A survey of quantum programming languages

Jake Trevor

In this talk:

1. QC?
2. PL?
3. QPL?
4. The universe as it stands

QC: A quick recap

Quantum Data:

Quantum data is represented with qubits

Qubits have some state, described by:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \qquad \alpha, \beta \in \mathbb{C}$$

with a normalisation condition:

$$|\alpha|^2 + |\beta|^2 = 1$$

Measurement

When we **measure** a qubit, we can only observe it as being in a basis state (either $|0\rangle$ or $|1\rangle$):

$$\mathrm{P}(|0\rangle) = |\alpha|^2$$
$$\mathrm{P}(|1\rangle) = |\beta|^2$$

Key Point: measurement is destructive

After measurement, the state "collapses" and becomes either $|0\rangle$ or $|1\rangle$.

Multi-qubit States

Multiple qubits are described in a similar way. Here's an example with 2:

$$|\phi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

With a parallel normalisation condition - the sum of the squares of the coefficient is always 1.

So long as that normalisation condition is met, any of these values can be 0.

This means that states such as this are possible:

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Entangled States

Some states can be entirely described in terms of its component qubits:

$$|00\rangle = |0\rangle \otimes |0\rangle$$

We say that these two qubits are 'separable' - because their states can be fully described separately.

But this is not always true. Take this state for example:

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

These qubits are not separable - they are entangled.

Measuring entangled states

When we have an entangled state, measuring a qubit impacts its entangled partners:

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

If we measure the first qubit here, we know the second one must have the same value.

→ Measuring an entangled qubit collapses the entire entangled state!

This is all notational convention for vectors:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

$$|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\\beta\end{bmatrix}$$

For multi-qubit states, the rows just count up in binary:

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix} \qquad \begin{matrix}|00\rangle\\|01\rangle\\|10\rangle\\|11\rangle\end{matrix}$$

Gates

We manipulate qubits using quantum gates - which are really just matrices.

For example, here is the Pauli X gate - the quantum version of NOT:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad X|0\rangle = |1\rangle$$

$$\begin{aligned} X|0\rangle &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= |1\rangle \end{aligned}$$

There's also a controlled version - which Xs the second bit only if the first one is $|1\rangle$:

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$CX|00\rangle = |00\rangle \qquad CX|11\rangle = |10\rangle$$

Circuits

We can combine and compose gates to get more complex behaviour.

Here's tensoring:

$$(X \otimes ID)(|0\rangle \otimes |0\rangle) = |1\rangle \otimes |0\rangle$$

And composition:

$$X(X|0\rangle) = X|1\rangle = |0\rangle$$

These combinations of gates are called circuits.

So:

Quantum computing is just complex-valued linear algebra with probabilities

easy right?

For a more thorough treatment, see Nielson and Chuang (2000)

An aside - Quantum Teleportation

The basic idea: transfer state from one qubit to another

If you want the details, come and speak to me (or Simon) afterwards

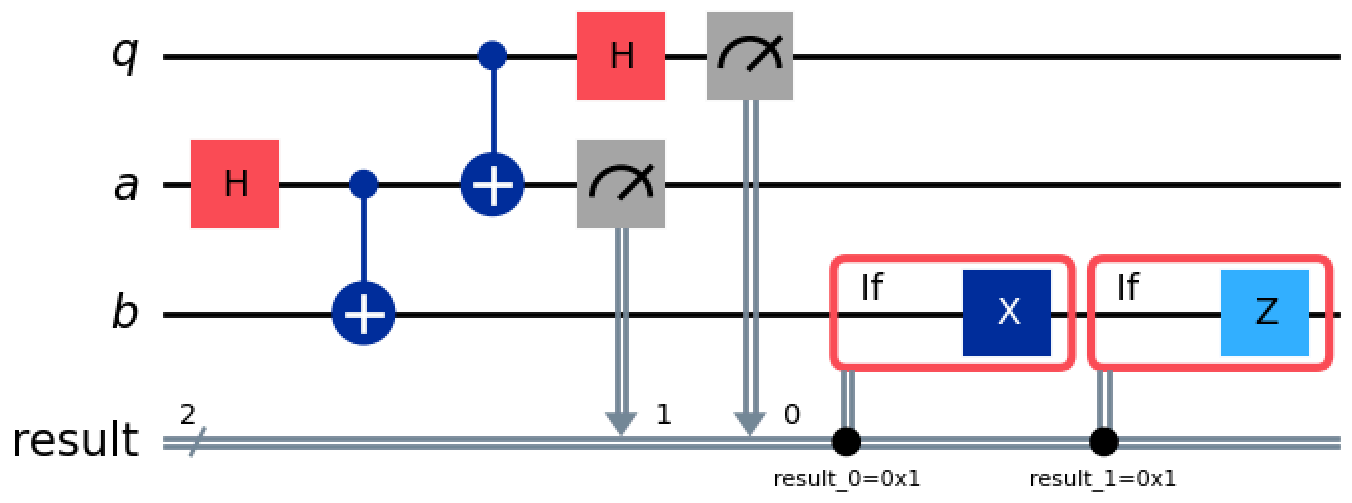
We will use this as our sample program today.

Here it is in a quantum pseudocode:

```
def bell_pair() {
        a, b <- fresh
        H a
        CX a b
        return (a, b)
}

def teleport(q) {
        a, b <- bell_pair()
        CX q a
        H q
        q', a' = M q a
        if (a') X b
        if (q') Z b
        return b
}
```

and as a circuit diagram:



PL: A quick recap

What are PL people trying to do?

There are broadly four goals:

Help programmers …

- write less code
- be sure their code is correct
- make reasoning about the code easier
- automatically optimise resource usage

In a little more detail …

Be Concise

Writing code is expensive and hard work

Requiring programmers to write less code often amounts to making it easier to write a program.

Another advantage:

The #1 source of bugs in code is the existence of code.

```
"An empty program that does nothing is always correct"
```

It's much easier to reason about and check a shorter program.

Guarantee Safety

Much modern PL research is about providing certain kind of safety guarantees.

e.g. memory safety - Rust, Java (& other GC languages, kind-of)

Domain specific safety conditions are quite popular.

- e.g. guarantee that a particular smart contract will not run out of gas

Provide Semantics

We want to be able to give strong mathematical meanings to programs.

Why?

- Prove safety guarantees
- Helps guide implementation
- Validate transformations on code
- Eliminate undefined behaviour (and associated vulnerabilities)

Automate Optimisation

Programmers should be able to write understandable (dare I say 'clean') code without paying a performance cost

Compiler optimisation aims to find automatic ways to translate understandable code into faster/smaller/leaner code.

→ Maybe even in ways the programmer could not have foreseen.

In an ideal world, we want our languages to:

- be concise & clean
- be easy to reason about
- make it hard to write code that's wrong
- make it easy to write code that's right
- run with tiny resource budgets

QPL

-- OR --

Why should we care about the intersection of QC and PL?

QPL is interesting

- QC poses several interesting challenges to the programmer that we would like to mitigate
- It also offers interesting challenges to PL researchers

→ QPL research could help push the boundaries of regular PL research.

QPL is necessary

- We can do some QC without a HL PL $\rightarrow$ circuit diagrams
- However, imagine if you had to write quicksort by drawing out its circuit diagram.

For advanced QC applications, a HL QPL is necessary

QPL is useful:

Using QPLs, we can:

- Write down quantum algorithms, making them:
  - precisely defined
  - easier to reason about
  - easier to talk about
- Perform cost estimations
  - real code analysis often yields tighter bounds than theoretical approaches
- Provide a combined setting for practical, mixed algorithms

Of course most important of all:

You get to say you work in quantum computing

Challenges

The good news: classical control is OK

- Quantum control flow is hard to think about (and implement)
- Fortunately, quantum data with classical control (QDCC) is *equivalent*

→ We can simulate a QTM on a quantum circuit with only classical control in linear time and space

- So we don't need a PL which grapples with a superposition of execution paths.

