Deconstructed Proto-Quipper: A Rational Reconstruction

Ryan Kavanagh (Université du Québec à Montréal)
Chuta Sano (McGill University)
Brigitte Pientka (McGill University)

Eastern Canada Logic and Programming Seminar 2026
Work published at LOPSTR+PPDP'26

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

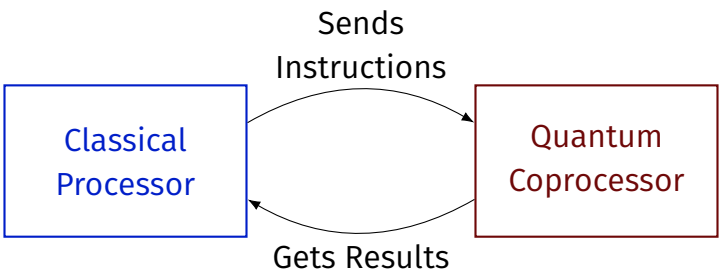
Quantum programming languages need

sound and tractable foundations

for quantum computing to deliver its full potential

Quantum Computing in 90 Seconds

The Quantum Programming Model



Quantum Coprocessing: Data

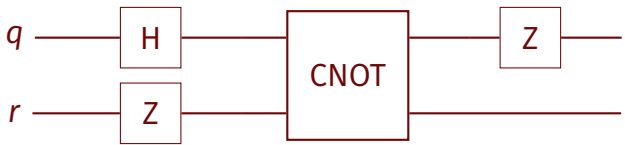
- **Qubits** are physical phenomena modelled by probability distributions on 0 and 1.
- They cannot be copied (no contraction).
- They require careful disposal (no weakening).
- Safe quantum languages must treat qubits **linearly**!

Quantum Coprocessing: Operations

- **Measuring** a qubit samples from its distribution to give a classical bit. The qubit “collapses” to this result.
- A **quantum gate** is a basic operation on qubits that changes their probability distributions.
- Quantum programs are sequences of gate applications, often notated as (and called) **quantum circuits**.

Quantum Circuits

Quantum circuits are box and wire diagrams that graphically describe the order in which gates (boxes) applied to qubits (wires):



Quipper: Circuit Metaprogramming

- **Quipper**[1] is a functional metaprogramming language for generating and manipulating quantum circuits with up to trillions of gates.
- It is an unsafe embedded language in Haskell: programs can have runtime errors.

[1] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. *Quipper: a scalable quantum programming language.* PLDI '13.

Proto-Quipper: Sound Foundations for Quipper

- The Proto-Quipper family of languages gives sound foundations to essentialized Quipper fragments.
- Large family (PQ-S in Ross's 2016 thesis, PQ-M at QPL'18, PQ-D at RC'20, PQ-K at TYPES'23, PQ-Dyn at POPL'23, PQ-C at QPL'25, etc.)
- They often rely on complex or ad hoc abstractions, e.g.:
  - fresh name generation
  - effectful semantics in terms of abstract operations on infinite sets

Contribution: Tractable Logical Foundations

1. **Proto-Quipper-A**, a rational reconstruction of Proto-Quipper-M on adjoint-logical foundations with a pure call-by-value semantics.
2. A recipe for normalizing terms with uninterpreted constants using commuting conversions.
3. A general logical relations technique for normalizing substructural systems based on approximate typing.

From Proto-Quipper-M to Proto-Quipper-A

Essentialized Proto-Quipper-M

Proto-Quipper-M is a linear λ-calculus extended with types:

- Wire types $U, S ::= \mathbb{Q} \mid 1 \mid U \otimes S$ inhabited by wire names and tuples of names
- Boxed circuits $\mathrm{Circ}(U, S)$ with input wires of shape U and outputs of shape S

It is extended with circuit manipulating operations

$$\mathrm{apply} : \mathrm{Circ}(U, S) \otimes U \multimap S \qquad \mathrm{box} : !(U \multimap S) \multimap !\mathrm{Circ}(U, S)$$

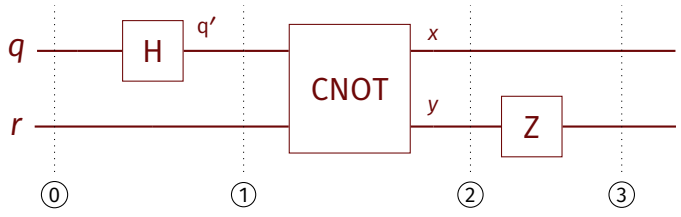
apply generates *fresh names* for the wires S.

