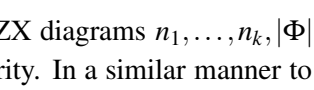
Lemma 4.1 Given an evaluable type A and a type judgement $\Phi \vdash L : A$, $\lfloor L \rfloor_\Phi \in \bigtimes_{x:P \in \Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.

Lemma 4.2 Given an evaluable type A, a type judgement $\Phi \vdash L : A$, and $M \to N$, then $\lfloor M \rfloor_\Phi = \lfloor N \rfloor_\Phi$.

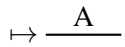
4.2 Diagram encoding. A non-evaluable type has necessarily the form $A = (n_1 : \mathtt{Nat}) \to \cdots \to (n_k : \mathtt{Nat}) \to S$, with S any state type. We call such types *translatable* since they correspond to terms that encode quantum operations that can be described as families of diagrams.

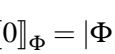
We first define a translation $[\![\cdot]\!]$ from state types into wire multiplicities as follows. Notice that due to the symmetries of the SZX diagrams the linear functions have the same representation as the products.

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

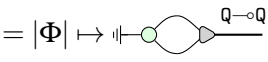
Given a translatable type judgement $\Phi, \Gamma \vdash M : (n_1 : \mathtt{Nat}) \to \cdots \to (n_k : \mathtt{Nat}) \to S$ we can encode it as a family of SZX diagrams $n_1, \ldots, n_k, |\Phi| \mapsto$ 

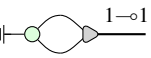
. We will omit the brackets in our diagrams for clarity. In a similar manner to the evaluation, we define the translation $[\![M]\!]_{\Phi,\Gamma}$ recursively on the terms as follows:

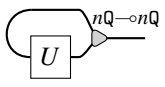
$[\![x]\!]_{\Phi,x:A} = |\Phi| \mapsto$ 

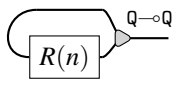
$\quad [\![0]\!]_\Phi = |\Phi| \mapsto$ 

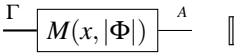
$\quad [\![1]\!]_\Phi = |\Phi| \mapsto$ 

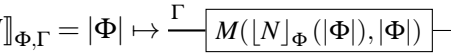
$\quad [\![\mathtt{meas}]\!]_\Phi = |\Phi| \mapsto$ 

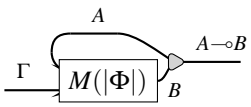
$[\![\mathtt{new}]\!]_\Phi = |\Phi| \mapsto$ 

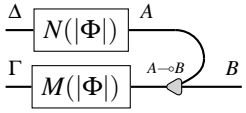
$\quad [\![U]\!]_\Phi = |\Phi| \mapsto$ 

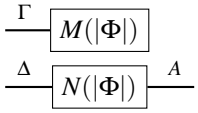
$\quad [\![R]\!]_\Phi = n, |\Phi| \mapsto$ 

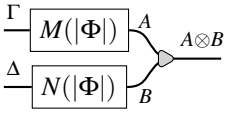
$[\![\lambda' x^A . M]\!]_{\Phi,\Gamma} = x, |\Phi| \mapsto$ 

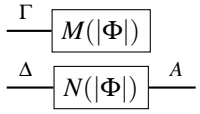
$\quad [\![M\ @N]\!]_{\Phi,\Gamma} = |\Phi| \mapsto$ 

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

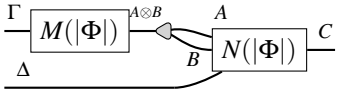
$\quad [\![M\ N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$ 

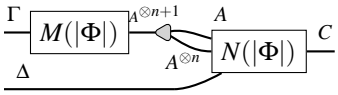
$[\![M;N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$ 

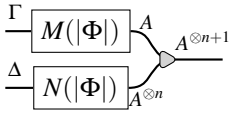
$\quad [\![\star]\!]_\Phi = |\Phi| \mapsto$ 
$\quad [\![M \otimes N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$ 

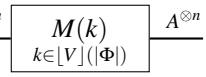
$[\![M ;_v N]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$ 

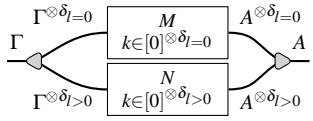
$[\![\texttt{VNil}]\!]_{\Phi} = |\Phi| \mapsto$

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

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

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

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

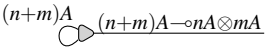
$[\![\texttt{ifz}\ L\ \texttt{then}\ M\ \texttt{else}\ N]\!]_{\Phi,\Gamma} = |\Phi| \mapsto$ 

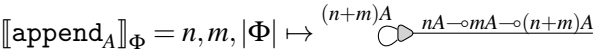
where δ is the Kronecker delta and $l = \lfloor L \rfloor (|\Phi|)$. Notice that the `new` and `meas` operations share the same translation. Although `new` can be encoded as a simple wire, we keep the additional node to maintain the symmetry with the measurement.

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 paper.

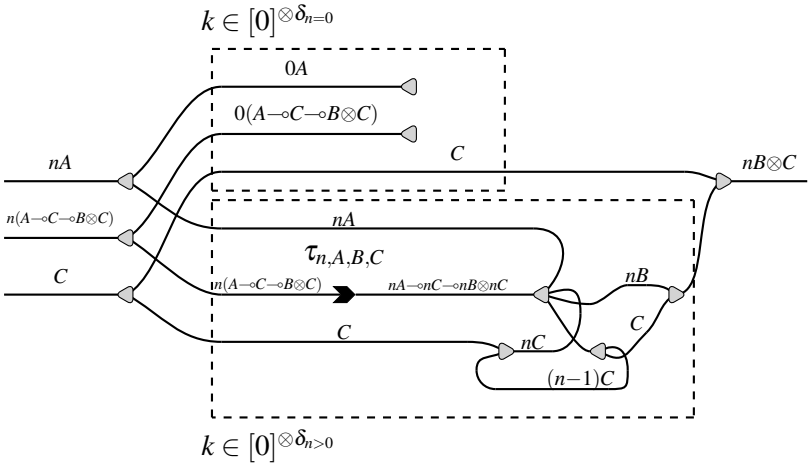
Name	Rz(n)	Rz^{-1}(n)	Rx(n)	Rx^{-1}(n)	H	CNOT
Encoding						

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.

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

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

$[\![\texttt{drop}]\!]_{\Phi} = n, |\Phi| \mapsto$ 

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

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,

$$\tau_{n,A,B,C}(i) = \begin{cases} i \bmod k + a*(i \operatorname{div} k) & \text{if } i \bmod k < a \\ i \bmod k + c*(i \operatorname{div} k) + a*(n-1) & \text{if } a \le i \bmod k < (a+c) \\ i \bmod k + b*(i \operatorname{div} k) + (a+c)*(n-1) & \text{if } (a+c) \le i \bmod k < (a+c+b) \\ i \bmod k + c*(i \operatorname{div} k) + (a+c+b)*(n-1) & \text{if } (a+c+b) \le i \bmod k \end{cases}$$

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. Notice that the ZX spiders, the ground, and the Hadamard operator are only produced in the translations of the quantum primitives. We may instead have used other variations of the calculus supporting the scalable extension, such as the ZH calculus [2], better suited for other sets of quantum operators.

Lemma 4.3 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 $[\![M]\!]_{\Phi,\Gamma} = [\![N]\!]_{\Phi,\Gamma}$.

5 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 [16]. The QFT function operates generically over n-qubit states and in general a circuit encoding of it requires $\mathcal{O}(n^2)$ 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 corresponding Proto-Quipper-D program is listed in Appendix D.

The following presentation divides the algorithm into three parts. The *crot* term applies a controlled rotation over a qubit with a parametrized angle. *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. Finally, *qft* repeats *apply_crot* for all values of k. In the terms, we use $n\dots m$ as a shorthand for `range` $@n\ @m$.

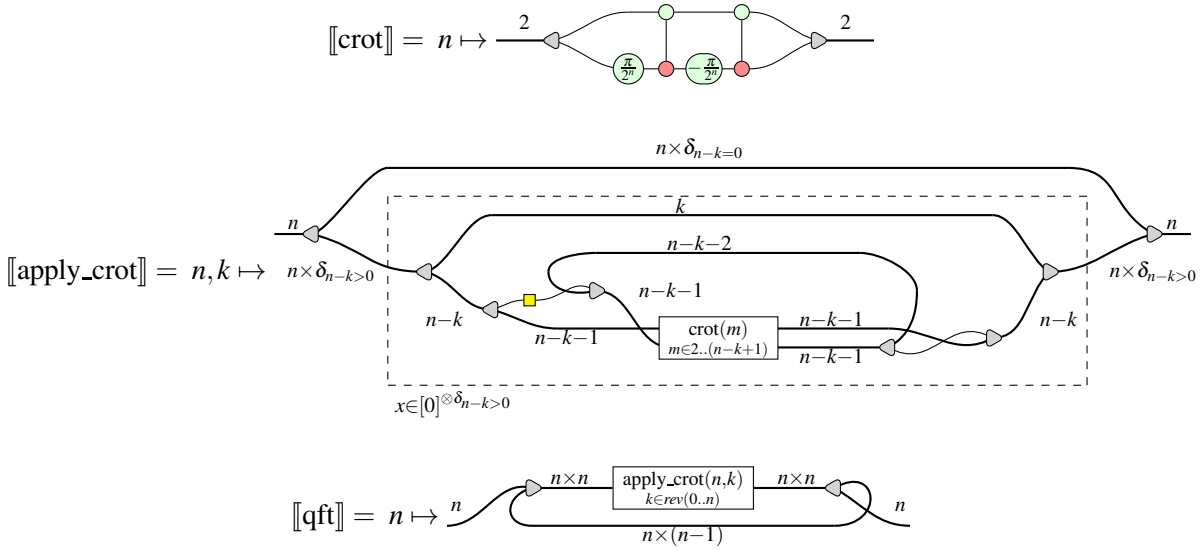
$$\begin{aligned}
&\text{crot} : (n : \texttt{Nat}) \to (\texttt{Q} \otimes \texttt{Q}) \multimap (\texttt{Q} \otimes \texttt{Q}) \\
&\text{crot} := \lambda' n^{\texttt{Nat}}.\lambda qs^{\texttt{Q}\otimes\texttt{Q}}.\texttt{let } c^{\texttt{Q}} \otimes q^{\texttt{Q}} = qs \texttt{ in let } c^{\texttt{Q}}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

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

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

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

The translation of each term into a family of diagrams is shown below. 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**). Notice that, in contrast to a quantum circuit encoding, the resulting diagram's size does not depend on the number of qubits n.



6 Discussion

In this article, we presented an efficient method to compile parametric quantum programs written in a fragment of the Proto-Quipper-D language into families of SZX diagrams. We restricted the fragment to strongly normalizing terms that can be represented as diagrams. Additionally, we introduced a notation to easily compose elements of a diagram family in parallel. We proved that our method produces compact diagrams and shown that it can encode non-trivial algorithms.

A current line of work is defining categorical semantics for the calculus and families of diagrams, including a subsequent proof of adequacy for the translation. More work needs to be done to expand the fragment of the Quipper language that can be translated.

We would like to acknowledge Benoît Valiron for helpful discussion on this topic, and Frank Fu for his help during the implementation of the Proto-Quipper-D primitives. This work was supported in part by the French National Research Agency (ANR) under the research projects SoftQPRO ANR-17-CE25-0009-02 and Plan France 2030 ANR-22-PETQ-0007, by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746-484124, by the project STIC-AmSud project Qapla' 21-SITC-10, the ECOS-Sud A17C03 project, the PICT-2019-1272 project, the PIP 11220200100368CO project, and the French-Argentinian IRP SINFIN.

References

[1] MD SAJID ANIS et al. (2021): *Qiskit: An Open-source Framework for Quantum Computing*, doi:10.5281/zenodo.2562111.

[2] Miriam Backens & Aleks Kissinger (2019): *ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity*. *Electronic Proceedings in Theoretical Computer Science* 287, pp. 23–42, doi:10.4204/eptcs.287.2.

[3] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski & John van de Wetering (2021): *There and back again: A circuit extraction tale*. *Quantum* 5, p. 421, doi:10.22331/q-2021-03-25-421. Available at https://doi.org/10.22331%2Fq-2021-03-25-421.

[4] Niel de Beaudrap & Dominic Horsman (2020): *The ZX calculus is a language for surface code lattice surgery*. *Quantum* 4, p. 218, doi:10.22331/q-2020-01-09-218. Tex.ids: debeaudrapZX-CalculusLanguage2020a arXiv: 1704.08670.

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

[6] Titouan Carette, Yohann D'Anello & Simon Perdrix (2021): *Quantum Algorithms and Oracles with the Scalable ZX-calculus*. *Electronic Proceedings in Theoretical Computer Science* 343, pp. 193–209, doi:10.4204/EPTCS.343.10.

[7] Titouan Carette, Dominic Horsman & Simon Perdrix (2019): *SZX-calculus: Scalable Graphical Quantum Reasoning*. *arXiv:1905.00041 [quant-ph]*, p. 15 pages, doi:10.4230/LIPIcs.MFCS.2019.55. ArXiv: 1905.00041.

[8] Titouan Carette, Emmanuel Jeandel, Simon Perdrix & Renaud Vilmart (2019): *Completeness of Graphical Languages for Mixed States Quantum Mechanics*. arXiv:1902.07143.

[9] Iliano Cervesato & Frank Pfenning (1996): *A linear logical framework*. *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pp. 264–275.

[10] Bob Coecke & Simon Perdrix (2012): *Environment and classical channels in categorical quantum mechanics*. *Logical Methods in Computer Science* Volume 8, Issue 4, doi:10.2168/LMCS-8(4:14)2012.

