Structural Proto-Quipper: Mechanization of a Linear Quantum Programming Language in a Structural Setting

Max Gross
max.gross@mail.mcgill.ca
Department of Mathematics and Statistics, McGill University
Montreal, Quebec, Canada

Abstract

This project aims to develop a general technique for mechanizing quantum programming languages that rely on linear logic for resource management, such as the handling of qubits. The main challenge is integrating linear logic, which enforces constraints like one-time use of resources, into systems built on structural logic frameworks, like the Higher Order Abstract Syntax. To address this, the project adopts the strategy of "enforcing linearity without linearity." As a proof of concept, we mechanize and improve Proto-Quipper, a quantum programming language used to generate circuits, utilizing Beluga, a tool for formal reasoning about systems. The approach is grounded in Crary's method for representing Girard's linear logic in Twelf, and introduces two linearity predicates that ensure classical and quantum variables are used linearly within typing judgments. This technique leverages Beluga's HOAS to streamline proofs and avoids the need for external extensions, unlike prior work by Mahmoud et al., who mechanized Proto-Quipper using a linear extension of Hybrid in Coq. Further, we optimize Proto-Quipper's treatment of circuits for mechanization by treating them as functions in a linear lambda calculus. While the mechanization of Proto-Quipper is successful, demonstrating the soundness of the approach, proofs for key properties such as subject reduction and progress are still under development. Future efforts will focus on completing these proofs and expanding the method to more complex quantum programming languages, such as Proto-Quipper-Dyn, which introduces dynamic lifting.

CCS Concepts

• **Theory of computation** → **Logic and verification**; **Quantum computation theory**.

Keywords

linear logic, quantum lambda calculus, verification, logical framework, Proto-Quipper, linear predicate

ACM Reference Format:

Max Gross. 2025. *Structural Proto-Quipper*: Mechanization of a Linear Quantum Programming Language in a Structural Setting. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, ?? pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The concept of quantum computation is relatively new, with Paul Benioff and Richard Feynman first proposing in 1982 that quantum systems could be used to perform computations. Feynman, in particular, argued that it was "impossible to represent the results of quantum mechanics with a classical universal device."[**?**] He envisioned quantum computers as tools for simulating quantum systems more efficiently than classical computers, which are limited by the prohibitive cost of modeling quantum processes[**?**][**?**].

In the 1990s, Peter Shor introduced algorithms to solve cryptographic problems, like factoring large integers and the discrete logarithm problem, that classical computers handle inefficiently. Shor's breakthrough highlighted the unique potential of quantum computers, sparking a surge of interest in their design. However, it wasn't until 1996 that researchers began to discuss the need for quantum programming languages, hindered by the lack of practical quantum hardware to support them[**?**]. Even today, achieving "quantum supremacy," where quantum computers outperform classical ones on complex tasks, remains an ongoing challenge.

Gay counters criticism regarding quantum programming languages by pointing out that, in classical computing, the lack of a solid semantic foundation in programming languages has led to significant issues in software engineering. He argues that allowing the programming cart to precede the computing horse can be beneficial[**?**]. This paper aligns with that perspective, as formal verification of programming languages has also faced criticism for being overly theoretical and detracting from practical language design[**?**]. However, coming together, quantum computation presents a compelling case for mechanization. Quantum programs are notoriously challenging to reason about due to the probabilistic nature of quantum mechanics, along with concepts like entanglement and superposition. Formal verification and proof assistants can play a crucial role in rigorously proving the correctness of quantum algorithms, thereby enhancing their reliability and effectiveness.

The primary challenge of mechanizing quantum programming languages lies in their reliance on linear logic to model quantum resources, specifically a linear lambda calculus with a typing system based on Girard's linear logic[**?**]. This approach stems from the "no-cloning property," which states that it is impossible to create perfect copies of an unknown quantum state[**?**].

In most encodings of linear programming languages, contexts are treated explicitly (for example, as lists). However, this becomes unwieldy when formalizing meta-theoretical statements, as it requires managing context-dependent operations while keeping track

of resource consumption. In the context of mechanizing quantum programming languages, properties of bindings in the host language using Higher Order Abstract Syntax can significantly aid in verifying aspects like alpha equivalence and substitution lemmas. However, existing HOAS frameworks manage contexts in a structural manner, which poses challenges for quantum computing languages that necessitate linear contexts. This discrepancy means that applying HOAS to handle quantum wires naively may not ensure linearity, complicating the proof of crucial meta-theoretic properties[**?**]. To effectively mechanize quantum programming languages and capture the essential linearity required, advancements in HOAS approaches are necessary.

In this paper, we present a use case for a novel technique that employs higher-order abstract syntax to mechanize linear quantum programming languages, originally proposed by Sano et al[**?**]. for use in concurrent settings and adapted from work by Crary[**?**]. Specifically, we offer a proof-of-concept mechanization of Proto-Quipper[**?**], a small circuit-building quantum programming language based on the aforementioned quantum lambda calculus, in Beluga, along with mechanization-facing improvements to its treatment of circuits.

To that end, we introduce Structural Proto-Quipper (SPQ), noting that its context (as in Beluga) is structural. Yet, we encode linearity using linearity predicates on typing judgements in the language, defining

$$\texttt{lin}\ \ (x : B;\ \Gamma, x : B \vDash a : A), \qquad \texttt{lin/q}\ \ (q;\ \Gamma, \{q\} \vDash a : A)$$

saying that within some typing judgment that a is of type A, where a depends on a variable x or a quantum variable q, x or q are used linearly. While this predicate is local to a typing judgment, by virtue of its application each time we use typing rules that add variable or quantum variable bindings, it is the case that SPQ operates globally linearly, without linearity. Further work will be made in proving something like adequacy between SPQ and PQ, although certain key differences will make this tricky.

This is because, in Proto-Quipper, circuits are treated abstractly as uninterpreted circuit constants coming from a countable set, where it assumes "that there exists a constant symbol for every possible quantum circuit."[**?**]. Operations on circuits, such as appending two circuits together, is interpreted as operations on lists - the exact problem we are avoiding in this paper. Thus, SPQ amends PQ to treat circuits as functions in a linear lambda calculus taking and returning inputs as tuples of wires as quantum variables. We also have a linearity predicate

$$\texttt{clamlin}\ \ (\lambda q.\ M)$$

saying that a quantum variable is being used linearly in an expression.

Note that our linearity approach at the term level differs from Sano et al. in that their linearity predicate is defined with respect to processes themselves, not typing judgments on them[**?**]. This is due to a feature of Proto-Quipper such that terms are the same whether or not they are of a linear or non-linear type, and thus we must type them to determine if linearity is required. There is a simple rule saying that linearity always holds on these shared resources. Note further that while Proto-Quipper has been mechanized by Mahmoud et al.[**?**], they did so using Hybrid, a linear extension to Coq. Our contribution is a mechanization within the logical framework itself.

2 Introduction to Linearity and Quantum Programming

Linear logic is a form of *substructural logic* that differs from classical structural logics by omitting two key rules: *contraction* and *weakening*. These rules, which are fundamental in traditional logics, are generalized as follows:

$$\frac{\Gamma \vdash t}{\Gamma, x \vdash t}\ \text{[Weakening-L]} \qquad \frac{\Gamma \vdash t}{\Gamma \vdash t, x}\ \text{[Weakening-R]}$$

$$\frac{\Gamma, x, x \vdash t}{\Gamma, x \vdash t}\ \text{[Contraction-L]} \qquad \frac{\Gamma \vdash t, t}{\Gamma \vdash t}\ \text{[Contraction-R]}$$

Linear logic introduces a discipline on resource management by rejecting these rules. In particular, it does not allow assumptions (resources) to be arbitrarily duplicated (*contraction*) or discarded (*weakening*).

The *linear lambda calculus* serves as the computational counterpart to linear logic, much like how the traditional lambda calculus corresponds to classical structural logic. In this sense, proofs in linear logic are represented as terms in the linear lambda calculus.

Semantically, linear logic was originally developed as a formal system to study *resource availability* [**?**]. In this framework, propositions are treated as resources that must be consumed exactly once—they cannot be freely duplicated (no contraction) or ignored (no weakening). This perspective has deep connections to quantum computing, where linear logic provides an ideal model for reasoning about quantum systems.

More specifically, linear logic has been interpreted as a form of *quantum logic* [**?**]. The absence of weakening mirrors the *no-cloning theorem* of quantum computation, which states that arbitrary quantum states cannot be copied. Similarly, the absence of contraction corresponds to the *no-deletion rule*, which forbids the arbitrary erasure of quantum information.

Building on these ideas, the *quantum lambda calculus* [**?**] was introduced as a computational model grounded in the principles of linear logic. This work led to the development of *Quipper* [**?**], a practical quantum programming language designed to apply formal methods to quantum algorithm analysis.

Our focus in this project is on mechanizing *Proto-Quipper*, which is "a limited (but still expressive) fragment of the Quipper language... [designed to be] completely type-safe" [**?**]. Proto-Quipper retains the resource-sensitive principles of linear logic and provides a foundation for rigorous reasoning about quantum programs within a type-theoretic framework..

3 Linearity Predicate

Our main goal is to leverage Higher Order Abstract Syntax (HOAS) to manage contexts and variable substitutions. Our contribution, therefore, is an encoding within the Logical Framework (LF) itself using the technique of linearity predicates, as opposed to relying on extended libraries, as done in previous work in the area [**?**]. Crary initially envisioned such a technique to encode the linear lambda calculus in Twelf [**?**]. Further work by Sano et al. [**?**] mechanized

Wadler's Classical Processes [**?**], a system based on linear logic, much like Proto-Quipper.

Crary argues that designing a technique for mechanization within the LF layer is worthwhile for several reasons. First, there is the question of accessibility: do researchers engaged in formalizing metatheory have access to tools capable of reasoning linearly, such as Linear LF? He suggests they do not and that it is best for researchers to work with the tools they already have. Second, even if extensions for linear languages become more widely available, the issue remains unresolved for those studying other substructural logics, such as affine, strict, or modal logic. Moreover, treating contexts explicitly for substructurality is simply not a viable option [**?**].

In general, the linearity predicate acts as a local well-formedness check that forms a pseudo-context whenever a new variable is introduced. It ensures that the new variable is used linearly, as expected, even though the actual context managed by HOAS remains structural. For example, in Crary's implementation in Twelf, the lambda rule is represented as:

$$\frac{\Gamma; (\Delta, x{:}A) \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : A \multimap B}$$

```
of/llam
: of (llam ([x] M x)) (lolli A B)
<- ({x:term} of x A -> of (M x) B)
<- linear ([x] M x).

linear/llam
: linear ([y] llam ([x] M y x))
<- ({x:term} linear ([y] M y x)).
```

Without delving into Twelf's syntax, this states that a lambda function is of type $A \multimap B$ if a given variable of type A, *used linearly*, ensures that its body is of type B. Similarly, a linearity rule based on this typing judgment asserts that if a variable is used linearly within the body of a function, then it is linear within that function. Globally, this implies that every variable in the program is used linearly.

As with the work of Sano et al. [**?**], we must introduce a version of the language to encode that operates structurally, using this pseudo-linear context. We achieve this below, creating *Structural Proto-Quipper* (SPQ). We begin with a discussion of circuits, as they operate as a linear lambda calculus, closely resembling Crary's system.

4 Treatment of Circuits

The way that Proto-Quipper handles circuits is challenging for mechanization. In the paper, they treat circuits abstractly as coming from a countable set C of circuit constants, equipped with functions `In` and `Out` which describe their inputs and outputs, respectively. To do so, they assume that every possible circuit is represented as some $C \in \mathcal{C}$ and, at the term level, are modeled as (t, C, a) with t and a as terms themselves providing structure to these inputs and outputs. At the type level, Proto-Quipper argues that circuit (t, C, a) is of type $\mathrm{Circ}(T, U)$ if t (resp. a) takes the shape of T (resp. U), where T and U are quantum data types. The typing rule is as follows:

$$\frac{Q_1 \vdash t : T \quad !\Delta; Q_2 \vdash a : U \quad \mathrm{In}(C) = Q_1 \quad \mathrm{Out}(C) = Q_2}{!\Delta; \emptyset \vdash (t, C, a) :!^n \mathrm{Circ}(T, U)} \ (circ)$$

However, in SPQ's structural setting where we let Beluga's HOAS handle contexts, it is impossible for us to say, without loss of generality, that some empty context in addition to the (uninterpreted) outputs of a circuit will prove that a is of type U.

Further, because Proto-Quipper is a circuit building language, it also describes operations on circuits. It is the case, for example, that Proto-Quipper is able to build up functions that are "boxed" into circuits. In the operational semantics, this is done through creating new circuits on free quantum wires, as below, with the `box` function taking a circuit generating function to a circuit by finding fresh variable names (Spec) and creating an identity circuit on those wires (new).

$$\frac{\mathrm{Spec}_{\mathsf{FQ}(v)}(T) = t \quad \mathrm{new}(\mathsf{FQ}(t)) = D}{[C, box^T(v)] \longrightarrow [C, (t, D, vt)]} \ (box)$$

Similarly, to continue building circuits, Proto-Quipper allows us to append two circuits circuits together over bindings by first turning one into a function, as in the rule below, with the `unbox` function taking a circuit to a circuit generating function applied to a new collection of input wires. This new circuit D and its binding are then tacked onto the circuit we are currently building through the Append function.

$$\frac{bind(v, u) = \mathfrak{b} \quad \mathrm{Append}(C, D, \mathfrak{b}) = (C', \mathfrak{b}') \quad \mathsf{FQ}(u') \subseteq \mathrm{dom}(\mathfrak{b}')}{[C, (unbox(u, D, u'))v] \longrightarrow [C', \mathfrak{b}'(u')]} \ (unbox)$$

In both cases, Proto-Quipper develops many tools to ensure freshness of quantum variables and one-to-one mappings of input wires to output wires. These tools, however, are set operations which require reasoning about explicit lists of variables; this is perpendicular to the goals of our research and makes proofs of metatheoretic properties deeply challenging.

To resolve this, Structural Proto-Quipper proposes a two-level system between circuits and their creation, i.e. between the term level and the circuit level of our language, with circuits expressed as functions in a linear lambda calculus on variables representing quantum wires. We treat box^T and $unbox$ as translations from circuits to terms and vice versa.

Thus, the terms w, u in our circuit linear lambda calculus are

$$w, u ::= q \mid * \mid \langle w, u \rangle \mid \lambda q.\ w \mid \mathrm{app}\ w\ u \mid \mathrm{let}\ * = w\ \mathrm{in}\ u \mid$$
$$\mathrm{let}\ \langle q, v \rangle = w\ \mathrm{in}\ u \mid \mathrm{rev}\ w$$

where we type collections of wires as base types

$$A_B, B_B ::= \mathrm{qubit} \mid 1 \mid B_B \otimes C_B$$

and circuits as reversible functions from base types to base types.

$$C_C ::= A_B \multimap B_B$$

.

The type system for our circuit level is nearly identical to Crary's higher order representation of the linear lambda calculus[**?**]. We say that $\Gamma \models w : A_B$ if w is of base type B_B under the structural context Γ. Further, we say that $\Gamma \models \lambda q.\ w :_C A_B \multimap B_B$ if a circuit is

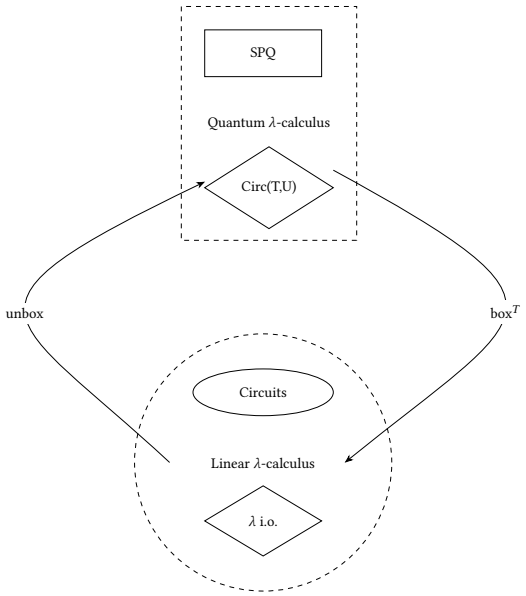

Figure 1: Two Levels of SPQ

of input base type A_B and output base type B_B under Γ. For linearity, there are no reusable resources and thus typing is no longer needed, so `clamlin` $(q;\ w)$ says that variable q is used linearly within term w.

First, we have a linearity rule for variables, reading that a variable q is linear in q

$$\frac{}{\texttt{clamlin}\ (q;\ q)}\ \{\texttt{clamlin/var}\}$$

Axiomatically, we type units as follows, with no linearity predicate as no variables can be used within.

$$\frac{}{\Gamma \models * :_B 1}\{*_i\}$$

For tuples of quantum variables, however, we must ensure linearity in both branches as follows.

$$\frac{\Gamma \models w :_B A_B \quad \Gamma \models u :_B B_B}{\Gamma \models \langle w, u\rangle :_B A_B \otimes B_B}\ \{\otimes_i\}$$

$$\frac{\texttt{clamlin}\ (q;\ w)}{\texttt{clamlin}\ (q;\ \langle w, u\rangle)}\ \{\texttt{clamlin/}\otimes_1\}$$

$$\frac{\texttt{clamlin}\ (q;\ u)}{\texttt{clamlin}\ (q;\ \langle w, u\rangle)}\ \{\texttt{clamlin/}\otimes_2\}$$

The lambda typing rule requires the argument to be used linearly in its body. The linearity rule for lambda functions says variable q is linear in a function if it is linear in its body. Here, the lambda functions represent circuits in Structural Proto–Quipper. It is the sole constructor for circuits.

$$\frac{\Gamma, q :_B A_B \models w :_B B_B \quad \texttt{clamlin}\ (q;\ w)}{\Gamma \models \lambda q.\ w :_C A_B \multimap B_B}\ \{\lambda\}$$

$$\frac{\texttt{clamlin}\ (x;\ u)}{\texttt{clamlin}\ (x;\ \lambda q.\ u)}\ \{\texttt{clamlin/}\lambda\}$$

The typing and linearity rules for applying circuits onto inputs is similar to that for the pair, in that we must check linearity of both the function and its input.

$$\frac{\Gamma \models w :_C A_B \multimap B_B \quad \Gamma \models u :_B B_B}{\Gamma \models \texttt{app}\ \ w\ u :_B B_B}\ \{app\}$$

$$\frac{\texttt{clamlin}\ (q;\ w)}{\texttt{clamlin}\ (q;\ \texttt{app}\ \ w\ u)}\ \{\texttt{clamlin/app1}\}$$

$$\frac{\texttt{clamlin}\ (q;\ u)}{\texttt{clamlin}\ (q;\ \texttt{app}\ \ w\ u)}\ \{\texttt{clamlin/app2}\}$$

.