Effectful Programming in Proto-Quipper-M: Circuit State

Given gates $\mathsf{H}, \mathsf{Z} : \mathsf{Circ}(\mathbb{Q}, \mathbb{Q})$ and $\mathsf{CNOT} : \mathsf{Circ}(\mathbb{Q} \otimes \mathbb{Q}, \mathbb{Q} \otimes \mathbb{Q})$, evaluating $q : \mathbb{Q}, r : \mathbb{Q} \vdash M : \mathbb{Q} \otimes \mathbb{Q}$ given by

$$
\begin{aligned}
M = \ & \mathsf{let}\ q' = \mathsf{apply}(\mathsf{H}, q)\ \mathsf{in} && ① \\
& \mathsf{let}\ \langle x, y \rangle = \mathsf{apply}(\mathsf{CNOT}, \langle q', r \rangle)\ \mathsf{in} && ② \\
& \langle x, \mathsf{apply}(\mathsf{Z}, y) \rangle && ③
\end{aligned}
$$

effectfully extends a circuit prefix that has output wires q and r:



Boxing Circuits for Reuse

We can box $q : \mathbb{Q}, r : \mathbb{Q} \vdash M : \mathbb{Q} \otimes \mathbb{Q}$,

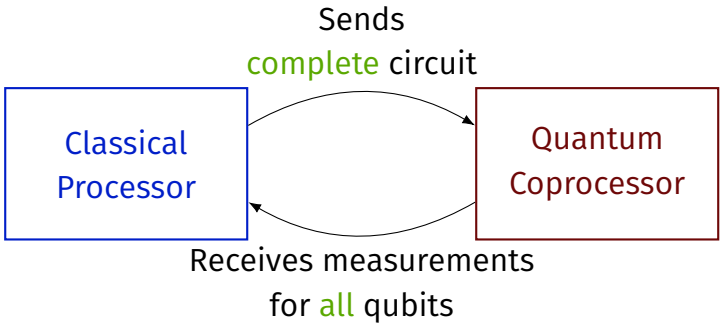
$$\vdash \mathrm{box}(\lambda \langle q, r \rangle : \mathbb{Q} \otimes \mathbb{Q}.\, M) : !\mathrm{Circ}(\mathbb{Q} \otimes \mathbb{Q}, \mathbb{Q} \otimes \mathbb{Q})$$

and then reuse it later.

For example, we can concatenate M with itself:

$$\vdash \begin{pmatrix} \mathrm{let}\ m = \mathrm{box}(\lambda \langle q, r \rangle .\, M)\ \mathrm{in} \\ \mathrm{box}(\lambda x : \mathbb{Q} \otimes \mathbb{Q}\ .\ \mathrm{let}\ x' = \mathrm{apply}(m, x)\ \mathrm{in} \\ \qquad\qquad \mathrm{apply}(m, x')) \end{pmatrix} : !\mathrm{Circ}(\mathbb{Q} \otimes \mathbb{Q}, \mathbb{Q} \otimes \mathbb{Q}).$$

“Static” Circuit Programming Model



(Chuta will present a “dynamic” or “hybrid” model that interleaves incomplete circuit generation and measurement)

Proto-Quipper-A Combines Two Languages

- A circuit description language: a highly restricted first-order linear λ-calculus. Not directly used by programmers.
- A functional metaprogramming language: a full linear/non-linear (LNL) λ-calculus.
- An adjoint-logical framework integrates both so that the functional layer can safely manipulate boxed circuits.

Circuit Description Language: Types

Wire types are as before, but in the *circuit* language instead of the functional language:

$$U, S ::= 1 \mid \mathbb{Q} \mid U \otimes S$$

The circuit description language's types are generated by

$$A_{\mathrm{Q}}, B_{\mathrm{Q}} ::= U \mid U \multimap S$$

Terms of type $U \multimap S$ are code that generate a circuit with outputs of shape S when applied to inputs of shape U.

These grammars **severely restrict** the data we allow, e.g., no $A_{\mathrm{Q}} \multimap (B_{\mathrm{Q}} \multimap B'_{\mathrm{Q}})$ or $U \otimes (U \multimap S)$.