[11] Cirq Developers (2021): *Cirq*, doi:10.5281/zenodo.5182845. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

[12] Ross Duncan, Aleks Kissinger, Simon Perdrix & John van de Wetering (2019): *Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus*. arXiv:1902.03178.

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

[14] Peng Fu, Kohei Kishida & Peter Selinger (2021): *Linear Dependent Type Theory for Quantum Programming Languages*. *arXiv:2004.13472 [quant-ph]*. ArXiv: 2004.13472.

[15] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *Quipper: a scalable quantum programming language*. *ACM SIGPLAN Notices* 48(6), p. 333, doi:10.1145/2499370.2462177.

[16] Michael A. Nielsen & Isaac L. Chuang (2011): *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press.

[17] Damian S. Steiger, Thomas Häner & Matthias Troyer (2018): *ProjectQ: an open source software framework for quantum computing*. Quantum 2, p. 49, doi:10.22331/q-2018-01-31-49.

[18] John van de Wetering (2020): *ZX-calculus for the working quantum computer scientist*, doi:10.48550/ARXIV.2012.13966.

A Semantics of the SZX calculus

We reproduce below the standard interpretation of $\mathrm{SZX}_{\frac{1}{2}}$ diagrams as density matrices and completely positive maps [8, 6], modulo scalars.

Let $D_n \subseteq \mathbb{C}^{2^n \times 2^n}$ be the set of n-qubit density matrices. We define the functor $\{\!\{\cdot\}\!\} : \mathrm{ZX}_{\frac{1}{2}} \to \mathbf{CPM}(\mathbf{Qubit})$ which associates to any diagram $\mathscr{D} : n \to m$ a completely positive map $\{\!\{\mathscr{D}\}\!\} : D_n \to D_m$, inductively as follows.

$$\{\!\{\mathscr{D}_1 \otimes \mathscr{D}_2\}\!\} := \{\!\{\mathscr{D}_1\}\!\} \otimes \{\!\{D_2\}\!\} \qquad \{\!\{\mathscr{D}_2 \circ \mathscr{D}_1\}\!\} := \{\!\{\mathscr{D}_2\}\!\} \circ \{\!\{\mathscr{D}_1\}\!\}$$

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

$$\left\{\!\!\left\{ \begin{smallmatrix} k & & k \\ & \diagdown\ \diagup & \\ n\ \vdots & \vec{\alpha} & \vdots\ m \\ & \diagup\ \diagdown & \\ k & & k \end{smallmatrix} \right\}\!\!\right\} := \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k} e^{ix \bullet \vec{\alpha}} |x\rangle^{\otimes m} \langle x|^{\otimes n}$$

$$\left\{\!\!\left\{ \begin{smallmatrix} k & & k \\ & \diagdown\ \diagup & \\ n\ \vdots & \vec{\alpha} & \vdots\ m \\ & \diagup\ \diagdown & \\ k & & k \end{smallmatrix} \right\}\!\!\right\} := \{\!\{ \overset{k}{\blacksquare}\!\!-\!\!\blacksquare\overset{k}{} \}\!\}^{\otimes m} \circ \left\{\!\!\left\{ \begin{smallmatrix} k & & k \\ & \diagdown\ \diagup & \\ n\ \vdots & \vec{\alpha} & \vdots\ m \\ & \diagup\ \diagdown & \\ k & & k \end{smallmatrix} \right\}\!\!\right\} \circ \{\!\{ \overset{k}{\blacksquare}\!\!-\!\!\blacksquare\overset{k}{} \}\!\}^{\otimes n}$$

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

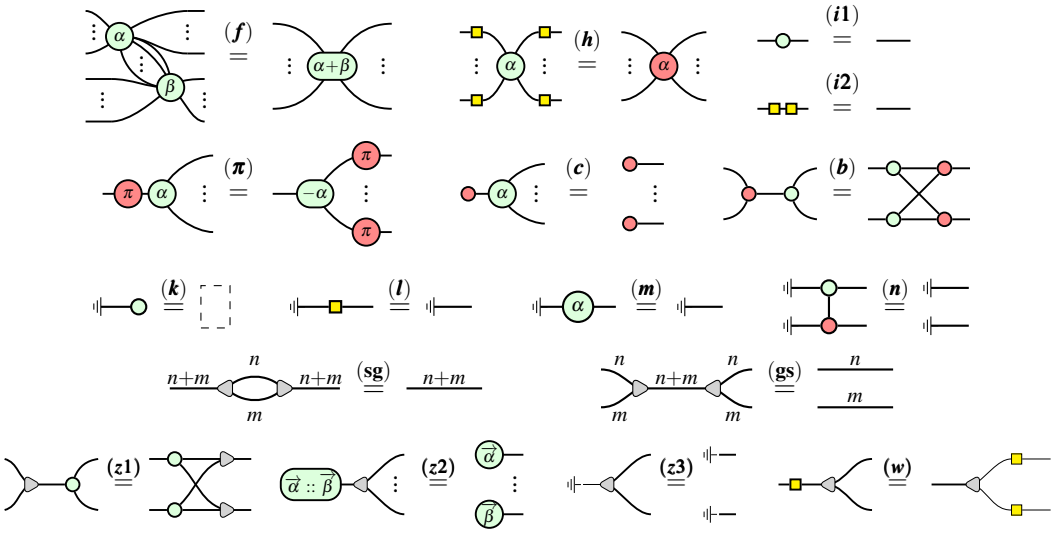
$$\left\{\!\!\left\{ \begin{smallmatrix} n & & \\ & \diagdown & \\ & & n+m \\ & \diagup & \\ m & & \end{smallmatrix} \right\}\!\!\right\} := \rho \mapsto \rho \qquad \{\!\{ \overset{\sigma}{\text{—}\blacktriangleright\text{—}} \}\!\} := \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k} |\sigma(x)\rangle\langle x|$$

$$\{\!\{ \overset{k}{)} \}\!\} := \rho \mapsto \sum_{x \in \mathbb{F}_2^k} \langle xx|\ \rho\ |xx\rangle \qquad \{\!\{ \overset{k}{(} \}\!\} := \sum_{x \in \mathbb{F}_2^k} |xx\rangle\langle xx| \qquad \{\!\{ \overset{}{\text{- -}} \}\!\} := Id_0$$

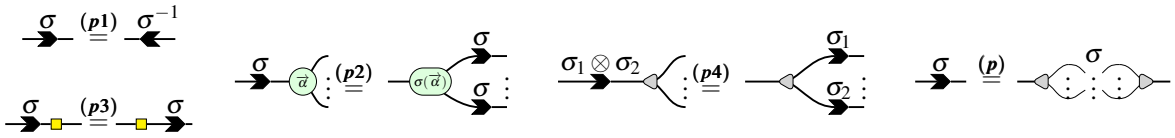
$$\left\{\!\!\left\{ \begin{smallmatrix} k & & l \\ & \diagdown\ \diagup & \\ & \diagup\ \diagdown & \end{smallmatrix} \right\}\!\!\right\} := \rho \mapsto V\ \rho\ V^\dagger \text{ where } V = \sum_{x \in \mathbb{F}_2^k, y \in \mathbb{F}_2^l} |yx\rangle\langle xy|$$

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

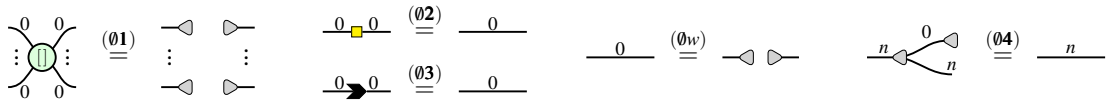
The $\mathrm{SZX}_{\frac{1}{2}}$ calculus defines a set of rewrite rules, shown below.



Additionally, for the arrows restricted to permutations of wires we have the following rules [6]:



Finally, since wires with cardinality zero correspond to empty mappings they can be discarded from the diagrams.



B Operational Semantics of the λ_D calculus

We define a weak call-by-value small step operational semantics on Table 1.

A key point to note here is that every rewriting rule preserves the state. There are no measurements or unitary operations applied, the rewriting is merely syntactical. Since our goal is translation into an SZX-diagram, this system is powerful enough. We include the rewrite rules for the primitives on Table 2.

Additionally, we define useful macros based on these functions on Table 3. They provide syntactic sugar to deal with state vectors.

C Implementation of primitives in Proto-Quipper-D

The implicitly recursive primitives defined in Section 3 can be implemented in proto-quipper-D as follows. The implementation has been checked with the `dpq` tool implemented by Frank Fu (see `https://gitlab.com/frank-peng-fu/dpq-remake`).

```
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
```

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

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

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

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

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

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

$$
\frac{M \to N}{\texttt{ifz } M \texttt{ then } L_1 \texttt{ else } L_2 \to \texttt{ifz } N \texttt{ then } L_1 \texttt{ else } L_2} \qquad \frac{M \to N}{\texttt{for } k \texttt{ in } M \texttt{ do } L \to \texttt{for } k \texttt{ in } N \texttt{ do } L}
$$

Table 1: Rewrite system for λ_D.

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

Table 2: Reductions pertaining to the primitives.

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

Table 3: Function macros.

```
    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'
```

D QFT algorithm in Quipper code

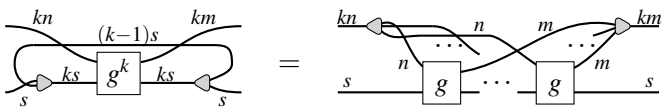
The following Proto-Quipper-D code corresponds to the algorithm presented in Section 5. This implementation has been checked with the `dpq` tool implemented by Frank Fu (see `https://gitlab.com/frank-peng-fu/dpq-remake`). Notice that, in contrast to the presented lambda terms, the type checker

implementation requires explicit encodings of the Leibniz equalities between parameter types.

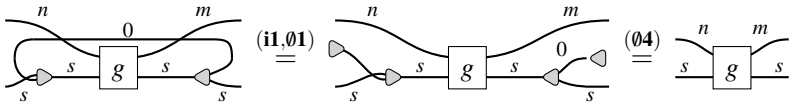
```
module Qft where
import "/dpq/Prelude.dpq"

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

-- 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')

-- 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
```

E Proofs

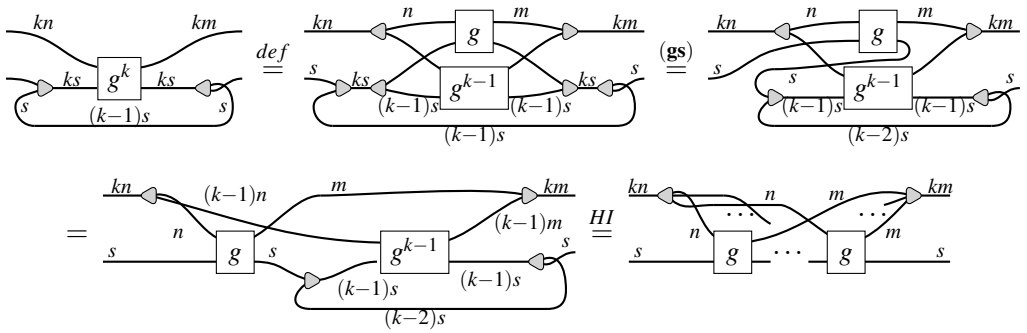
Lemma (2.1) Let $g : 1_n \otimes 1_s \to 1_m \otimes 1_s$ and $k \geq 1$, then



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

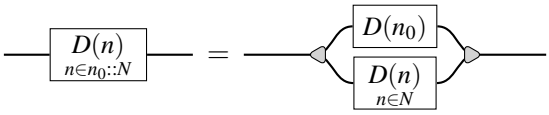


If $k > 1$,



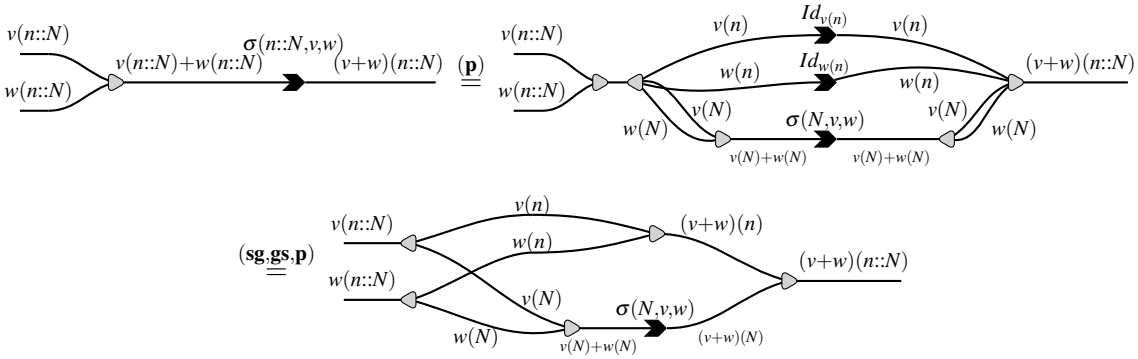
□

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



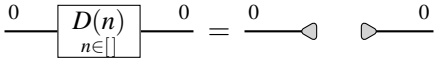
Proof By induction on the term construction

- If $\mathscr{D}$ 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 $\mathscr{O}(n)$ 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. □

Lemma E.5 Given type judgements $\Phi, x:A \vdash M:B$, and $\Phi \vdash N:A$. $\lfloor M \rfloor_{x:A,\Phi}(\lfloor N \rfloor_\Phi, |\Phi|) = \lfloor M[N/x] \rfloor_\Phi(|\Phi|)$.

Proof Proof by straightforward induction on M. □

Lemma (4.1) Given an evaluable type A and a type judgement $\Phi \vdash M:A$, $\lfloor M \rfloor_\Phi \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.