The same goes for unit elimination.

$$\frac{\Gamma \models w :_B 1 \quad \Gamma \models u :_B A_B}{\Gamma \models \texttt{let}\ * = w\ \texttt{in}\ u :_B A_B}\ \{*_e\}$$

$$\frac{\texttt{clamlin}\ (q;\ w)}{\texttt{clamlin}\ (q;\ \texttt{let}\ * = w\ \texttt{in}\ u)}\ \{\texttt{clamlin/letunit1}\}$$

$$\frac{\texttt{clamlin}\ (q;\ u)}{\texttt{clamlin}\ (q;\ \texttt{let}\ * = w\ \texttt{in}\ u)}\ \{\texttt{clamlin/letunit2}\}$$

And for the tensor elimination, as with the lambda typing rule, we must ensure linearity of the variables we use for substitution. Here,

$$\frac{\begin{array}{l}\Gamma \models w :_B A_B \otimes B_B \\ \Gamma, q :_B A_B, u :_B B_B \models u :_B D_B \\ \texttt{clamlin}\ (q;\ u) \\ \texttt{clamlin}\ (v;\ u)\end{array}}{\Gamma \models \text{let}\ \langle q, v\rangle = w\ \text{in}\ u :_B D_B}$$

$$\frac{\texttt{clamlin}\ (q;\ w)}{\texttt{clamlin}\ (q;\ \texttt{let}\ \langle q, v\rangle = w\ \texttt{in}\ u)}\ \{\texttt{clamlin/letpair1}\}$$

$$\frac{\texttt{clamlin}\ (q;\ u)}{\texttt{clamlin}\ (q;\ \texttt{let}\ \langle q, v\rangle = w\ \texttt{in}\ u)}\ \{\texttt{clamlin/letpair2}\}$$

Lastly, reversing a circuit is equivalent to swapping the shape of its inputs and outputs, as below. This is the main novelty versus Crary's paper. Linearity of a variable within a circuit implies linearity of a c variable within its reverse.

$$\frac{\Gamma \models w :_C A_B \multimap B_B}{\Gamma \models \texttt{rev}\ \ w :_C B_B \multimap A_B}\ \{rev\}$$

$$\frac{\texttt{clamlin}\ (q;\ w)}{\texttt{clamlin}\ (q;\ \texttt{rev}\ \ w)}\ \{\texttt{clamlin/rev}\}$$

Below is our encoding of the linear lambda calculus for circuits in the LF layer of Beluga. It gives a good idea of how linearity and typing are intertwined.

```
    clam : type.
    clam/unit : clam.
    clam/pair : clam -> clam -> clam.
    clam/lam : (clam -> clam) -> clam.
    clam/app : clam -> clam -> clam.
    clam/let_unit : clam -> clam -> clam.
    clam/let_pair : (clam -> clam -> clam
        ) -> clam -> clam.
    clam/rev : clam -> clam.

  ctp_base : type.
```

```
  ctp_base/qubit : ctp_base.
  ctp_base/one : ctp_base.
  ctp_base/times : ctp_base -> ctp_base
       -> ctp_base.

ctp_circ : type.
  ctp_circ/lolli : ctp_base -> ctp_base
       -> ctp_circ.

ofctp_base : clam -> ctp_base -> type.
ofctp_circ : clam -> ctp_circ -> type.
clamlin : (clam -> clam) -> type.

clamlin/var : clamlin (\x. x).

ofctp_base/unit : ofctp_base clam/unit
    ctp_base/one.

ofctp_base/pair : ofctp_base a A ->
    ofctp_base b B
         -> ofctp_base (clam/pair a b)
              (ctp_base/times A B).
  clamlin/pair1 : clamlin (\x. M x) ->
      clamlin (\x. clam/pair (M x) N).
  clamlin/pair2 : clamlin (\x. N x) ->
      clamlin (\x. clam/pair M (N x)).

ofctp_circ/lam : clamlin (\x. M x)
         -> ({x : clam} ofctp_base x A
             -> ofctp_base (M x) B)
         -> ofctp_circ (clam/lam (\x. M
              x)) (ctp_circ/lolli A B).
  clamlin/lam : ({x : clam} clamlin (\y
      . M y x))
       -> clamlin (\y. clam/lam (\x. M y
            x)).

ofctp_circ/rev : ofctp_circ M (ctp_circ
    /lolli A B)
         -> ofctp_circ (clam/rev M) (
             ctp_circ/times B A).

ofctp_base/app : ofctp_base N A
         -> ofctp_circ M (ctp_circ/
             lolli A B)
         -> ofctp_base (clam/app M N) B
              .
  clamlin/app1 : clamlin (\x. N x)
       -> clamlin (\x. clam/app M (N x)
            ).
  clamlin/app2 : clamlin (\x. M x)
       -> clamlin (\x. clam/app (M x) N
            ).

ofctp_base/let_unit : ofctp_base N C
              -> ofctp_base M ctp_base/
                  one
              -> ofctp_base (clam/
                  let_unit M N) C.
  clamlin/let_unit1 : clamlin (\x. M x)
            -> clamlin (\x. clam/
                let_unit (M x) N).
  clamlin/let_unit2 : clamlin (\x. N x)
            -> clamlin (\x. clam/
                let_unit M (N x)).

ofctp_base/let_pair : ({x : clam}
    clamlin (\y. N x y))
              -> ({y : clam} clamlin (\
                  x. N x y))
              -> ({x : clam} ofctp_base
                   x A
                 -> {y : clam}
                      ofctp_base y B
                 -> ofctp_base (N x y)
                      C)
              -> ofctp_base M (ctp_base
                  /times A B)
              -> ofctp_base (clam/
                  let_pair M (\x. \y. N
                   x y)) C.
  clamlin/let_pair1 : clamlin (\z. M z)
            -> clamlin (\z. clam/
                let_pair (M z) (\x. \y.
                 N x y)).
  clamlin/let_pair2 : ({x : clam} {y :
      clam} clamlin (\z. N z x y))
            -> clamlin (\z. clam/
                let_pair M (\x. \y. N z
                 x y)).
```

5 Treatment of Terms, Typing, and Linearity

First, we model the terms and types of Structural Proto-Quipper nearly identically to Proto-Quipper, save for the difference in circuits expressed above. The types are as follows, and are identical to those found in PQ:

$$A, B \quad ::= \quad \textbf{qubit} \mid 1 \mid \textbf{bool} \mid A \otimes B \mid A \multimap B \mid !A \mid \mathrm{Circ}(T, U).$$

However, for terms, we express circuits differently. Instead of as in PQ where they are expressed as triples (t, C, a), as explained previously, we choose to represent them as tuples (C, u) where C is a circuit lambda from the circuit level and u is a term-level function modelling its outputs. This will be evident in its typing rule.

$$\begin{aligned} a, b, c ::= \quad & x \mid q \mid (C, u) \mid \mathrm{True} \mid \mathrm{False} \mid \langle a, b \rangle \mid * \mid ab \mid \lambda x.a \mid \\ & \mathit{rev} \mid \mathit{unbox} \mid \mathit{box}^T \mid \text{ if } a \text{ then } b \text{ else } c \mid \text{ let } * = a \text{ in } b \mid \\ & \text{let } \langle x, y \rangle = a \text{ in } b. \end{aligned}$$