Circuit Description Language: Terms

$C, D ::= g$	gate constants (boxes)
$\mid CD$	circuit application (horizontal composition)
$\mid \lambda x.\, C$	input naming
$\mid \langle C, D \rangle$	pairing (vertical stacking)
$\mid \langle\rangle$	unit (vertical stacking unit)
$\mid \mathsf{let}\ \langle x, y \rangle = C\ \mathsf{in}\ D$	naming C's output wires for use in D
$\mid \mathsf{let}\ \langle\rangle = C\ \mathsf{in}\ D$	unit elimination (horizontal juxtaposition)

Linear typing rules are standard, but subject to the restricted type grammar!

The Functional Layers: Linear/Non-Linear Calculus (LNL)

Programmers use the functional language formed of a linear layer L and an unrestricted (non-linear) layer U. They are integrated by adjoint shifts:

$$A_U, B_U ::= 1_U \mid A_U \times B_U \mid A_U \to B_U \mid \uparrow_L^U A_L$$
$$A_L, B_L ::= 1_L \mid A_L \otimes B_L \mid A_L \multimap B_L \mid \downarrow_L^U A_U$$

Subscripts indicate the layer.

The Functional Layers: LNL terms

Terms are standard, except for shifts:

$M, N ::= x$	variables
$\mid \langle\rangle \mid \mathsf{let}\ \langle\rangle = M\ \mathsf{in}\ N$	units
$\mid \langle M, N\rangle \mid \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N$	pairs
$\mid \lambda x.\, M \mid MN$	functions
$\mid \mathsf{susp}\, M \mid \mathsf{force}\, M$	suspensions ($\uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}}$)
$\mid \mathsf{down}(M) \mid \mathsf{let}\ \mathsf{down}(x) = M\ \mathsf{in}\ N$	value embeddings ($\downarrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{U}}$)

Adjoint Logic Integrates Layers

Extend the linear functional layer L with suspended circuit-layer Q terms:

$$A_{\mathsf{L}} ::= \cdots \mid \uparrow^{\mathsf{L}}_{\mathsf{Q}} A_{\mathsf{Q}}$$

The circuit layer is a priori, so the A_{Q} grammar is unchanged.

Must also extend the term languages:

$$M ::= \cdots \mid \mathsf{susp}\, C \mid \mathsf{let}\ \langle\rangle = C\ \mathsf{in}\ N \mid \mathsf{let}\ \langle x, y\rangle = C\ \mathsf{in}\ N$$
$$C ::= \cdots \mid \mathsf{force}\, M$$

We can now encode Proto-Quipper-M's functional types for quantum resources, and implement box and apply for programmer use (exercise):

$$\ulcorner \mathbb{Q} \urcorner \triangleq \uparrow^{\mathsf{L}}_{\mathsf{Q}} \mathbb{Q} \qquad \ulcorner U \otimes S \urcorner \triangleq \uparrow^{\mathsf{L}}_{\mathsf{Q}} \ulcorner U \urcorner \otimes \uparrow^{\mathsf{L}}_{\mathsf{Q}} \ulcorner S \urcorner \qquad \ulcorner \mathrm{Circ}(S, U) \urcorner \triangleq \uparrow^{\mathsf{L}}_{\mathsf{Q}} (S \multimap U)$$

Typing Judgment

A mode preorder specifies allowed dependencies between layers, where the lesser layer can depend on values from the higher layer:

$$\mathsf{L} < \mathsf{Q} \qquad \mathsf{Q} < \mathsf{L} \qquad \mathsf{L} < \mathsf{U}$$

It restricts the contexts of typing judgments:

$\Delta_{\mathsf{U}} \vdash M : A_{\mathsf{U}}$	Δ_{U} only contains $x : B_{\mathsf{U}}$
$\Delta \vdash M : A_{\mathsf{L}}$	Δ can contain variables from any layer
$\Delta \vdash C : A_{\mathsf{Q}}$	Δ can contain variables from any layer