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

- If $\Phi \vdash n : \mathtt{Nat}$, 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 $\Phi \vdash M \Box N : \mathtt{Nat}\Phi, \Gamma, \Delta \vdash M :: N : \mathtt{Vec}\ (n+1)\ A$, then $\lfloor M \Box N \rfloor_\Phi = |\Phi| \mapsto \lfloor M \rfloor_\Phi(|\Phi|) \Box \lfloor N \rfloor_\Phi(|\Phi|)$. By inductive hypothesis, $\lfloor M \rfloor_\Phi(|\Phi|), \lfloor M \rfloor_\Phi(|\Phi|) \in \mathbb{N}$. Then, $|\Phi| \mapsto \lfloor M \rfloor_\Phi(|\Phi|) \Box \lfloor N \rfloor_\Phi(|\Phi|) \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \mathbb{N}$.
- 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 $x \mapsto \lfloor M \rfloor_\Phi(x, \Phi) \in \lfloor x : \mathtt{Nat} \to B[x] \rfloor$. 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 $\Phi \vdash \mathtt{VNil}^A : \mathtt{Vec}\ 0\ A$, then $\left\lfloor \mathtt{VNil}^P \right\rfloor_\Phi = |\Phi| \mapsto [] \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \mathbb{N}^0$.
- If $\Phi, \Gamma, \Delta \vdash M :: N : \mathtt{Vec}\ (n+1)\ A$, 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 $\Phi, \Gamma \vdash \mathtt{ifz}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N : A$, then
  $\lfloor \mathtt{ifz}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \rfloor_\Phi = |\Phi| \mapsto \begin{cases} \lfloor M \rfloor_\Phi(|\Phi|) \text{ if } \lfloor L \rfloor_\Phi(|\Phi|) = 0 \\ \lfloor N \rfloor_\Phi(|\Phi|) \text{ otherwise} \end{cases}$.
  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 $\lfloor \mathtt{ifz}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N \rfloor_\Phi \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.
- If $\Phi \vdash \mathtt{for}\ k\ \mathtt{in}\ N\ \mathtt{do}\ M : \mathtt{Vec}\ n\ A$, then $\lfloor \mathtt{for}\ k\ \mathtt{in}\ V\ \mathtt{do}\ M \rfloor_\Phi = |\Phi| \mapsto \bigtimes_{k\in\lfloor N \rfloor_\Phi(|\Phi|)} \lfloor M \rfloor_{k:\mathtt{Nat}\Phi}(k, |\Phi|)$. By induction hypothesis, $\lfloor N \rfloor_\Phi(|\Phi|) \in \mathtt{Nat}^n$ and $x \mapsto \lfloor M \rfloor_\Phi(x, \Phi) \in \lfloor k : \mathtt{Nat} \to A[k] \rfloor$. Then $|\Phi| \mapsto \bigtimes_{k\in\lfloor N \rfloor_\Phi(|\Phi|)} \lfloor M \rfloor_{k:\mathtt{Nat}\Phi}(k, |\Phi|) \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \lfloor A[k/N] \rfloor^n$.
- If $\Phi \vdash \mathtt{let}\ x^B : y^{\mathtt{Vec}\ n\ B} = M\ \mathtt{in}\ N : A$, then $\left\lfloor \mathtt{let}\ x^P :: y^{\mathtt{Vec}\ n\ P} = M\ \mathtt{in}\ N \right\rfloor_\Phi = |\Phi| \mapsto \lfloor N \rfloor_{x:P,y:\mathtt{Vec}\ n\ P\Phi}(y_1, [y_2, \ldots, y_n], |\Phi|)$ 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 $\lfloor N \rfloor_{x:P,y:\mathtt{Vec}\ n\ P\Phi}(y_1, [y_2, \ldots, y_n], |\Phi|) \in \lfloor A \rfloor$. Then $|\Phi| \mapsto \lfloor N \rfloor_{x:P,y:\mathtt{Vec}\ n\ P\Phi}(y_1, [y_2, \ldots, y_n], |\Phi|) \in \bigtimes_{x:P\in\Phi} \lfloor P \rfloor \to \lfloor A \rfloor$.
- If $\Phi \vdash \mathtt{range} : (n:\mathtt{Nat}) \to (m:\mathtt{Nat}) \to \mathtt{Vec}\ (m-n)\ \mathtt{Nat}$, then $\lfloor \mathtt{range} \rfloor_\Phi = n, m, |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i \in \bigtimes_{z:P\in x:\mathtt{Nat},y:\mathtt{Nat},\Phi} \lfloor P \rfloor \to \mathbb{N}^{m-n}$ □

Lemma (4.2) Given an evaluable type A, a type judgement $\Phi \vdash M:A$, 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 $M \to M_1 \square N$ with $M_2 \to N$, then:

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

  - If $M \to N \square V$ with $M_1 \to N$, then:

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

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

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

- 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:

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

- 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:

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

  - If $M \to L$ with $M' = 0$, then:

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

  - 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:

$$\begin{aligned}\left\lfloor \texttt{for}\ k\ \texttt{in}\ M'\ \texttt{do}\ R \right\rfloor_\Phi &= |\Phi| \mapsto \bigtimes_{k \in \lfloor M' \rfloor_\Phi(|\Phi|)} \lfloor R \rfloor_\Phi(k, |\Phi|) \\ &= |\Phi| \mapsto \bigtimes_{k \in \lfloor N \rfloor_\Phi(|\Phi|)} \lfloor R \rfloor_\Phi(k, |\Phi|) \\ &= \left\lfloor \texttt{for}\ k\ \texttt{in}\ N\ \texttt{do}\ R \right\rfloor_\Phi\end{aligned}$$

  - If $M \to R[k/M_1]\ :\texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R$ with $M' = V :: L$, then:

$$\begin{aligned}\left\lfloor \texttt{for}\ k\ \texttt{in}\ M_1\ ::\ M_2\ \texttt{do}\ R \right\rfloor_\Phi &= |\Phi| \mapsto \bigtimes_{k \in \lfloor M_1\ ::\ M_2 \rfloor_\Phi(|\Phi|)} \lfloor R \rfloor_\Phi(k, |\Phi|) \\ &= |\Phi| \mapsto \bigtimes_{k \in \lfloor M_1 \rfloor_\Phi(|\Phi|) \times \lfloor M_2 \rfloor_\Phi(|\Phi|)} \lfloor R \rfloor_\Phi(k, |\Phi|) \\ &= |\Phi| \mapsto \lfloor R \rfloor_{k,\Phi}(\lfloor M_1 \rfloor_\Phi(|\Phi|), |\Phi|) \times \bigtimes_{k \in \lfloor M_2 \rfloor_\Phi(|\Phi|} \lfloor R \rfloor_{k,\Phi}(k, |\Phi|) \\ &= \lfloor R[M_1/k] \rfloor_\Phi \times \left\lfloor \texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R \right\rfloor_\Phi \\ &= \lfloor R[M_1/k]\ ::\ \texttt{for}\ k\ \texttt{in}\ M_2\ \texttt{do}\ R \rfloor_\Phi\end{aligned}$$

  - 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:

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

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

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

    where $[y_1, \ldots, y_n] = \lfloor M_1 :: M_2 \rfloor_\Phi (|\Phi|)$.
- If $M = \texttt{range}\ \ @n\ @M_2$ then:

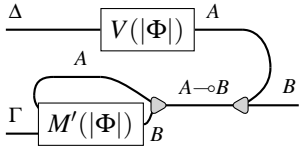
$$\begin{aligned}\lfloor \texttt{range}\ @n @m \rfloor_\Phi &= |\Phi| \mapsto \bigtimes_{i=n}^{m-1} i \\ &= |\Phi| \mapsto \begin{cases} [] \text{ if } n - m = 0 \\ n \times \bigtimes_{i=n+1}^{m-1} i \text{ otherwise} \end{cases} \\ &= |\Phi| \mapsto \begin{cases} [] \text{ if } \lfloor n - m \rfloor_\Phi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

Lemma (4.3) The translation procedure is correct in respect to the operational semantics. If A is a translatable type, $\Phi, \Gamma \vdash M : A$, and $M \to N$, then $[\![M]\!]_{\Phi,\Gamma} = [\![N]\!]_{\Phi,\Gamma}$.

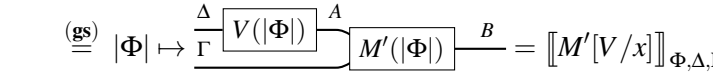
Proof By case analysis on the reductions of translatable terms.

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

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



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



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

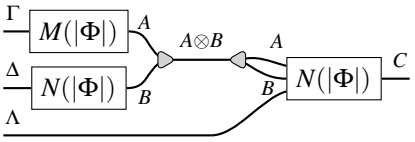
- If $M = (\lambda' x^A.M')@V$ and $N = M'[V/x]$,

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

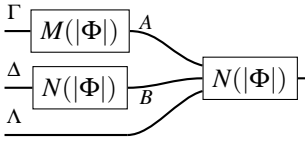
$$= |\Phi| \mapsto \overset{\Gamma}{\boxed{\lfloor (M') \rfloor_\Phi (\lfloor V \rfloor_\Phi (|\Phi|), |\Phi|)}}\ \overset{B}{} = [\![M'[V/x]]\!]_{\Phi,\Gamma}$$

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

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



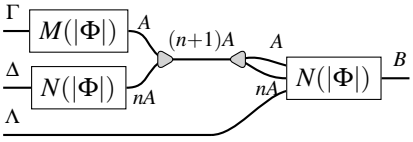
$$\overset{(\mathbf{gs})}{=} |\Phi| \mapsto$$



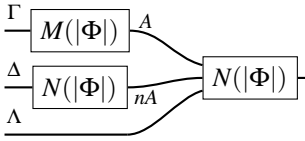
$$= \left[\!\left[M'[V_1/x][V_2/y]\right]\!\right]_{\Phi,\Delta,\Gamma,\Lambda}$$

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

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



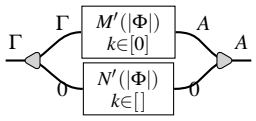
$$\overset{(\mathbf{gs})}{=} |\Phi| \mapsto$$



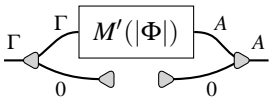
$$= \left[\!\left[M'[V_1/x][V_2/y]\right]\!\right]_{\Phi,\Delta,\Gamma,\Lambda}$$

- If $M = \texttt{ifz}\ L\ \texttt{then}\ M'\ \texttt{else}\ N'$,
  - if $L = 0$ and $N = M'$, $\lfloor L \rfloor_\Phi = 0$ and

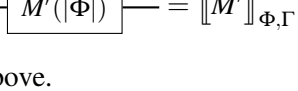
$$\left[\!\left[\texttt{ifz}\ L\ \texttt{then}\ M'\ \texttt{else}\ N'\right]\!\right]_{\Phi,\Gamma} = |\Phi| \mapsto$$



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



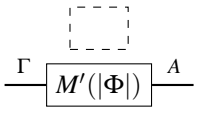
$$\overset{(\mathbf{04})}{=} |\Phi| \mapsto$$



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

  - The case where $L > 0$ and $N = N'$ is symmetric to the case above.
- If $M = \texttt{VNil};M'$ and $N = M'$,

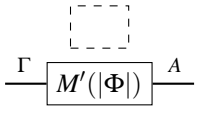
$$\left[\!\left[\texttt{VNil};_\nu M'\right]\!\right]_{\Phi,\Gamma} = |\Phi| \mapsto$$



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

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

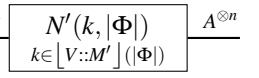
$$\left[\!\left[\star;M'\right]\!\right]_{\Phi,\Gamma} = |\Phi| \mapsto$$

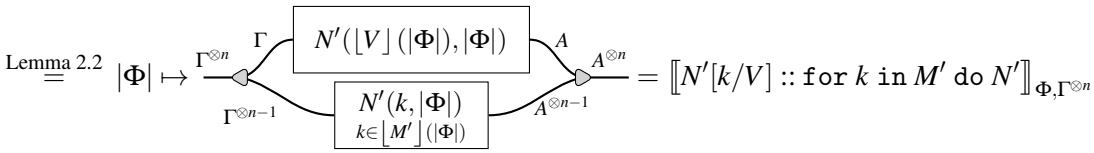


$$= \left[\!\left[M'\right]\!\right]_{\Phi,\Gamma}$$

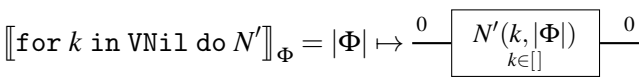
- If $M = \texttt{for}\ k\ \texttt{in}\ V :: M'\ \texttt{do}\ N'$ and $N = N'[k/V] :: \texttt{for}\ k\ \texttt{in}\ M'\ \texttt{do}\ N'$,

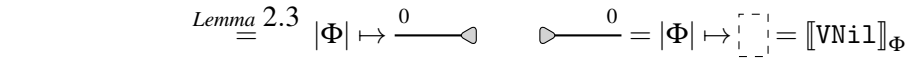
$$\left[\!\left[\texttt{for}\ k\ \texttt{in}\ V :: M'\ \texttt{do}\ N'\right]\!\right]_{\Phi,\Gamma^{\otimes n}} = |\Phi| \mapsto$$





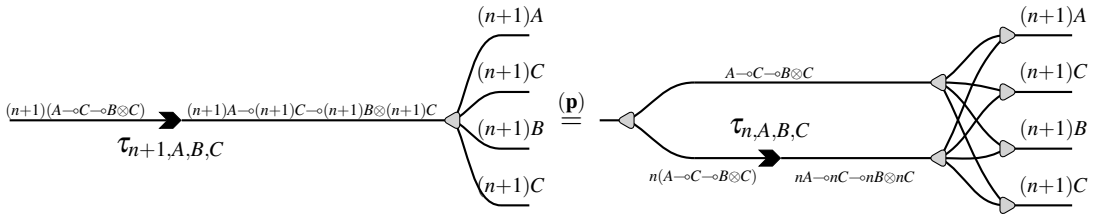
- If $M = \texttt{for}\ k\ \texttt{in}\ \texttt{VNil}\ \texttt{do}\ N'$ and $N = \texttt{VNil}$,