We compare the typing judgments in Proto-Quipper (indicated in round parentheses), assuming a linear logic, to those in Structural Proto-Quipper [indicated in square parentheses], assuming a structural logic supported by our linear predicate(s). First, we note that a reusable resource (indicated with an exclamation mark in the language) is always a sufficient condition for the linearity predicate, as in the following rule

$$\frac{}{\mathtt{lin}\ (x :!B;\ \Gamma, x :!B \vDash a : A)}\ [\mathtt{lin!}]$$

Now, the axiom rule for classical resources below states that given a variable of type A where A is a subtype of type B, the same

variable is of type B as well. Further, we have that a variable is always used linearly in such a judgment.

$$\frac{A <: B}{!\Delta, x : A; \emptyset \vdash x : B}\ (ax_c) \qquad \frac{A <: B}{\Gamma, x : A \vDash x : B}\ [ax_c]$$

$$\frac{}{\texttt{lin}\ \ (x : A;\ \Gamma, x : A \vDash_{[ax_c]} b : B)}\ [\texttt{lin/axc}]$$

The axiom rule for qubits is much simpler, and we note that said quantum variable is used linearly in the typing judgment.

$$\frac{}{!\Delta; \{q\} \vdash q : \textbf{qubit}}\ (ax_q) \qquad \frac{}{\Gamma; \{q\} \vDash q : \textbf{qubit}}\ [ax_q]$$

$$\frac{}{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash_{[ax_q]} q : \textbf{qubit})}\ [\texttt{lin/q/axq}]$$

The three constant functions in Proto-Quipper (box^{T}, unbox, rev) have defined types as follows, with typing through subtyping.

$$A_{box}(T, U) = !(T \multimap U) \multimap !\,\text{Circ}(T, U)$$

$$A_{unbox}(T, U) = \text{Circ}(T, U) \multimap !(T \multimap U)$$

$$A_{rev}(T, U) = \text{Circ}(T, U) \multimap !\,\text{Circ}(U, T)$$

$$\frac{!A_c(T, U) <: B}{!\Delta; \emptyset \vdash c : B}\ (cst)$$

The rule is the same in SQP, with no associated linearity predicate.

$$\frac{!A_C(T, U <: B)}{\Gamma \vDash c : B}\ [cst]$$

The unit is typed with no linear resources.

$$\frac{}{!\Delta; \emptyset \vdash * :!^n 1}\ (*_i) \qquad \frac{}{\Gamma \vDash * :!^n 1}\ [*_i]$$

We have two lambda rules, deciding whether or not the abstraction can be used non-linearly. Since do not want to allow any pre-existing linear resources to be used under the bang, there is no linearity rule for the second.

$$\frac{\Gamma, x : A; Q \vdash b : B}{\Gamma; Q \vdash \lambda x.b : A \multimap B}\ (\lambda_1)$$

$$\frac{\Gamma, x : A \vDash b : B \quad \texttt{lin}\ \ (x : A; \Gamma, x : A \vDash b : B)}{\Gamma \vDash \lambda x.b : A \multimap B}\ [\lambda_1]$$

$$\frac{\texttt{lin}\ \ (y : C; \Gamma, y : C, x : A \vDash b : B)}{\texttt{lin}\ \ (y : C; \Gamma, y : C \vDash_{[\lambda_1]} \lambda x.b : A \multimap B)}\ [\texttt{lin/}\lambda_1]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma, x : A; \{q\} \vDash b : B)}{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash_{[\lambda_1]} \lambda x.b : A \multimap B)}\ [\texttt{lin/q/}\lambda_1]$$

$$\frac{!\Delta, x : A; \emptyset \vdash b : B}{!\Delta; \emptyset \vdash \lambda x.b :!^{n+1}(A \multimap B)}\ (\lambda_2)$$

$$\frac{\Gamma, x : A \vDash b : B \quad \texttt{lin}\ \ (x : A; \Gamma, x : A \vDash b : B)}{\Gamma \vDash \lambda x.b :!^{n+1}(A \multimap B)}\ [\lambda_2]$$

In the following application rule for SPQ, we can pass on a linearity predicate from either of the two assumptions.

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash c : A \multimap B \quad \Gamma_2, !\Delta; Q_2 \vdash a : A}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash ca : B}\ (app)$$

$$\frac{\Gamma \vDash c : A \multimap B \quad \Gamma \vDash a : A}{\Gamma \vDash ca : B}\ [app]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash c : A \multimap B)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[app]} ca : B)}\ [\texttt{lin/app1}]$$

$$\frac{\texttt{lin/q}\ \ (x : C; \Gamma, x : C \vDash a : A)}{\texttt{lin/q}\ \ (x : C; \Gamma, x : C \vDash_{[app]} ca : B)}\ [\texttt{lin/q/app2}]$$

$$\frac{\texttt{lin}\ \ (q; \Gamma, x : C \vDash c : A \multimap B)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[app]} ca : B)}\ [\texttt{lin/app1}]$$

$$\frac{\texttt{lin/q}\ \ (x : C; \Gamma, x : C \vDash a : A)}{\texttt{lin/q}\ \ (x : C; \Gamma, x : C \vDash_{[app]} ca : B)}\ [\texttt{lin/q/app2}]$$

This is also true for the tensor introduction rule.

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash a :!^n A \quad \Gamma_2, !\Delta; Q_2 \vdash b :!^n B}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash \langle a, b\rangle :!^n (A \otimes B)}\ (\otimes_i)$$

$$\frac{\Gamma \vDash a :!^n A \quad \Gamma \vDash b :!^n B}{\Gamma \vDash \langle a, b\rangle :!^n (A \otimes B)}\ [\otimes_i]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash a : A)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[\otimes_i]} \langle a, b\rangle :!^n (A \otimes B)}\ [\texttt{lin/}\otimes 1]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash b : B)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[\otimes_i]} \langle a, b\rangle :!^n (A \otimes B)}\ [\texttt{lin/}\otimes 2]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma, \{q\} : C \vDash a : A)}{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash_{[\otimes_i]} \langle a, b\rangle :!^n (A \otimes B)}\ [\texttt{lin/q/}\otimes 1]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash b : B)}{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash_{[\otimes_i]} \langle a, b\rangle :!^n (A \otimes B)}\ [\texttt{lin/q/}\otimes 2]$$

But the tensor elimination rule is more similar to our lambda rules, as we must ensure linearity of **both** bound variables.

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash b :!^n (B_1 \otimes B_2) \quad \Gamma_2, !\Delta, x :!^n B_1, y :!^n B_2; Q_2 \vdash a : A}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A}\ (\otimes_e)$$

$$\frac{\begin{array}{l}\Gamma \vDash b :!^n (B_1 \otimes B_2)\\ \Gamma, x :!^n B_1, y :!^n B_2 \vDash a : A\\ \texttt{lin}\ \ (x :!^n B_1; \Gamma, x :!^n B_1, y :!^n B_2 \vDash a : A)\\ \texttt{lin}\ \ (y :!^n B_2; \Gamma, x :!^n B_1, y :!^n B_2 \vDash a : A)\end{array}}{\Gamma \vDash \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A}\ [\otimes_e]$$

$$\frac{\texttt{lin}\ \ (z : C; \Gamma, z : C \vDash b :!^n (B_1 \otimes B_2))}{\texttt{lin}\ \ (z : C; \Gamma, z : C \vDash_{[\otimes_e]} \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A)}\ [\texttt{lin/}\otimes 1]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash b :!^n (B_1 \otimes B_2))}{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash_{[\otimes_e]} \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A)}\ [\texttt{lin/q/}\otimes 1]$$

$$\frac{\texttt{lin}\ \ (z : C; \Gamma, z : C, x :!^n B_1, y :!^n B_2 \vDash a : A)}{\texttt{lin/}(z : C; \Gamma, z : C \vDash_{[\otimes_e]} \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A)}\ [\texttt{lin}\otimes 2]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma, x :!^n B_1, y :!^n B_2; \{q\} \vDash a : A)}{\texttt{lin/q}\ \ (q; \Gamma, \{q\} \vDash_{[\otimes_e]} \text{let}\ \langle x, y\rangle = b\ \text{in}\ a : A)}\ [\texttt{lin/q/}\otimes 2]$$

For unit elimination, we have (unsurprisingly)

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash b :!^n 1 \quad \Gamma_2, !\Delta; Q_2 \vdash a : A}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash \text{let} * = b \text{ in } a : A}\ (*_e)$$

$$\frac{\Gamma \vDash b :!^n 1 \quad \Gamma \vDash a : A}{\Gamma \vDash \text{let} * = b \text{ in } a : A}\ [*_e]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash b : B)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[*_e]} \text{let} * = b \text{ in } a : A)}\ [\texttt{lin/*1}]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash b : B)}{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash_{[*_e]} \text{let} * = b \text{ in } a : A)}\ [\texttt{lin/q/*1}]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash a : A)}{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash_{[*_e]} \text{let} * = b \text{ in } a : A)}\ [\texttt{lin/*2}]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash a : A)}{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash_{[*_e]} \text{let} * = b \text{ in } a : A)}\ [\texttt{lin/q/*2}]$$

The booleans are typed as expected, relying on no linear resources so there are no associated linearity predicates.

$$\overline{!\Delta; \emptyset \vdash \text{True} :!^n \textbf{bool}}\ (\top) \quad \overline{!\Delta; \emptyset \vdash \text{False} :!^n \textbf{bool}}\ (\bot)$$

$$\overline{\Gamma \vDash \text{True} :!^n \textbf{bool}}\ [\top] \quad \overline{\Gamma \vDash \text{False} :!^n \textbf{bool}}\ [\bot]$$

Further, although the $[if]$ rule has three variables, we check linearity in the condition b or in both the consequences a_1, a_2, as we do not want to lose linearity (say) if we checked only that a variable x was used linearly in the truth branch but the false branch executed.

$$\frac{\Gamma_1, !\Delta; Q_1 \vdash b : \textbf{bool} \quad \Gamma_2, !\Delta; Q_2 \vdash a_1 : A \quad \Gamma_2, !\Delta; Q_2 \vdash a_2 : A}{\Gamma_1, \Gamma_2, !\Delta; Q_1, Q_2 \vdash \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A}\ (if)$$

$$\frac{\Gamma \vDash b : \textbf{bool} \quad \Gamma \vDash a_1 : A \quad \Gamma \vDash a_2 : A}{\Gamma \vDash \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A}\ [if]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash b : \textbf{bool})}{\texttt{lin}\ \ (x : C, \Gamma, x : C \vDash_{[if]} \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A)}\ [\texttt{lin/if1}]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash b : \textbf{bool})}{\texttt{lin/q}\ \ (q, \Gamma; \{q\} \vDash_{[if]} \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A)}\ [\texttt{lin/q/if1}]$$

$$\frac{\texttt{lin}\ \ (x : C; \Gamma, x : C \vDash a_1 : A) \quad \texttt{lin}\ \ (x : C; \Gamma, x : C \vDash a_2 : A)}{\texttt{lin}\ \ (x : C, \Gamma, x : C \vDash_{[if]} \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A)}\ [\texttt{lin/if2}]$$

$$\frac{\texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash a_1 : A) \quad \texttt{lin/q}\ \ (q; \Gamma; \{q\} \vDash a_2 : A)}{\texttt{lin/q}\ \ (q, \Gamma; \{q\} \vDash_{[if]} \textbf{if } b \text{ then } a_1 \text{ else } a_2 : A)}\ [\texttt{lin/q/if1}]$$

Lastly, we model circuits differently than in Proto-Quipper. Below is the rule in PQ for typing circuits. Here, t of type T and a of type U can be thought of as the inputs and the outputs (up to reduction), respectively, of a circuit C from a countable set $\mathcal{C}$ of circuit constants. These circuits are treated abstractly[?].

$$\frac{Q_1 \vdash t : T \quad !\Delta; Q_2 \vdash a : U \quad \text{In}(C) = Q_1 \quad \text{Out}(C) = Q_2}{!\Delta; \emptyset \vdash (t, C, a) :!^n \text{Circ}(T, U)}\ (circ)$$

Given that in SQP, we are able to treat circuits as actual functions, our typing rule argues that, at the term level, a circuit tuple (C, u) is of type $\text{Circ}(T, U)$ if C is of type $T' \multimap U'$ and u is of type $U \multimap U$ where T' as a base type in the circuit level is equivalent to T as a type in the term level (and ditto for U, U').

$$\frac{\Gamma \models C :_{\mathcal{C}} T' \multimap U' \quad \Gamma \models u : U \multimap U \quad T \equiv T' \quad U \equiv U'}{\Gamma \vDash (C, u) : \text{Circ}(T, U)}$$

Here is the code for typing and linearity in Beluga.

```
    oft : tm -> tp -> type.
oftq : qtm -> qtp -> type.
lin : ({x : tm} oft x A -> oft (b x) B)
    -> type.
lin/q : ({x : qv} oft (b x) B) -> type.

lin/bang : {D : ({y : tm} {ty : oft y (!
    B)} oft (a y) A)} -> lin D.

oft/axc : A subtype B -> oft z A -> oft z
     B.

lin/axc : lin (\x.\tx.oft/axc _ tx).

oft/axq : oft (qvar q) qubit.

oftq/axq : {q : qv} oftq (qtm/qvar q) qtm
    /qubit.

lin/q/qvar : lin/q (\q. (oft/axq : oft (
    qvar q) qubit)).

const_tp : const_name -> tp -> type.

const_tp/box : tp2qtp T' T -> tp2qtp U' U
     -> const_tp (boxt T) ((! (T' lolli U
    )) lolli (! (circ T U))).

const_tp/unbox : tp2btp T' T -> tp2btp U'
     U -> const_tp unbox ((circ T U)
    lolli (! (T' lolli U'))).

const_tp/rev : tp2btp T' T -> tp2btp U' U
     -> const_tp rev ((circ T U) lolli (!
     (circ U T))).

oft/cst : const_tp c A -> (! A) subtype B
     -> oft (const c) B.

oft/unit : strip_bangs A one -> oft unit
    A.

oftq/unit : oftq qtm/unit qtp/unit.

oft/lam1 : {D : ({x : tm} oft x A -> oft
    (b x) B)} -> lin D -> oft (lam b) (A
    lolli B).

```

```
lin/lam1 : ({x : tm} {tx : oft x A} lin
    (\y.\ty.D x tx y ty)) -> lin (\y.\ty.
    oft/lam1 (\x.\tx.D x tx y ty) (L y ty
    )).

lin/q/lam1 : ({x : tm} {tx : oft x A} lin
    /q (\q. D x tx q)) -> lin/q (\q. oft/
    lam1 (\x.\tx.D x tx q) (L q)).

oft/lam2 : {D : ({x : tm} oft x A -> oft
    (b x) B)} -> lin D -> bang_of C (A
    lolli B) -> oft (lam b) C.

oft/app : oft c (A lolli B) -> oft a A ->
     oft (app c a) B.

lin/app1 : {D : {x : tm} oft x _ -> oft (
    c x) (A lolli B)} -> lin D -> lin (\x
    .\tx.oft/app (D x tx) _).

lin/app2 : {D : {x : tm} oft x _ -> oft (
    a x) A} -> lin D -> lin (\x.\tx.oft/
    app _ (D x tx)).

lin/q/app1 : {D : {q : qv} oft (c q) (A
    lolli B)} -> lin/q D -> lin/q (\q.oft
    /app (D q) _).

lin/q/app2 : {D : {q : qv} oft (c q) A}
    -> lin/q D -> lin/q (\q.oft/app _ (D
    q)).

oft/tensor_intro : oft a A -> oft b B ->
    equibang AB A (A' tensor B') A' ->
    equibang AB B (A' tensor B') B' ->
    oft (pair a b) AB.

oftq/tensor_intro : oftq a qA -> oftq b
    qB -> tp2qtp A qA -> tp2qtp B qB ->
    tp2qtp (A tensor B) qAB -> oftq (qtm/
    pair a b) qAB.

lin/tensor_intro1 : {D : {x : tm} oft x _
     -> oft (a x) A} -> lin D -> lin (\x
    .\tx.oft/tensor_intro (D x tx) _ _ _)
    .

lin/tensor_intro2 : {D : {x : tm} oft x _
     -> oft (b x) B} -> lin D -> lin (\x
    .\tx.oft/tensor_intro _ (D x tx) _ _)
    .

lin/q/tensor_intro1 : {D : {q : qv} oft (
    a q) A} -> lin/q D -> lin/q (\q.oft/
    tensor_intro (D q) _ _ _).

lin/q/tensor_intro2 : {D : {q : qv} oft (
    b q) B} -> lin/q D -> lin/q (\q.oft/
    tensor_intro _ (D q) _ _).
```

```
oft/tensor_elim : oft b B1B2 -> {D : {x :
     tm} oft x B1 -> {y : tm} oft y B2 ->
     oft (a x y) A} -> ({x : tm} {tx :
    oft x B1} lin (D x tx)) -> ({y : tm}
    {ty : oft y B2} lin (\x.\tx.D x tx y
    ty)) -> equibang B1B2 B1 (B1' tensor
    B2') B1' -> equibang B1B2 B2 (B1'
    tensor B2') B2' -> oft (let_pair b a)
     A.

lin/tensor_elim2 : {D : {z : tm} oft z C
    -> oft (b z) B1B2} -> lin D -> lin (\
    z.\tz.oft/tensor_elim (D z tz) _ _ _
    _ _).

lin/tensor_elim1 : ({x : tm} {tx : oft x
    X} {y : tm} {ty : oft y Y} lin (\z.\
    tz.D z tz x tx y ty)) -> lin (\z.\tz.
    oft/tensor_elim _ (D z tz) (Ly z tz)
    (Lx z tz) _ _).

lin/q/tensor_elim1 : ({x : tm} {tx : oft
    x X} {y : tm} {ty : oft y Y} lin/q (\
    q.D q x tx y ty)) -> lin/q (\q.oft/
    tensor_elim _ (D q) (Ly q) (Lx q) _ _
    ).

lin/q/tensor_elim2 : {D : {q : qv} oft (b
     q) B1B2} -> lin/q D -> lin/q (\q.oft
    /tensor_elim (D q) _ _ _ _ _).

oft/let_unit : strip_bangs B one -> oft b
     B -> oft a A -> oft (let_unit b a) A
    .

lin/let_unit1 : {D : {x : tm} oft x C ->
    oft (b x) B} -> lin D -> lin (\x.\tx.
    oft/let_unit _ (D x tx) _).

lin/let_unit2 : {D : {x : tm} oft x C ->
    oft (a x) A} -> lin D -> lin (\x.\tx.
    oft/let_unit _ _ (D x tx)).

lin/q/let_unit1 : {D : {q : qv} oft (b q)
     B} -> lin/q D -> lin/q (\q.oft/
    let_unit _ (D q) _).

lin/q/let_unit2 : {D : {q : qv} oft (a q)
     A} -> lin/q D -> lin/q (\q.oft/
    let_unit _ _ (D q)).

oft/circ : ofctp_circ C (ctp_circ/lolli T
     U) -> oft u (U lolli U) -> tp2btp T
    T' -> tp2btp U U' -> oft (qcirc C u)
    (circ T U).
```

6 Operational Semantics

To reiterate, the reason that we modify Proto-Quipper's circuits is to aid in the operational semantics, where Ross devotes much time and effort to rigorously define set functions on wires – time and

effort increased exponentially when it comes to mechanization. We will demonstrate how our changes avoid such labour.