Sample Typing Rules for $P ::= C \mid M$

$$\frac{}{\Delta_U, x : A \vdash x : A}$$

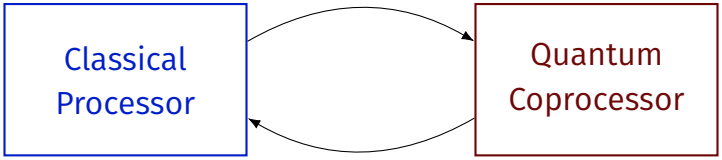
$$\frac{\Delta_U \vdash M : A_L}{\Delta_U \vdash \mathsf{susp}\, M : \uparrow_L^U A_L} \qquad \frac{\Delta \vdash C : A_Q}{\Delta \vdash \mathsf{susp}\, C : \uparrow_Q^L A_Q}$$

$$\frac{\Delta_U \vdash M : \uparrow_L^U A_L}{\Delta_U \vdash \mathsf{force}\, M : A_L} \qquad \frac{\Delta \vdash M : \uparrow_Q^L A_Q}{\Delta \vdash \mathsf{force}\, M : A_Q}$$

$$\frac{\Delta_U, \Delta_1 \vdash C : U \otimes S \qquad \Delta_U, \Delta_2, x : U, y : S \vdash M : A_L}{\Delta_U, \Delta_1, \Delta_2 \vdash \mathsf{let}\ \langle x, y \rangle = C\ \mathsf{in}\ M : A_L}$$

Commuting Conversions to Normalize Despite Constants

Revisiting the Static Model



1. Programmer implements $M : \uparrow^{\mathsf{L}}_{\mathsf{Q}}(U \multimap S) \equiv \ulcorner \mathsf{Circ}(U, S) \urcorner$
2. Initialize qubits $u : U$
3. Unquote and generate the circuit: normalize $(\mathsf{force}\, M)\, u : S$
4. Run it and measure outputs
5. Get the results

Crucial Point: We must eliminate all functional U, L subterms in step #3!

Circuit Normal Forms

- Close correspondence to graphical representation and are similar to A-normal forms.
- No functional subterms in applied circuits!
- Gate constants g have no computational content.

Canonical Forms (constructors)

$$K ::= \langle\rangle \mid \langle V, V' \rangle \mid \lambda x.C$$

Neutral Forms (stuck open terms)

$$R ::= x \mid gR \mid gK$$

Normal Forms (neutral eliminations around canonicals or neturals)

$$V ::= \text{let } \langle\rangle = R \text{ in } V \mid \text{let } \langle x, y \rangle = R \text{ in } V \mid R \mid K$$

Functional Normal Forms

Only run terms with no free functional variables, so no neutral terms.

Canonical Forms (constructors)

$$K ::= \langle\rangle \mid \langle V, V' \rangle \mid \lambda x.M$$

Normal Forms (neutral eliminations around canonicals or neturals)

$$V ::= \text{let } \langle\rangle = R \text{ in } V \mid \text{let } \langle x, y \rangle = R \text{ in } V \mid K$$

Pure Call-By-Value Reduction Semantics

Three classes of reduction rules. Normal form stratification syntactically characterizes when to perform permuting conversions:

1. Commute destructors past lets eleminating neutrals:

$$\mathsf{force}\,(\mathsf{let}\ \langle x, y\rangle = R\ \mathsf{in}\ V) \longrightarrow \mathsf{let}\ \langle x, y\rangle = R\ \mathsf{in}\ \mathsf{force}\,V$$

$$g\,(\mathsf{let}\ \langle\rangle = R\ \mathsf{in}\ V) \longrightarrow \mathsf{let}\ \langle\rangle = R\ \mathsf{in}\ gV$$

$$\mathsf{let}\ \langle a, b\rangle = (\mathsf{let}\ \langle x, y\rangle = R\ \mathsf{in}\ V)\ \mathsf{in}\ P \longrightarrow \mathsf{let}\ \langle x, y\rangle = R\ \mathsf{in}\ \mathsf{let}\ \langle a, b\rangle = V\ \mathsf{in}\ P$$

CBV Reduction Semantics II

2. Eliminate canonical forms as usual:

$$\begin{aligned} \text{let } \langle x, y\rangle = \langle V_1, V_2\rangle \text{ in } P &\longrightarrow [V_1, V_2/x, y]P \\ \text{force}\,(\text{susp}\,P) &\longrightarrow P \\ (\lambda x.P)\,K &\longrightarrow [K/x]P \end{aligned}$$

3. Close up under appropriate congruences:

$$\begin{aligned} \text{force}\,M &\longrightarrow \text{force}\,M' \quad \text{if } M \longrightarrow M' \\ \langle P, Q\rangle &\longrightarrow \langle P', Q\rangle \quad \text{if } P \longrightarrow P' \end{aligned}$$

Logical-Relations Normalization Proofs for Substructural Systems

Logical Relations for Normalization

- We use a Kripke logical relation $\mathcal{N}(A)$ to characterize normalization of terms with only free circuit variables to normal forms of type A.
- Two parted proof technique:
  - Show that if $(\Delta \vdash P) \in \mathcal{N}(A)$, then P normalizes.
  - Show that if $\Delta \vdash P : A$, then $(\Delta \vdash P) \in \mathcal{N}(A)$.
- Substructurality makes the second part tedious.

Fundamental Theorem of Logical Relations

If $\Delta \vdash P : A$, then $(\Gamma \vdash [\sigma]P) \in \mathcal{N}(A)$ for all reducible substitutions $(\Gamma \vdash \sigma) \in \mathcal{N}(\Delta)$.

Proof: By induction on $\Delta \vdash P : A$.

Case

$$\frac{\Delta_U, \Delta_1 \vdash C : A \qquad \Delta_U, \Delta_2 \vdash D : B}{\Delta_U, \Delta_1, \Delta_2 \vdash \langle C, D \rangle : A \otimes B}$$

Let $(\Gamma \vdash \sigma) \in \mathcal{N}(\Delta_U, \Delta_1, \Delta_2)$ be arbitrary.
To use the IH, we need to split σ into $(\Gamma_i \vdash \sigma_i) \in \mathcal{N}(\Delta_U, \Delta_i)$ for $i = 1, 2$.
Sit down and cry into your proof assistant.

Approximate Typing to the Rescue!

Key observation: substructurality is orthogonal to normalization when reduction is untyped.

Approximate typing $\Delta \Vdash P : A$ forgets substructurality restrictions and enables textbook logical-relations proofs.

$$\frac{\Delta_U, \Delta_1 \vdash C : A \qquad \Delta_U, \Delta_2 \vdash D : B}{\Delta_U, \Delta_1, \Delta_2 \vdash \langle C, D \rangle : A \otimes B} \quad \rightsquigarrow_{\text{approximated by}} \quad \frac{\Delta \Vdash C : A \qquad \Delta \Vdash D : B}{\Delta \Vdash \langle C, D \rangle : A \otimes B}$$

The Normalization via Approximate Typing Recipe

1. **Proposition (Structural Embedding):** If $\Delta \vdash P : A$, then $\Delta \Vdash P : A$.
2. Prove the Fundamental Theorem of Logical Relations for the approximate system using the standard techniques.
3. **Corollary:** If $\Delta \Vdash P : A$ (approximately), then P normalizes.
4. **Corollary:** If $\Delta \vdash P : A$ (substructurally), then P normalizes.

This recipe, with a carefully constructed Kripke logical relation that handles neutral let eliminations, shows that our language is normalizing.

Evaluation

- Our logically-motivated abstractions simplify mechanization: we mechanized normalization and subject reduction for Proto-Quipper-A and its approximate version in 2750 lines of Beluga code using standard techniques.
- Our foundation is extensible: we have since captured dynamic lifting, integrating compile-time and run-time semantics for functional and quantum systems.

Future Work

- Carefully flesh out a denotational semantics to give a precise comparison with Proto-Quipper-M.
- Extend with natural number dependent types to capture functions on circuits of variable width, e.g.,

$$\Pi\, n : \mathbb{N}.\Pi\, m : \mathbb{N}.\mathsf{Circ}(\mathbb{Q}^n, \mathbb{Q}^n) \multimap \mathsf{Circ}(\mathbb{Q}^m, \mathbb{Q}^m) \multimap \mathsf{Circ}(\mathbb{Q}^{n+m}, \mathbb{Q}^{n+m}).$$

- Extend with type-safe pattern matching on circuits to support circuit analysis and optimization (Rémi)

Thank You

1. **Proto-Quipper-A**, a rational reconstruction of Proto-Quipper-M on adjoint-logical foundations with a pure call-by-value semantics.
2. A recipe for normalizing terms with uninterpreted constants using commuting conversions.
3. A general logical relations technique for normalizing substructural systems based on approximate typing.