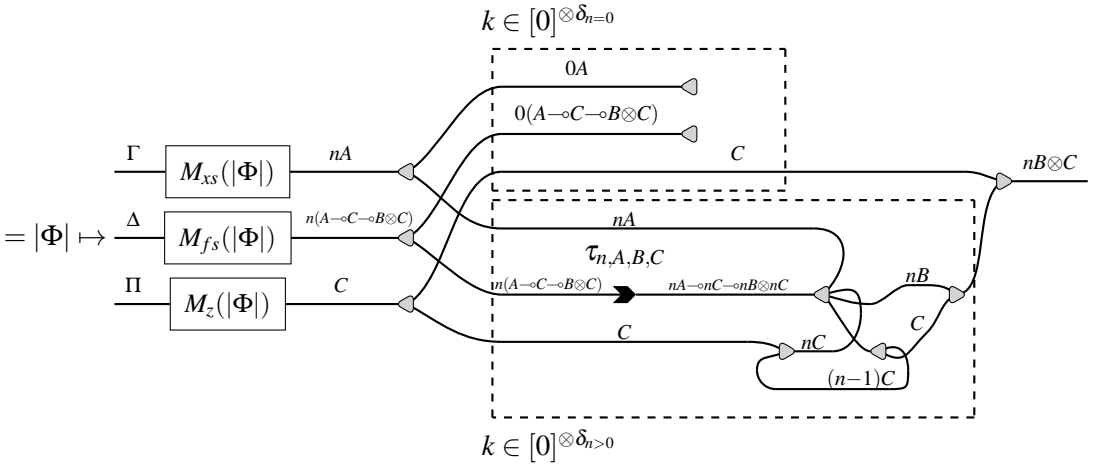
- If $M = \texttt{accuMap}_{A,B,C}\ N'\ M_{xs}\ M_{fs}\ M_z$ and $N = \texttt{ifz}\ N'\ \texttt{then}\ xs\ ;_v\ fs\ ;_v\ \texttt{VNil} \otimes M_z\ \texttt{else}\ \texttt{let}\ x :: xs' = M_{xs}\ \texttt{in}\ \texttt{let}\ f :: fs' = M_{fs}\ \texttt{in}\ \texttt{let}\ y \otimes z' = f\ x\ z\ \texttt{in}\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

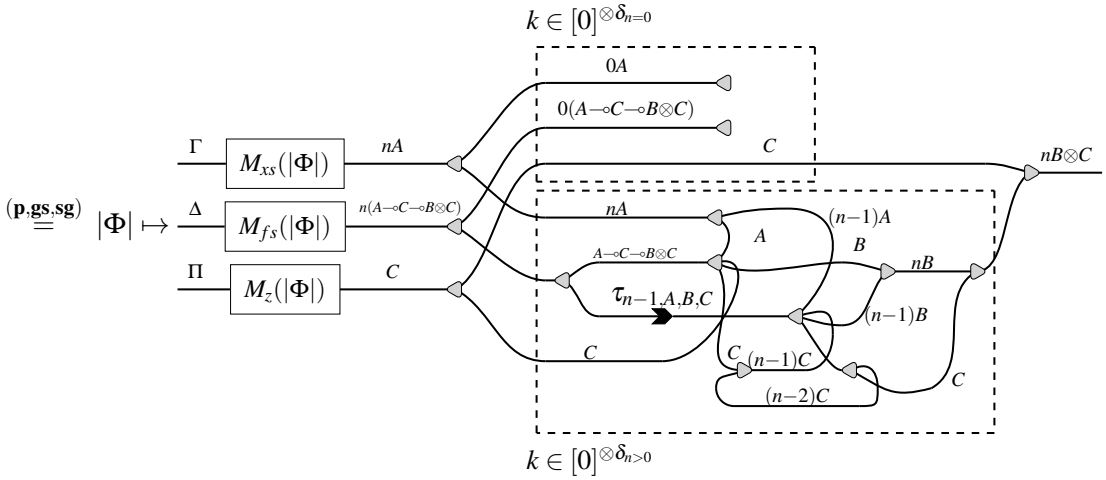
  Notice that, by definition of $\tau_{n,A,B,C}$,



  Therefore,

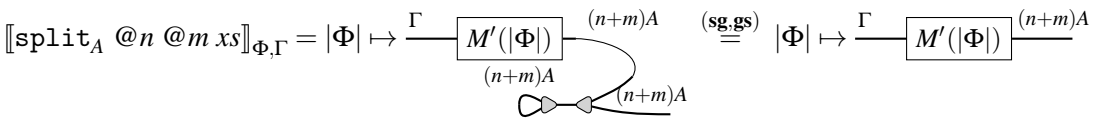
  $[\![\texttt{accuMap}_{A,B,C}\ N'\ M_{xs}\ M_{fs}\ M_z]\!]_{\Phi,\Gamma,\Delta,\Pi}$

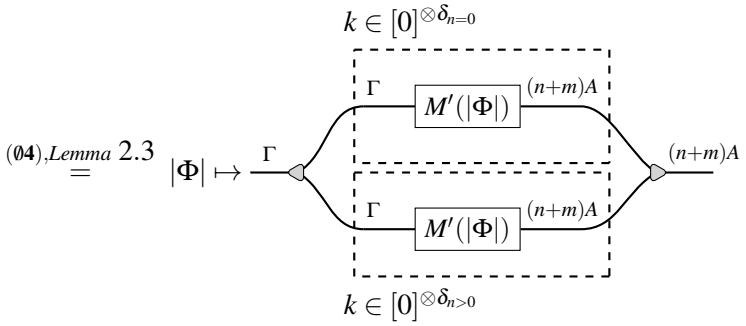


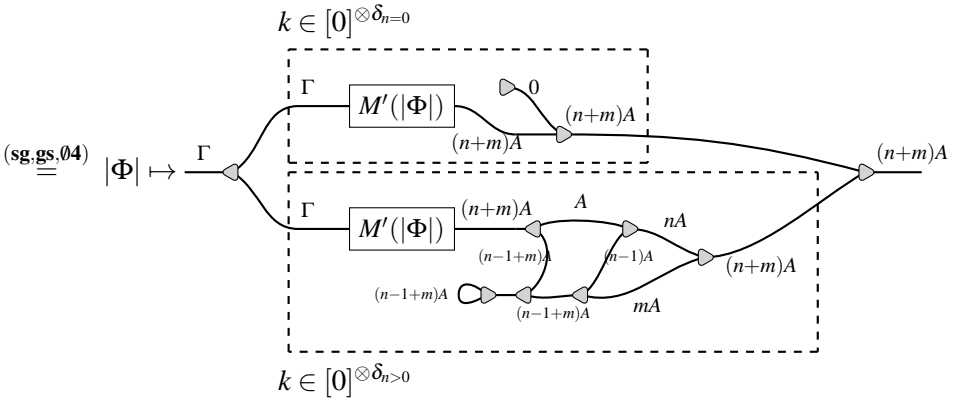


$= [\![\texttt{ifz}\ N' \ \texttt{then}\ xs\ ;_v\ fs\ ;_v\ \texttt{VNil} \otimes M_z\ \texttt{else}\ \texttt{let}\ x :: xs' = M_{xs}\ \texttt{in}\ \texttt{let}\ f :: fs' = M_{fs}\ \texttt{in}\ \texttt{let}\ y \otimes z' = f\ x\ z\ \texttt{in}\ \texttt{let}\ ys \otimes z'' = \texttt{accuMap}\ @(N'-1)\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

