

Control-flow Expressiveness in Quantum Computing

Jake Trevor

In this talk

I will present some work in-review:

- Control-flow expressiveness in classical computing
- Why quantum is different
- The CCF Hierarchy

And some work in-progress:

- Semantics for the lattice

In classical computing,

there are two building blocks of control flow:

- Conditional behaviour (`if` statements)
- looping behaviour (`while` loops)

Looping behaviour subsumes conditional behaviour

```
if (a) {  
    b  
}
```

```
flag = true  
  
while (flag && a) {  
    b  
    flag = false  
}
```

And there are things we can write with while loops that we can't write without:

```
while (true) {  
    print("hello world")  
}
```

While *increases the expressiveness* of the language

Similar is true for if statements:

```
if (randomChoice()) {  
    print("heads")  
} else {  
    print("tails")  
}
```

So, very roughly ...

- Languages with `while` loops are more powerful than
- Languages with `if` statements, which are more powerful than
- Languages with no CF whatsoever

Aside: Is conditional behaviour really more powerful than flat?

Morally - yes, but the devil is in the details.

- If we are talking about instruction languages ...
- and an 'if statement' really means a conditional jump ...
→ Then the difference in expressiveness depends on the instruction set.

Lots of things can be done branchlessly, if you're clever

Aside: Is conditional behaviour really more powerful than flat?

- If all we have is a function pointer, (i.e. to an externally defined function) ...
- the only way to conditionally call it is with a `cjump`
→ i.e. exactly an if statement.

Key takeaway: Quantifying this difference is hard
But it does exist!

Why is QC any different?

Models of Quantum Computation

We will review the three big models of quantum computation:

1. Circuits
2. QRAM
3. Co-Processor Computers

Circuits

A Circuit represents *straight-line code*.
Generally, there isn't any control flow

Deutsch's algorithm:

```
function deutsch() {
    a, b = fresh
    X b
    H a b
    f(a, b) -- our function to measure
    H a

    return M a
}
```

Circuits

- Fixed number of qubits (allocation is static)
- Some fixed set of gates are applied (no control flow)
→ Actually, we sometimes slightly relax this.
- We sometimes allow conditioned gate application, based on some measurement result

As in teleportation:

```
function teleport(q) {  
    (a, b) <- fresh  
    H a  
    CX a b  
  
    CX q a  
    H q  
  
    [q', a'] = M [q, a]  
  
    if (a') { X b }  
    if (q') { Z b }  
}
```

QRAM (Quantum Random Access Machine)

First presented by Knill

- Access to some (large) amount of quantum memory
 - Generally allows control flow via jumps
- Fully expressive

Co-processor

→ Extension to QRAM

Two key components:

- the *QPU* (Quantum Processing Unit)
- the *scheduler* (really, just a regular classical computer).

The scheduler tells the QPU what to do

What about the other way?

QPU → Scheduler dataflow

In the co-processor model,
dataflow in this direction is
given special treatment.

It's called *Dynamic Lifting*

Why the special treatment?

Notionally, because dynamic lifting **expensive**

Requires the QPU to wait for some classical computation to finish.

Qubits are unstable - they can only hold their state for so long before *losing coherence*.

This is what the co-processor model tries to capture.

Not all dynamic lifting is interesting:

- $C \rightarrow Q \rightarrow C$ — pre- and post-processing
 - Cheap, boring
- $Q_1 \rightarrow C \rightarrow Q_2$ — measurement conditioned continuation
 - Expensive, interesting
 - especially if Q_2 depends on some data from Q_1 being coherently retained

We only really care about the latter case ...

But trying to capture this is very difficult.

Dynamic lifting makes the CF lattice a little more complex:

The Classical Control-Flow CCF Hierarchy:

- F, CC and CL are familiar
→ These are just the points of the classical CF hierarchy
- QC, CLQC and QL are new
→ These involve introducing *dynamic lifting* into the program.

This can help us classify all kinds of things:

- Hardware
- Abstract machines / Models of computation
- Algorithms
- Programming languages

What we looked at earlier:

At F (Flat):

- Circuits without conditioned application
- Deutsch's algorithm

At QC:

- Circuits with conditioned application
- Quantum teleportation

At QL:

- QRAM and Co-processor

Work in progress: Semantics for the Lattice

Idea:

- Produce a family of laguages ...
 - related by adding syntactic features ...
 - which capture different levels of the lattice.
- Give a semantics to each, and we have a formal way to study the lattice!

Problem: the full lattice is hard.

What is the difference between CLQC - QL?

- CLQC allows data dependencies on measurements for `ifs`, but not `whiles`
- QL allows both

How do we model this syntactically?

→ I haven't come up with a nice way to do this.

(Possibly Temporary) Solution: Reduced lattice

Effectful vs recorded measurement

A taste of what's cooking:

Give a denotational semantics, in terms of a function on state

The state of a QC is probabilistic; At any given time, it may occupy one of many given states, with some associated probability.

Therefore, we model the state as a list of fragments (p, σ) such that the sum of probabilities $\sum_p p = 1$

The data part of a fragment (σ) is comprised of three components:

1. The quantum state $|\phi\rangle$: a quantum state vector
2. The classical state: a classical bit vector
3. The name map: Mapping of *names* to *data addresses*

Name \rightarrow Either Int Int

- Left $x \rightarrow$ Qubit at index x
- Right $y \rightarrow$ Bit at index y

We use a bit vector for simplicity, but this technique should generalise to arbitrarily complex classical data

Semantics of gate application:

$$[\![G]\!]_N = G \bigotimes_{|G|}^N ID$$

$$[\![Gq_1 \dots q_n]\!]_N = \text{map}(\lambda \sigma. \text{let } \phi = (\pi^{-1} [\![q_1 \dots q_n]\!]_N \circ [\![G]\!]_N \circ \pi [\![q_1 \dots q_n]\!]_N) \sigma. \phi \text{ in } \{\dots \sigma, \phi\})$$

The classical components aren't too interesting:

$$[\![\text{if}(c)\{B\}]\!] = \text{flatmap}(\lambda\sigma. \text{if}([\![c]\!]\sigma) \text{ then } [\![B]\!]\sigma \text{ else } \sigma)$$

$$[\![\text{while}(c)\{B\}]\!] = \mu X. X \circ [\![\text{if}(c)\{B\}]\!]$$

The basic language: