A Functional Quantum Programming Language

Thorsten Altenkirch, Jonathan Grattage (Nottingham)

Presented by Wil Cram

October 8, 2025

Outline

1. **Motivation**
2. The Category FQC
3. QML Semantics
4. Conclusion

Motivation

Over the years, the abstractions used to define quantum algorithms have been lifted from Deutsch's quantum turing machines to quantum circuits and eventually quantum programming languages

One of the first major QPLs was QCL, an imperative language using the qRAM model of computation

Motivation

In 2004, van Tonder defined a quantum lambda calculus which allows arbitrary quantum computation but forbids measurement

That same year, Selinger proposed a rudimentary *functional* QPL which supported measurement, relying on classical control and quantum data

What about *quantum* control and quantum data?

A QML Program

$$\texttt{had} : \mathbf{Q}_2 \multimap \mathbf{Q}_2$$

$$\begin{aligned}\texttt{had}\ \texttt{x} = \ & \mathbf{if}^\circ x \\ & \mathbf{then}\{\texttt{qfalse} \mid \texttt{-qtrue}\} \\ & \mathbf{else}\{\texttt{qfalse} \mid \texttt{qtrue}\}\end{aligned}$$

The $\mathbf{if}^\circ$ statement does *not* measure the qubit (unlike **if**), this is our quantum control

How do we interpret this as a quantum circuit? Is it even a well-formed program?

Outline

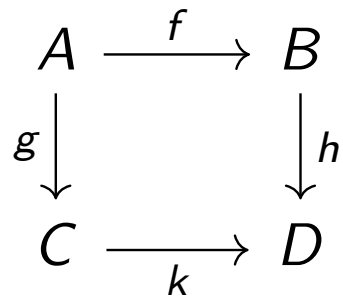
1. Motivation
2. **The Category FQC**
3. QML Semantics
4. Conclusion

Background - Category Theory

A category is a collection of objects and arrows between them, such that:

1. Compatible arrows compose
2. Composition is associative
3. An identity arrow exists for every object

We can visualize some part of a category using a diagram. We say the diagram **commutes** if every path composes to give the same arrow.



Background - Category Theory

Some examples of categories include:

1. Sets and functions between them
2. "*-morphisms" between Groups, Rings, Vector Spaces
3. Any partial order (lattice)

Any (terminating) programming language gives rise to a category of types and functions between them:

$$\texttt{toString} : \texttt{Int} \to \texttt{String}$$

The Category **FQC**

FQC (Finite Quantum Computation) is the category we will use to determine what our QML programs "mean"

Objects are finite sets (A), but we identify them with the Hilbert space ($\mathbb{C}^A$) they generate

Example: the object $\mathbf{Q}_2 \equiv \mathbb{C}^2$ is the one-qubit space

Arrows are unfortunately a bit more complicated...

Arrows in **FQC**

What should happen when I run a program on my quantum computer?

An arrow in **FQC** is given by the following data:

1. A basis H for the space of initial heaps
2. A "heap initializer" $h \in \mathbb{C}^H$
3. A basis G for the space of garbage states
4. A unitary $\phi : \mathbb{C}^A \otimes \mathbb{C}^H \to \mathbb{C}^B \otimes \mathbb{C}^G$

Arrows compose by stacking the transforms together

We define a subcategory $\mathbf{FQC}^\circ$ of arrows which produce no garbage

Extensional Equality

When doing classical computation, we consider programs as "extensionally equal" if they have the same output for every valid input

```
x := 0; for i in [1..n]:  x += i
```

$$\cong$$

```
x := n(n+1)/2
```

More formally, the induced functions $A \to B$ are equal as relations

Extensional Equality in **FQC**

In the quantum case, we use the density matrix formulation and the partial trace

$$\mathrm{Tr}^{A}\,\frac{1}{2}(|10\rangle\langle 10| + |11\rangle\langle 11|) = \frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)$$

Extensional equality of quantum computations can now be defined similarly to the classical case

We "quotient out" by this extensional equality when talking about the collection of arrows between objects

Outline

Background - Type Systems

In order to ensure our programs are semantically sound, we want a way of saying what kinds of values work with which kinds of operation

$$(+) : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

This can be done with a type system, and every sound term in the programming language can be assigned a type

```
if (x == 2) then 5 else 7 :  Int
```

Background - Type Derivations

A **context** Γ is a mapping from variables to types

We have *typing judgements* of the form $\Gamma \vdash t : \sigma$, read as "in context Γ, t has type σ."

Typing rules allow us to derive judgements in a procedural way:

$$\textsc{If-T}\quad \frac{\Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma \quad \Gamma \vdash b : \mathsf{Bool}}{\Gamma \vdash \text{if } b \text{ then } t_1 \text{ else } t_2 : \sigma}$$

QML offers a **denotational semantics**: a procedure to transform typing derivations into **FQC** arrows (quantum circuits)

Linear Logic

The typing rules of QML are based on (strict) linear logic

In such logics, derivations require any subderivations to be used exactly once

Treats judgements as a "resource" that can't be thrown away or duplicated

Linear Logic

```
let s1 = String::from("hello");
let s2 = s1;

println!("{s1}, world!");
```

Rust's borrow checker uses linear logic to implement move semantics

Why use linear logic for QPLs?

Strictness

We discussed earlier the subcategory $\mathbf{FQC}^\circ$ of strict arrows (no garbage)

In QML, there is also a strict type judgement $\Gamma \vdash^\circ t : \sigma$ with the additional requirement that t is strict

$$\textsc{Strict}\ \frac{\Gamma \vdash^\circ t : \sigma}{\Gamma \vdash t : \sigma}$$

Strictness is really just saying that decoherence never happens during execution

Sum and Product Types

Given some types τ and σ, we would like to form composite types using them

Values of the product $\tau \otimes \sigma$ can be viewed as tuples (x, y) with $x : \tau, y : \sigma$

Values of the sum $\tau \oplus \sigma$ can be viewed as a "tagged union" which is either a τ or a σ

In QML, the *only* types are those that can be created via sums and products of the unit type $\mathbf{1}$

$$\mathbf{Q}_2 = 1 \oplus 1$$

Denotational Semantics of QML

For every type, we can define its magnitude which is the qubit-width of its values

$$|1| = 0$$

$$|\sigma \otimes \tau| = |\sigma| + |\tau|$$

$$|\sigma \oplus \tau| = \max(|\sigma|, |\tau|) + 1$$

We can identify a type with the **FQC** object of the right size:

$$[[\sigma]] = \mathbf{Q}_2^{|\sigma|}$$

Denotational Semantics of QML

The denotation of a context is defined recursively:

$$[[\Gamma]] = [[(x_i : \tau_i)_i]] = \bigotimes_i [[\tau_i]]$$

If we construct a typing derivation, $\Gamma \vdash t : \sigma$, this implies that

$$[[t]] \in \mathbf{FQC}[[\Gamma]][[\sigma]]$$

In other words: the program t transforms $[[\Gamma]]$ (the inputs) into $[[\sigma]]$

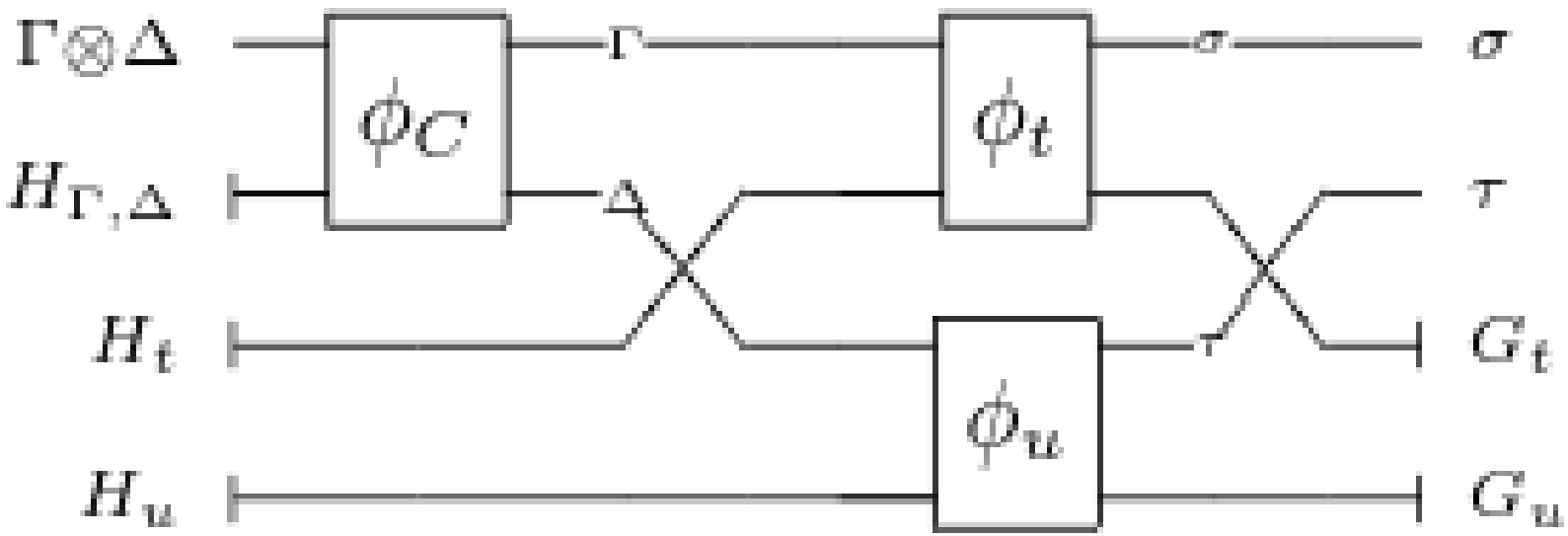
A taste of the rules

$$\textsc{Var-Strict}\ \frac{\emptyset}{x : \sigma \vdash^{\circ} x : \sigma}$$

$$\textsc{Var}\ \frac{\emptyset}{\Gamma, x : \sigma \vdash x : \sigma}$$

A taste of the rules

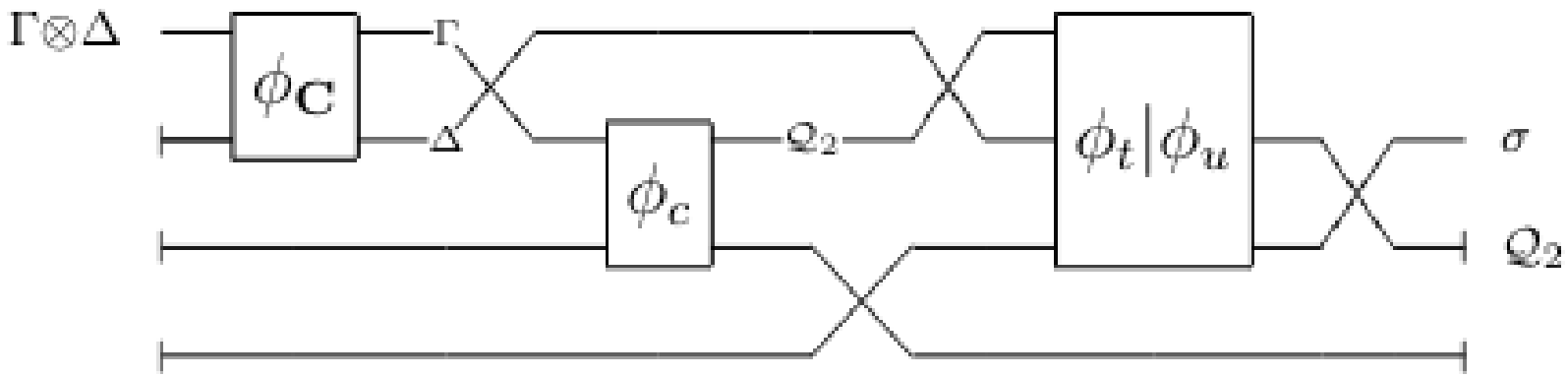
$$\textsc{Prod}\ \frac{\Gamma \vdash^{\circ} t : \sigma \quad \Delta \vdash^{\circ} u : \tau}{\Gamma \otimes \Delta \vdash^{\circ} (t, u) : \sigma \otimes \tau}$$



A taste of the rules

IF-MEASURE

$$\frac{\Gamma \vdash c : \mathbf{Q}_2 \quad \Delta \vdash t, u : \sigma}{\Gamma \otimes \Delta \vdash \textbf{if } c \text{ then } t \text{ else } u : \sigma}$$



Orthogonality

For $\mathbf{if}^\circ$, we need an additional "orthogonality judgement" $t \perp u$, built up using similar inference rules

This is needed so that $\mathbf{if}^\circ$ is always unitary:
the branches don't "mix"

The separation of **if** and $\mathbf{if}^\circ$ is one of the defining features of QML

Outline

1. Motivation
2. The Category FQC
3. QML Semantics
4. **Conclusion**

Novel Results

Using the rules so far, we are able to construct arbitrary quantum functions between types and compile them to circuits

The authors claim this is all we need to write any quantum program! (why?)

The authors also have implemented a quantum circuit compiler for QML in Haskell[1] (a classical functional PL)

[1] https://arxiv.org/pdf/0806.2735

Takeaways

Linear logic and types are the natural choice for QPLs

QPLs should have decoherence embedded in their semantics

Higher-level abstractions are needed to reasonably describe the next generation of quantum algorithms

Additional References

D. A. Sofge, "A Survey of Quantum Programming Languages: History, Methods, and Tools"
https://ieeexplore.ieee.org/document/4455934

B. Pierce, "Advanced Topics in Types and Programming Languages", Section 1.2, *A linear type system*

(The original PHD thesis) J. J. Grattage, "A functional quantum programming language",
https://eprints.nottingham.ac.uk/10250/1/thesis.pdf