- 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 $m = \lfloor N_2 \rfloor_\Phi(|\Phi|)$.



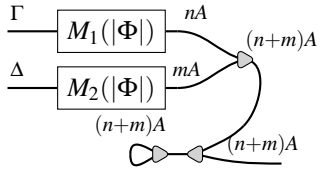




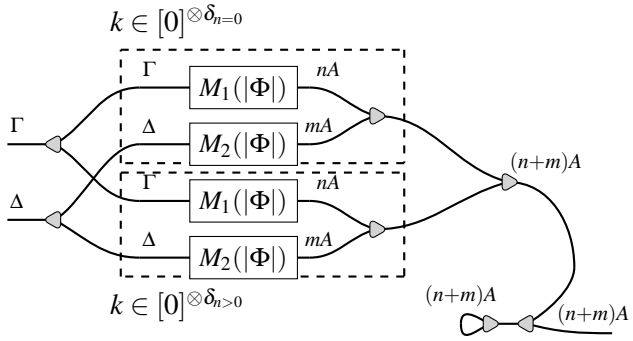
$= [\![\texttt{ifz}\ n\ \texttt{then}\ \texttt{VNil} \otimes xs\ \texttt{else}\ \texttt{let}\ y :: xs' = xs\ \texttt{in}\ \texttt{let}\ ys_1 \otimes ys_2 = \texttt{split}@(n-1)\ @m\ xs'\ \texttt{in}\ (y :: ys_1) \otimes ys_2]\!]_{\Phi,\Gamma}$

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

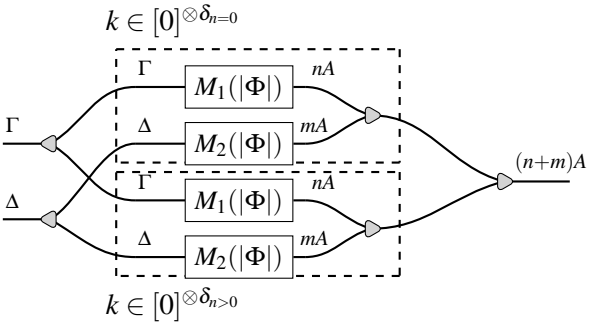
$$[\![\texttt{append}_A\ @N_1\ @N_2\ M_1\ M_2]\!]_{\Phi,\Gamma,\Delta} = |\Phi| \mapsto$$



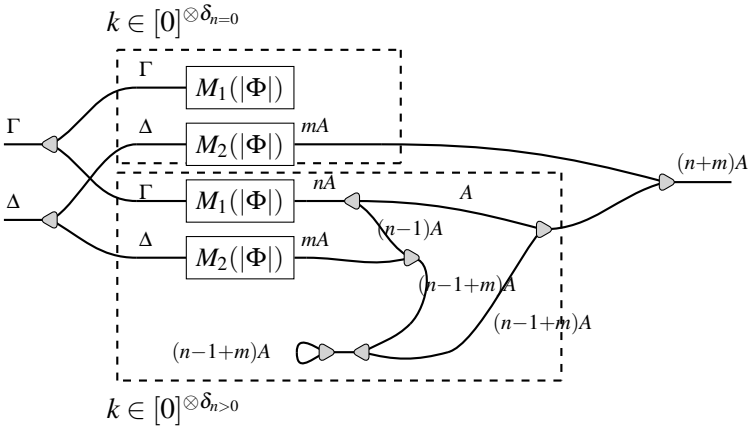
$$\overset{(\emptyset\mathbf{4}),\textit{Lemma } 2.3}{=} |\Phi| \mapsto$$



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



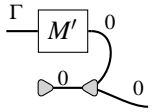
$$\overset{(\mathbf{i1},\emptyset\mathbf{1},\emptyset\mathbf{4},\mathbf{sg},\mathbf{gs})}{=} |\Phi| \mapsto$$

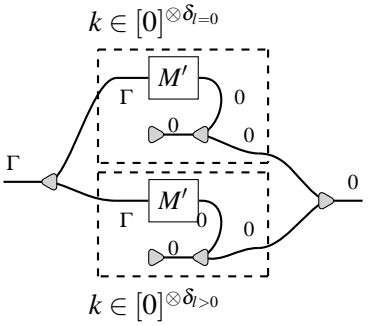


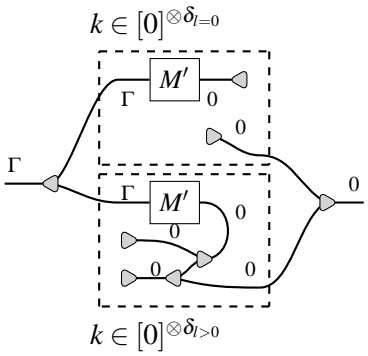
$$= [\![\texttt{ifz}\ N_1\ \texttt{then}\ M_1\ ;_\nu\ M_2\ \texttt{else}\ \texttt{let}\ x :: xs' = M_1\ \texttt{in}\ x :: (\texttt{append}\ @(N_1 - 1)\ @N_2\ M_1\ M_2]\!]_{\Phi,\Gamma,\Delta}$$

- If $M = \texttt{drop}\ @N'\ M'$ and $N = \texttt{ifz}\ N'\ \texttt{then}\ M'\ ;\ \star\ \texttt{else}\ \texttt{let}\ x :: xs' = M'\ \texttt{in}\ x\ ;\ \texttt{drop}\ @(N' -$

1) xs'. Let $l = \lfloor N' \rfloor_{\Phi}(|\Phi|)$,

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

$\overset{(\emptyset\mathbf{4}),\textit{Lemma}\ 2.3}{=} |\Phi| \mapsto$ 

$\overset{(\mathbf{z3},\emptyset\mathbf{4})}{=} |\Phi| \mapsto$ 

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

- If $M \to N$ is an internal reduction of a translatable term, then the diagrams result equivalent via the inductive hypothesis.
- If $M \to N$ is an internal reduction of an evaluable term, then the diagrams result equivalent via the inductive hypothesis and Lemma 4.2. □