Structural Proto-Quipper

Mechanization of a Linear Quantum Programming Language in a Structural Setting

Max Gross[1] Brigitte Pientka [2] Ryan Kavanagh [3] Chuta Sano [2]

[1]Department of Mathematics and Statistics
McGill University

[2]School of Computer Science
McGill University

[3]Département d'informatique
UQÁM

ECLaPS, Dec. 7th 2024

Table of Contents

1. Enforcing Linearity...
2. Without Linearity
3. Structural Proto-Quipper

Linear Logic

1. Linear logic is a sub-structural logic without the contraction and weakening rules

Linear Logic

1. Linear logic is a sub-structural logic without the contraction and weakening rules
   - i.e. Propositions, resources, etc. must be used once and exactly once

Linear Logic

1. Linear logic is a sub-structural logic without the contraction and weakening rules
   i.e. Propositions, resources, etc. must be used once and exactly once
2. Similarly to the λ-calculus, we utilize the Curry-Howard isomorphism to understand proofs in linear logic as terms in a **linear λ-calculus**

Leap to Quantum

There exists a natural application to quantum information theory regarding linear logic as a quantum-circuit language

Leap to Quantum

There exists a natural application to quantum information theory regarding linear logic as a quantum-circuit language

1. No contraction $\iff$ "no cloning property"

Leap to Quantum

There exists a natural application to quantum information theory regarding linear logic as a quantum-circuit language

1. No contraction $\iff$ "no cloning property"
2. No weakening $\iff$ "no deletion property"

Leap to Quantum

There exists a natural application to quantum information theory regarding linear logic as a quantum-circuit language

1. No contraction $\iff$ "no cloning property"
2. No weakening $\iff$ "no deletion property"

Thus, we have a **quantum λ-calculus** (which is linear),the basis of many quantum programming languages.

Leap to Quantum

There exists a natural application to quantum information theory regarding linear logic as a quantum-circuit language

1. No contraction $\iff$ "no cloning property"
2. No weakening $\iff$ "no deletion property"

Thus, we have a **quantum λ-calculus** (which is linear),the basis of many quantum programming languages.
"quantum programming language captures the ideas of quantum computation in a linear type theory" (Staton, 2015)

Mechanization

Our goal is to mechanize linear quantum programming languages.

Mechanization

Our goal is to mechanize linear quantum programming languages.

1. Quantum programs are notoriously difficult to reason about, mechanization allows rigorous, machine-checkable proofs of correctness for quantum programs.

Mechanization

Our goal is to mechanize linear quantum programming languages.

1. Quantum programs are notoriously difficult to reason about, mechanization allows rigorous, machine-checkable proofs of correctness for quantum programs.
2. As quantum programming languages are developed, important for developers to simultaneously advance the language and formalize its metatheory.

Mechanization... Is Hard

Mechanization... Is Hard

- Our main challenge is managing variable bindings in linear contexts.

Mechanization... Is Hard

- Our main challenge is managing variable bindings in linear contexts.
- Previous mechanizations of linear type systems treat contexts explcitly.

Mechanization... Is Hard

- Our main challenge is managing variable bindings in linear contexts.
- Previous mechanizations of linear type systems treat contexts explcitly.
  - i.e treat contexts as lists of variables, operate on those lists to prove various lemmas.

Mechanization... Is Hard

- Our main challenge is managing variable bindings in linear contexts.
- Previous mechanizations of linear type systems treat contexts explcitly.
  - i.e treat contexts as lists of variables, operate on those lists to prove various lemmas.
- Inefficient and burdensome to prove metatheoretic results, like progress and type preservation.

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant
  - Our host language manages contexts for us!

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant
  - Our host language manages contexts for us!

However, HOAS is not a cure-all:

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant
  - Our host language manages contexts for us!

However, HOAS is not a cure-all:

- It manages contexts structurally, **NOT** linearly.

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant
  - Our host language manages contexts for us!

However, HOAS is not a cure-all:

- It manages contexts structurally, **NOT** linearly.

$\Longrightarrow$ We cannot encode a linear logic within the HOAS.

Higher Order Abstract Syntax

Why Higher Order Abstract Syntax is awesome:

- HOAS relieves the need for explicitly encoded contexts
- Variables are encoded as functions in our host language or proof assistant
  - Our host language manages contexts for us!

However, HOAS is not a cure-all:

- It manages contexts structurally, **NOT** linearly.

$\Longrightarrow$ We cannot encode a linear logic within the HOAS. But we can come close.

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.
- To that end, we design **Structural Proto-Quipper**

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.
- To that end, we design **Structural Proto-Quipper**
  1. Operates like Proto-Quipper, but with a structural context

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.
- To that end, we design **Structural Proto-Quipper**
  1. Operates like Proto-Quipper, but with a structural context
  2. Optimized treatment of circuits for mechanization

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.
- To that end, we design **Structural Proto-Quipper**
  1. Operates like Proto-Quipper, but with a structural context
  2. Optimized treatment of circuits for mechanization
- We encode it in the LF layer and mechanize proofs of its safety properties (TBD)

Our Contribution

- We mechanize a quantum programming language within the HOAS itself (no extensions)
- Specifically, we mechanize Proto-Quipper, a small circuit-building language based on the quantum λ-calculus within Beluga.
- To that end, we design **Structural Proto-Quipper**
  1. Operates like Proto-Quipper, but with a structural context
  2. Optimized treatment of circuits for mechanization
- We encode it in the LF layer and mechanize proofs of its safety properties (TBD)
- This is based on a techique from (Sano et al, 2023)

Proto-Quipper I

Qubit rule:

$$\frac{}{!\Delta; \{q\} \vdash q : \mathbf{qubit}} \ (ax_q)$$

Proto-Quipper I

Qubit rule:

$$\frac{}{!\Delta; \{q\} \vdash q : \mathbf{qubit}} \ (ax_q)$$

Issue

How do we ensure the (linear) context is empty?

Proto-Quipper II

Application rule:

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

Proto-Quipper II

Application rule:

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

Issue

How do we split the (linear) context into Γ_1, Γ_2, Q_1, Q_2?

Proto-Quipper III

λ_1 rule

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

Proto-Quipper III

λ_1 rule

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

Issue

How can we select out variable x from the context?

Linear Predicates

We use local **linearity predicates** to ensure that all well-typed programs use their internally bound classical and quantum variables linearly.

Linear Predicates

We use local **linearity predicates** to ensure that all well-typed programs use their internally bound classical and quantum variables linearly.

i.e. Wherever an assumption is introduced, as part of the typing rule that introduced it, we can check that that assumption is used linearly

Linear Predicates

We use local **linearity predicates** to ensure that all well-typed programs use their internally bound classical and quantum variables linearly.

i.e. Wherever an assumption is introduced, as part of the typing rule that introduced it, we can check that that assumption is used linearly

Predicate:

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

says that classical (resp. quantum) variable x of type A (resp. q) is used linearly within a typing judgement that b is of type B.

Structural Proto-Quipper I

Qubit rule:

$$\frac{}{!\Delta; \{q\} \vdash q : \textbf{qubit}} \ (ax_q)$$

Issue

How do we ensure the (linear) context is empty?

Structural Proto-Quipper I

Qubit rule:

$$\frac{}{!\Delta; \{q\} \vdash q : \textbf{qubit}} \ (ax_q)$$

Issue

How do we ensure the (linear) context is empty?

Solution

$$\frac{}{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash q : \textbf{qubit})} \ [\texttt{lin/q/var}]$$

Structural Proto-Quipper II

Application rule:

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

Issue

How do we split the (linear) context into Γ_1, Γ_2, Q_1, Q_2?

Structural Proto-Quipper II

Application rule:

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

Issue

How do we split the (linear) context into Γ_1, Γ_2, Q_1, Q_2?

Solution

$$\frac{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash c : A \multimap B) \quad x \notin \texttt{FV(a)}}{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash ca : B)}\ [\texttt{lin/app1}]$$

$$\frac{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash a : A) \quad x \notin \texttt{FV(c)}}{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash ca : B)}\ [\texttt{lin/app2}]$$

$$\ldots$$

Structural Proto-Quipper II

Application rule:

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

Issue

How do we split the (linear) context into Γ_1, Γ_2, Q_1, Q_2?

Solution

$$\frac{\ldots}{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash c : A \multimap B) \quad q \notin \texttt{FQV(a)}}\ [\texttt{lin/q/app1}]$$

$$\frac{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash a : A) \quad q \notin \texttt{FQV(c)}}{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash ca : B)}\ [\texttt{lin/q/app2}]$$

Structural Proto-Quipper III

λ_1 rule

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

Issue

How can we select out variable x from the context?

Structural Proto-Quipper III

λ_1 rule

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

Issue

How can we select out variable x from the context?

Solution

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

Difference from (Sano et al., 2023)

- Our linearity predicates are on variables on typing judgements, while the original predicate was made on terms themselves.

Difference from (Sano et al., 2023)

- Our linearity predicates are on variables on typing judgements, while the original predicate was made on terms themselves.
- This is due to the fact that there is no term-level difference in Proto-Quipper between linear and structural resources.

Difference from (Sano et al., 2023)

- Our linearity predicates are on variables on typing judgements, while the original predicate was made on terms themselves.
- This is due to the fact that there is no term-level difference in Proto-Quipper between linear and structural resources.

Non-Linear Resources

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

Difference from (Sano et al., 2023)

- Our linearity predicates are on variables on typing judgements, while the original predicate was made on terms themselves.
- This is due to the fact that there is no term-level difference in Proto-Quipper between linear and structural resources.

Non-Linear Resources

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

- Non-linear resources are always being used linearly, counter-intuitively.

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.
2. We have constant terms $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev`

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.
2. We have constant terms $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev`
   1. $\texttt{box}^{\mathsf{T}}$ takes a circuit generating function and returns a circuit.

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.
2. We have constant terms $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev`
   1. $\texttt{box}^{\mathsf{T}}$ takes a circuit generating function and returns a circuit.
   2. `unbox` takes a circuit and returns a circuit generating function.

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.
2. We have constant terms $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev`
   1. $\texttt{box}^{\mathsf{T}}$ takes a circuit generating function and returns a circuit.
   2. `unbox` takes a circuit and returns a circuit generating function.
   3. `rev` reverses a circuit.

Circuits in Proto-Quipper

In Proto-Quipper, circuits are modeled as terms (t, C, a) with t, a terms and C is an uninterpreted circuit from a countable set of circuit constants (one for every possible circuit).

1. Every circuit constant is equipped with functions `IN` and `OUT` which specify its inputs and outputs.
2. We have constant terms $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev`
   1. $\texttt{box}^{\mathsf{T}}$ takes a circuit generating function and returns a circuit.
   2. `unbox` takes a circuit and returns a circuit generating function.
   3. `rev` reverses a circuit.

Circuit Typing

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

Issues with Circuits in Proto-Quipper

1. Because circuits are uninterpeted, we cannot say that the input Q_1 of C entails that t is of type T (same for output)

Issues with Circuits in Proto-Quipper

1. Because circuits are uninterpeted, we cannot say that the input Q_1 of C entails that t is of type T (same for output)
2. Proto-Quipper's implementation of $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev` requires complex operations on lists to create new circuits, append circuits together, and reverse circuits.

Issues with Circuits in Proto-Quipper

1. Because circuits are uninterpeted, we cannot say that the input Q_1 of C entails that t is of type T (same for output)
2. Proto-Quipper's implementation of $\texttt{box}^{\mathsf{T}}$, `unbox`, and `rev` requires complex operations on lists to create new circuits, append circuits together, and reverse circuits.
   Remember, this is what this project seeks to avoid!

Circuits in SPQ

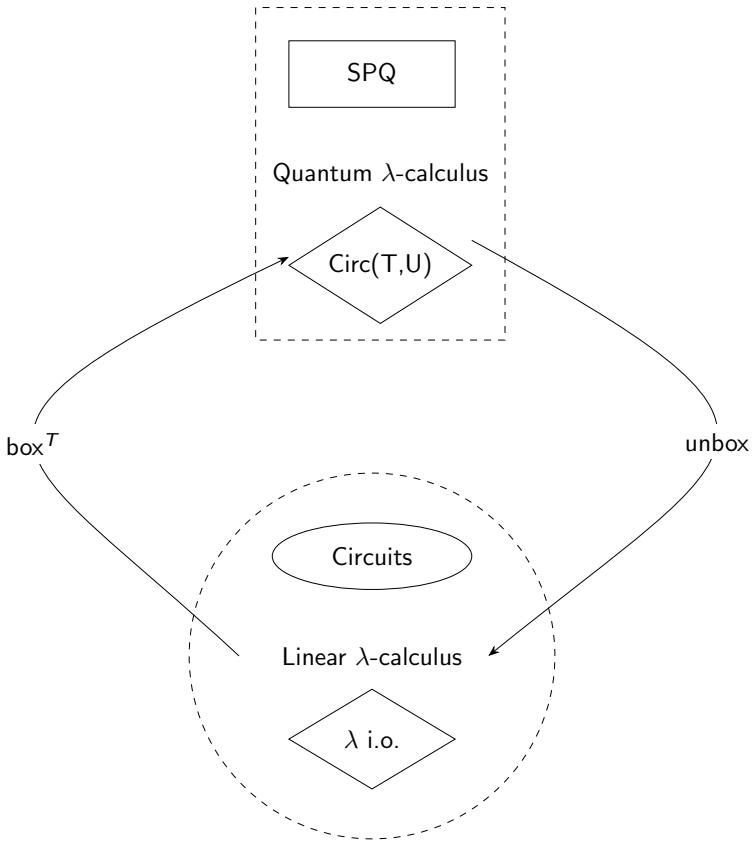

Figure: Two Levels of SPQ

Instead, we treat everything quantum in Structural Proto-Quipper as elements in a linear λ-calculus such that:

Circuits in SPQ

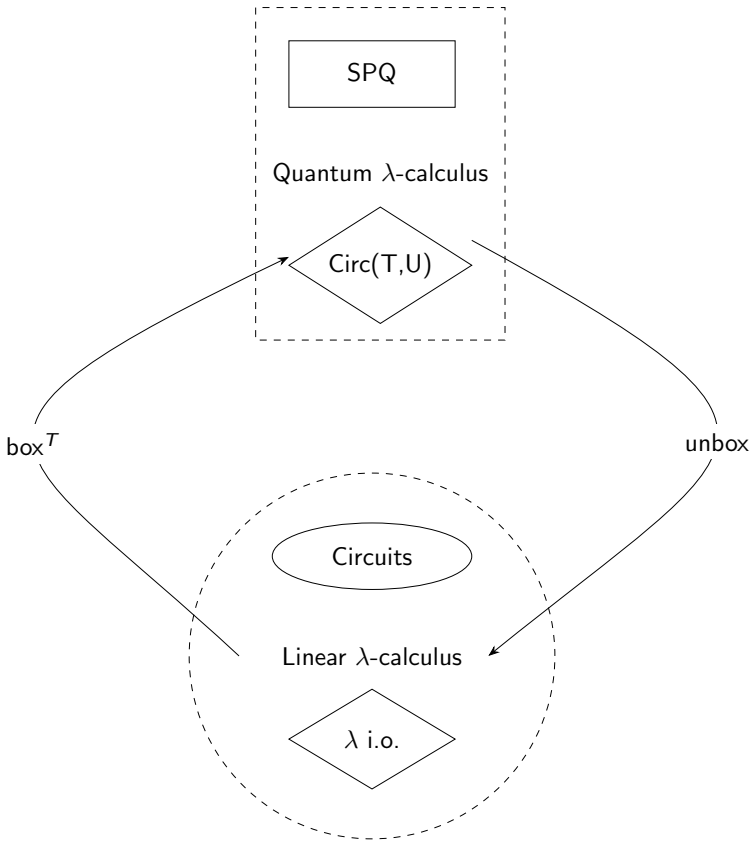

Figure: Two Levels of SPQ

Instead, we treat everything quantum in Structural Proto-Quipper as elements in a linear λ-calculus such that:

- quantum wires are represented by tuples of variables passed into...

Circuits in SPQ

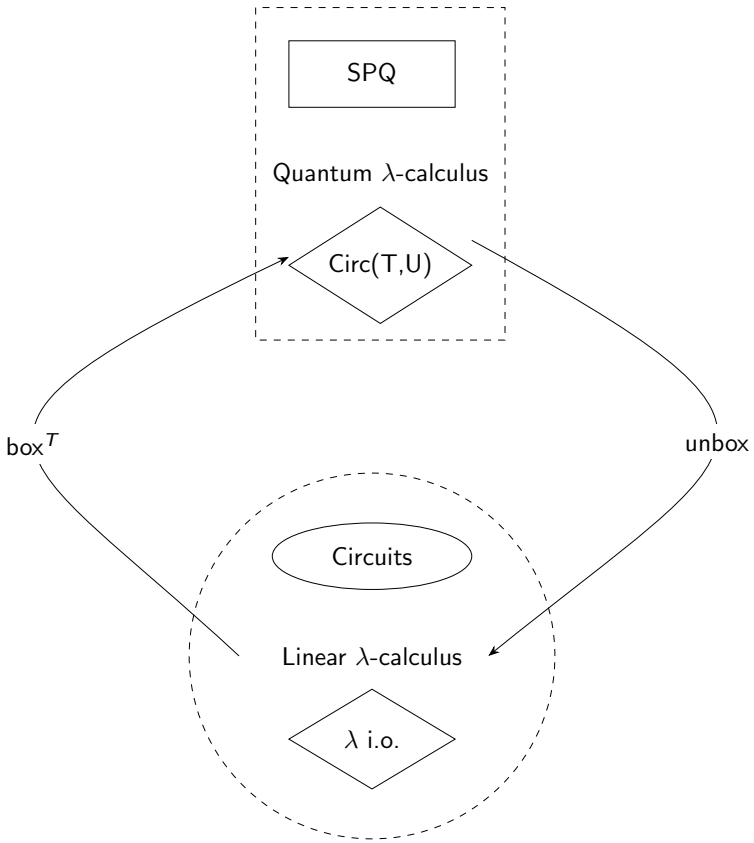

Figure: Two Levels of SPQ

Instead, we treat everything quantum in Structural Proto-Quipper as elements in a linear λ-calculus such that:

- quantum wires are represented by tuples of variables passed into...
- quantum circuits, regarded as function abstractions.

Circuits in SPQ

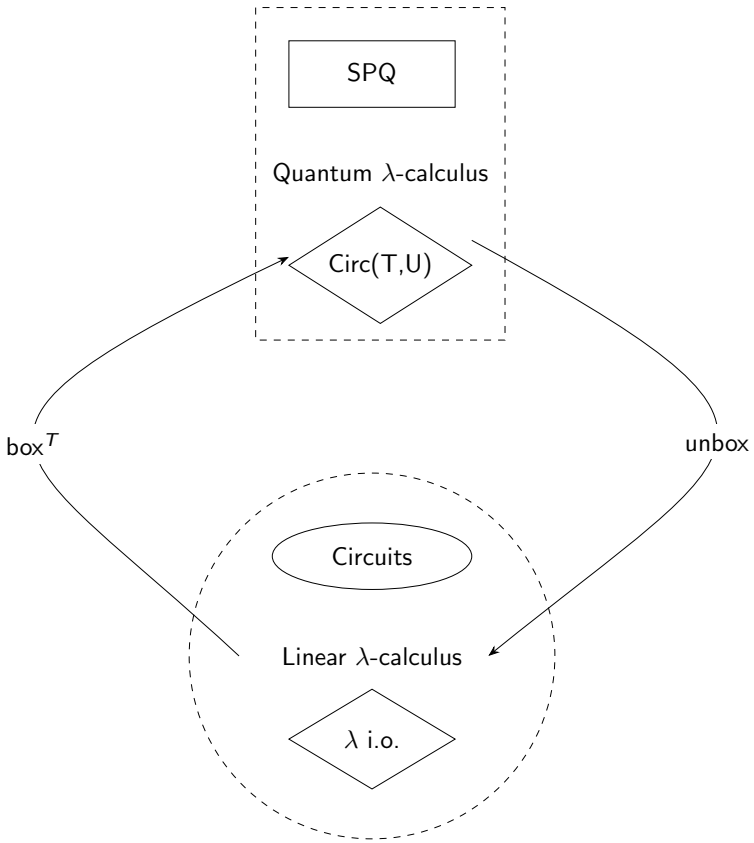

Figure: Two Levels of SPQ

Instead, we treat everything quantum in Structural Proto-Quipper as elements in a linear λ-calculus such that:

- quantum wires are represented by tuples of variables passed into...
- quantum circuits, regarded as function abstractions.

Appending circuits becomes function composition & creating new circuits is simply the identity map.

Circuits in SPQ

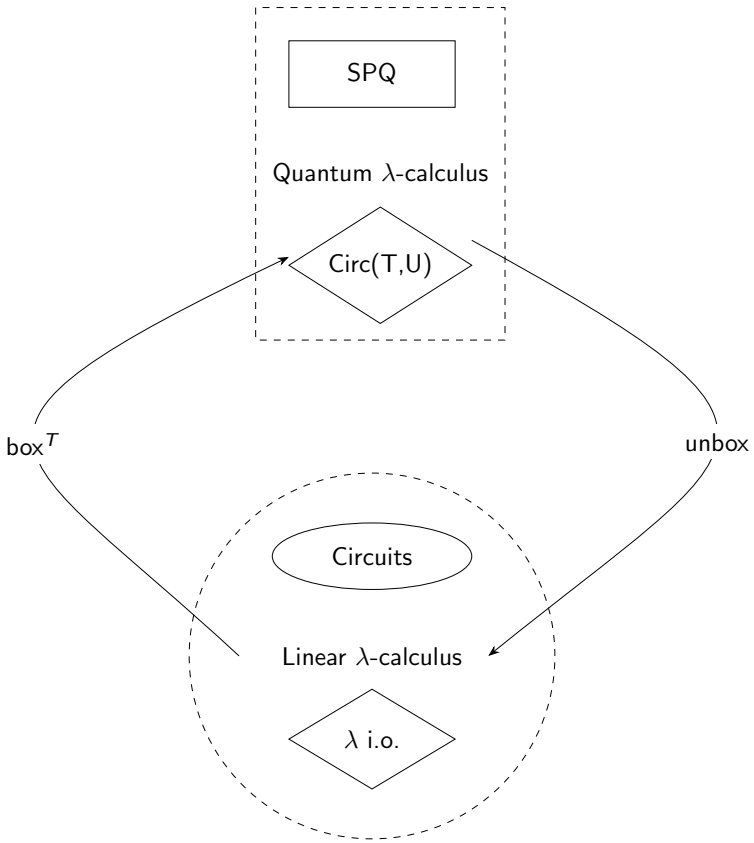

Figure: Two Levels of SPQ

$\texttt{box}^{\mathsf{T}}$ moves us from our circuit level to SPQ and `unbox` moves us SPQ to our circuit level.

Circuits in SPQ

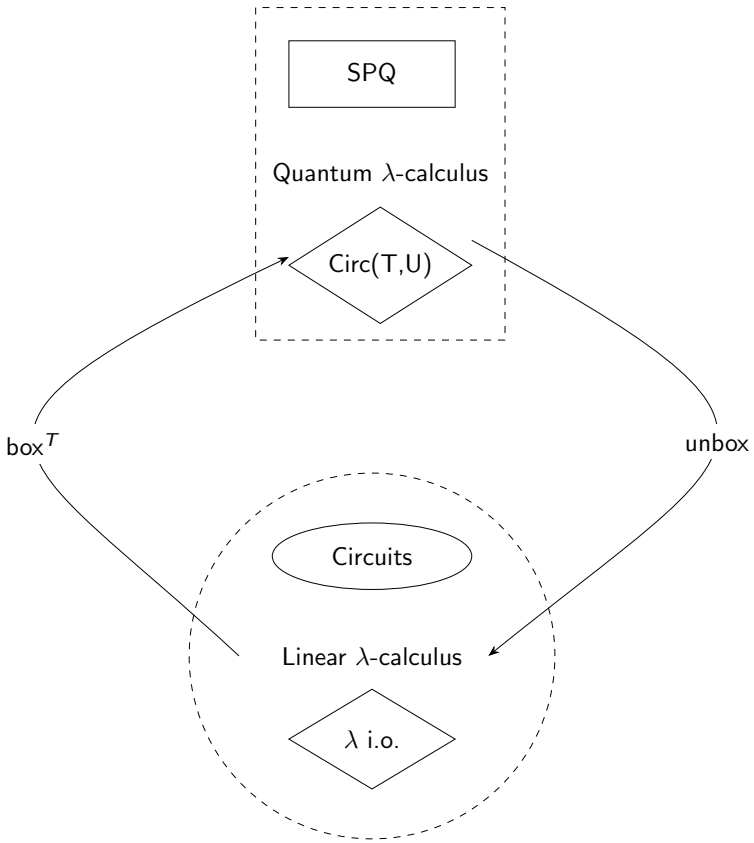

Figure: Two Levels of SPQ

$\texttt{box}^{\mathsf{T}}$ moves us from our circuit level to SPQ and `unbox` moves us SPQ to our circuit level.

Typing Circuits

$$\frac{\Gamma \vDash C : T \multimap U \quad \Gamma \vDash u : U \multimap U}{\Gamma \vDash (C, u) : \mathsf{Circ(T,\ U)}}$$

LF Encoding of Types

$$\begin{aligned}\mathsf{tp} &: \mathsf{type}.\\ \mathsf{tm} &: \mathsf{type}.\\ \mathsf{clam} &: \mathsf{type}.\\ \mathsf{ctp\_base} &: \mathsf{type}.\\ \mathsf{ctp\_circ} &: \mathsf{type}.\\ \mathsf{qv} &: \mathsf{type}.\end{aligned}$$

$A, B ::= \ldots$
qubit
$A \multimap B$
$\mathsf{Circ}(T, U)$

$$\begin{aligned}\mathsf{tp/qubit} &: \mathsf{tp}.\\ \mathsf{tp/}\multimap &: \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}.\\ \mathsf{tp/circ} &: \mathsf{ctp\_base} \to \mathsf{ctp\_base} \to \mathsf{tp}.\end{aligned}$$

LF Encoding of Terms

$$\begin{aligned}\mathsf{tp} &: \mathsf{type}.\\ \mathsf{tm} &: \mathsf{type}.\\ \mathsf{clam} &: \mathsf{type}.\\ \mathsf{ctp\_base} &: \mathsf{type}.\\ \mathsf{ctp\_circ} &: \mathsf{type}.\end{aligned}$$

$a, b ::= \ldots$
q
$\lambda x.a$
(C, u)

$$\begin{aligned}\mathsf{tm/qubit} &: \mathsf{qv} \to \mathsf{tm}.\\ \mathsf{tm/lam} &: (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}.\\ \mathsf{tm/circ} &: \mathsf{clam} \to \mathsf{tm} \to \mathsf{tm}.\end{aligned}$$

LF Encoding of Linearity Predicates

$x : A$
`lin` $(x : A;\ \Gamma, x : A \vDash b : B)$
`lin/q` $(q;\ \Gamma, \{q\} \vDash b : B)$

```
oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.
```

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

$$\frac{}{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash q : \textbf{qubit})}\ [\texttt{lin/q/var}]$$

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

$$\frac{}{\texttt{lin/q}\ (q;\ \Gamma, \{q\} \vDash q : \textbf{qubit})}\ [\texttt{lin/q/var}]$$

$$\Downarrow$$

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

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

$$\frac{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash c : A \multimap B) \quad x \notin \texttt{FV(a)}}{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash ca : B)}\ [\texttt{lin/app1}]$$

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

$$\frac{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash c : A \multimap B) \quad x \notin \texttt{FV(a)}}{\texttt{lin}\ (x : B;\ \Gamma, x : B \vDash ca : B)}\ [\texttt{lin/app1}]$$

$$\Downarrow$$

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

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

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

LF Encoding of Linearity Predicates

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

oft : tm → tp → type.
lin : ({x:tm} oft x A
→ oft (b x) B) → type.
lin/q : (x : qv oft (b x) B) → type.

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

$$\Downarrow$$

oft/lam1 : {D:({x:tm} oft x A → oft (b x) B)} lin D
→ oft (lam b) (A ⊸ B).

LF Encoding of Circuits

$$A_{\text{circ}} ::= B_{\text{base}} \multimap C_{\text{base}}$$

$$w : B_{\text{base}}$$

$$C : A_{\text{circ}}$$
$$\texttt{clamlin}\ (w, W)$$

$$\frac{\Gamma \vDash C : T \multimap U \quad \Gamma \vDash u : U \multimap U}{\Gamma \vDash (C, u) : \text{Circ}(\text{T}, \text{U})}$$

clam : type.
ctp_base : type.
ctp_circ : type.

ctp_circ/—∘: ctp_base → ctp_base → ctp_circ.
ofctp_base : clam → ctp_base → type.
ofctp_circ : clam → ctp_circ → type.
clamlin : (clam → clam) → type.
oft/circ : ofctp_circ C (T ⊸ U) → oft u (U ⊸ U) → oft (tm/circ C u) (tp/circ T U).

Conclusion

- Structural Proto-Quipper is a proof-of-concept for (Sano et al., 2023)'s approach to mechanizing linear programming languages in a quantum setting.

Conclusion

- Structural Proto-Quipper is a proof-of-concept for (Sano et al., 2023)'s approach to mechanizing linear programming languages in a quantum setting.
- We further offer mechanization-optimized improvements to Proto-Quipper through treating circuits and wires as elements in a linear lambda calculus.

Conclusion

- Structural Proto-Quipper is a proof-of-concept for (Sano et al., 2023)'s approach to mechanizing linear programming languages in a quantum setting.
- We further offer mechanization-optimized improvements to Proto-Quipper through treating circuits and wires as elements in a linear lambda calculus.
- Remaining to show proofs of metatheoretic properties, like progress and type preservation and some kind of equivalence between Proto-Quipper and SPQ.

Conclusion

- Structural Proto-Quipper is a proof-of-concept for (Sano et al., 2023)'s approach to mechanizing linear programming languages in a quantum setting.
- We further offer mechanization-optimized improvements to Proto-Quipper through treating circuits and wires as elements in a linear lambda calculus.
- Remaining to show proofs of metatheoretic properties, like progress and type preservation and some kind of equivalence between Proto-Quipper and SPQ.
- Further work includes mechanizing Proto-Quipper-Dyn, which supports dynamic lifting (i.e. circuits can be changed based on their outputs).