First, however, we note that SPQ's non-circuit related operational semantics are essentially equivalent. In PQ, they define a closure $[C, a]$ on a circuit C from their circuit constants and a a term. SPQ does so with C a circuit lambda from the circuit level and a, too, a term. Below are the rules which are identical in PQ and SPQ.

$$\frac{[C, a] \to [C', a']}{[C, ab] \to [C', a'b]}(fun)$$

$$\frac{[C, b] \to [C', b']}{[C, vb] \to [C', vb']}(arg)$$

$$\frac{[C, b] \to [C', b']}{[C, \langle a, b\rangle] \to [C', \langle a, b'\rangle]}(right)$$

$$\frac{[C, a] \to [C', a']}{[C, \langle a, v\rangle] \to [C', \langle a', v\rangle]}(left)$$

$$\frac{[C, a] \to [C', a']}{[C, \text{let} * = a \text{ in } b] \to [C', \text{let} * = a' \text{ in } b]}\ (let*)$$

$$\frac{[C, a] \to [C', a']}{[C, \text{let } \langle x, y\rangle = a \text{ in } b] \to [C', \text{let } \langle x, y\rangle = a' \text{ in } b]}\ (let)$$

$$\frac{[C, a] \to [C', a']}{[C, \text{if } a \text{ then } b \text{ else } c] \to [C', \text{if } a' \text{ then } b \text{ else } c]}(cond)$$

$$\overline{[C, (\lambda x.a)v] \to [C, a[v/x]]}(\beta)$$

$$\overline{[C, \text{let} * = * \text{ in } a] \to [C, a]}(unit)$$

$$\overline{[C, \text{let } \langle x, y\rangle = \langle v, w\rangle \text{ in } a] \to [C, a[v/x, w/y]]}\ (pair)$$

$$\overline{[C, \text{if False then } a \text{ else } b] \to [C, b]}(ifF)$$

$$\overline{[C, \text{if True then } a \text{ else } b] \to [C, a]}(ifT)$$

The circuit rule in PQ is as expected.

$$\frac{[D, a] \to [D', a']}{[C, (t, D, a)] \to [C, (t, D', a')]}\ (circ)$$

SQP does the same, although on circuit tuples (D, a) instead.

$$\frac{[D, a] \to [D', a']}{[C, (D, a)] \to [C, (D', a')]}\ [circ]$$

SPQ becomes more interesting when we consider the $[box]$, $[unbox]$, and $[rev]$ rules. We return where we left off in our discussion of the (box) rule:

$$\frac{\text{Spec}_{\text{FQ}(v)}(T) = t \quad \text{new}(\text{FQ}(t)) = D}{[C, box^T(v)] \to [C, (t, D, vt)]}\ (box)$$

Here, $Spec_X(T)$ returns an X-specimen for T, which is a quantum data term t that is "fresh" with respect to the quantum variables appearing in X. Too, new creates a new identity circuit on those wires. SPQ accomplishes the same overarching goal through generating a new circuit explicitly, without necessitating the creation of $Spec_X(T)$ or new, as the HOAS handles fresh variables.

$$\frac{\Gamma \vdash v : T \multimap U \quad \Gamma \models D :_C T' \rightharpoonup U' \quad T \equiv T' \quad U \equiv U'}{[C, box^T(v)] \to [C, (D, u)]}$$

Similarly, the $(unbox)$ rule is difficult to encode, as has previously been explained. *bind*, Append, FQ, and dom all rely on explicit reasoning about lists.

$$\frac{bind(v, u) = \mathfrak{b} \quad \text{Append}(C, D, \mathfrak{b}) = (C', \mathfrak{b}') \quad \text{FQ}(u') \subseteq \text{dom}(\mathfrak{b}')}{[C, (unbox(u, D, u'))v] \to [C', \mathfrak{b}'(u')]}\ (unbox)$$

Instead, SPQ utilizes the idea that circuit appending is equivalent to function application of our circuit lambas, which is a quite elegant solution.

$$\overline{[C, unbox(D, u)] \to [\lambda x.\ app\ D\ (app\ C\ x), u]}\ [unbox]$$

Not all of this has yet been encoded in Beluga, but it is currently being worked on.

7 Remaining Work and Conclusion

The primary remaining task for this project is to formalize the metatheory of Structural Proto-Quipper (SPQ). Two key proofs need to be encoded in Beluga's computational layer using the Curry-Howard Isomorphism: progress and type preservation.

Progress asserts that a well-typed term is either a value or can take a step further in execution. Type preservation guarantees that if a term takes a step, it remains well-typed [**?**]. These properties are fundamental safety guarantees in Proto-Quipper and are proven for typed closures of the form:

$$\Gamma; Q \vdash [C, a] : A, (Q' \mid Q'')$$

Here, Γ and Q represent linear and quantum contexts, $[C, a]$ denotes a closure, $\Gamma; Q \vdash a : A$ is a valid typing judgment, Q' represents the inputs to C, and Q, Q'' are the outputs of C. At present, we lack a clear representation of these concepts in SPQ. Should no direct mapping exist, significant effort will be required to conceptualize how to formulate progress and type preservation proofs for SPQ. In their absence, the work remains incomplete.

Another critical avenue is proving equivalence between the mechanization of a language and the language itself. However, since SPQ operates at the circuit level, which differs substantially from Proto-Quipper, reasoning about such equivalence poses a challenge. It is uncertain what form this equivalence would take, or even if it is achievable.

Additionally, as this project serves as a proof-of-concept for Crary and Sano et al.'s linearity predicate technique, we are interested in applying it to other frameworks. Among a family of Proto-Quipper-adjacent languages, one promising candidate is Proto-Quipper-Dyn [**?**]. This language introduces dynamic lifting, allowing outputs of quantum circuits to influence further circuit generation. Mechanizing Proto-Quipper-Dyn with the linearity predicate technique could provide significant insights.

In conclusion, we have demonstrated the potential of using a linearity predicate to replicate linearity within a Higher-Order Abstract Syntax (HOAS) system. This work introduced a structural variant of a quantum programming language grounded in linear logic, optimized for mechanization. We presented the typing rules and operational semantics for SPQ and believe strongly in the

potential of this technique. Even on a small scale, it has the power to significantly simplify the verification of quantum algorithms for researchers. We differ from Mahmoud et al.[**?**] in that we do so exclusively within the HOAS.

As much of this research is still taking place, certain elements of this report may be amended in the next months. We hope this provides an overview of the project as it stands right now.

8 Acknowledgments

I extend my heartfelt gratitude to Brigitte Pientka, Ryan Kavanagh, and Chuta Sano for their invaluable mentorship and guidance throughout this research process. To Professor Kavanagh, thank you for the privilege of being your first student. I have no doubt that you will excel as an advisor to many more in the future. To Chuta, I deeply appreciate your unwavering assistance in helping me navigate the intricacies of Beluga. To Professor Pientka, it has been an absolute pleasure to have you supervise my honors research project and to work as a TA for your undergraduate class. The experiences of researching and teaching under your guidance will remain pivotal in my academic journey.

We would also like to acknowledge the support provided by McGill University, UQÀM and the Natural Sciences and Engineering Research Council of Canada (NSERC).