The bad news: QC is hard anyway

There are four things we need to address:

- non-locality
- linearity
- reversibility
- dynamic lifting

Non-locality

When we entangle qubits, we are forced to reason about them as a group, rather than individually.

Recall this state:

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

If we measure only one of the qubits, we know the value of the other.
In fact, the quantum state as a whole collapses - the other qubit also returns to either $|0\rangle$ or $|1\rangle$ (a 'basis' state).

Linearity (in the resource sense)

Qubits are linear resources

Measurement presents an obvious example of linear behaviour …

… but it's more general than just measurement

The No Cloning Theorem (A well known result):

Given an arbitrary qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, it is impossible to produce a state $|\psi\rangle \otimes |\psi\rangle$.

Reversibility

- A result of the unitary property of quantum gates
- All quantum operations (apart from measurement) are reversible.
    - any construction is also therefore reversible.
- Often we want to take a classical function and generate a quantum implementation (called a 'quantum oracle').

→ Reversibility makes this a fraction more challenging.

- automatic reversion has applications in memory management and optimisation

Dynamic lifting

A very fancy name for a very simple concept:

Conditioning circuit continuations based on measurement results.

```
a, b <- fresh
H a
x = M a
if (x) X b
//  ^ x here is classical!
```

- It requires special semantics
- not all languages support it

Dynamic lifting in quantum teleportation

```
def bell_pair() {
        a, b <- fresh
        H a
        CX a b
        return (a, b)
}

def teleport(q) {
        a, b <- bell_pair()
        CX q a
        H q
        q', a' = M q a
        // Dynamic lifting:
        if (a') X b
        if (q') Z b
        return b
}
```

Quantum programming languages

(The universe as it stands)

The rest of this talk:

I'm going to break things down into two broad sections

1. High level work, usually on programmer-facing languages
2. Low level work, on quantum assemblies, compilers and optimisation

High level languages:

We can broadly break down high-level quantum programming languages into three categories:

1. Research languages
2. Industrial languages
   1. Python libraries

Python

- The fastest way to just 'get started' with QC - just pip install.
- Our own QC course is taught mostly in Qiskit, but at least one alternative also exists (Cirq).

→ they're super fast and easy to learn

Python

- Circuit description style
  → describe which gates to apply to which qubits
- generally target an existing quantum IR (more on these later).
  → a big plus, since they're 'portable'
- Qiskit can run on real IBM QCs

Quantum teleportation in Qiskit:

```
from qiskit import QuantumCircuit

teleport = QuantumCircuit(3, 2)
teleport.h(1)
teleport.cx(1, 2)
teleport.cx(0, 1)
teleport.h(0)
teleport.measure([0, 1],[0, 1])

with teleport.if_test((0, 1)): # cbit 0 = 1?
    teleport.x(2)
with teleport.if_test((1, 1)): # cbit 1 = 1
    teleport.z(2)
```

Somethings to note:

- no functions/subroutines
- all lifetimes are program-static
- no names - only indicies

Python isn't great

- excellent for hacking at a problem and exploring quantum computing ...
- But QC in python will always be second class.
- circuit description style is fast to learn, but ugly for larger programs/protocols
- will never offer everything dedicated languages can
- no strong semantics.
- **Critically, it doesn't do anything interesting from a PL perspective**

Q#

Introduced by Microsoft.
Syntactically very similar to C#.

Interops with CLR - so C#, F#, etc.

One of a small number of truly free-standing QPLs.

Q# is interesting because it is not a circuit description language. It takes a different view of QC:

A quantum computer is just a hardware accelerator (like a GPU).

Q#

- Treats quantum computations just like how we treat classical ones - as lists of instructions.

→ Classical control flow naturally supported

- functors - functions that act on gates.
  The big application: automatic inverse generation.

```
namespace Teleportation {
        open Microsoft.Quantum.Intrinsic;
        open Microsoft.Quantum.Canon;
        open Microsoft.Quantum.Measurement;

        operation bell_pair(a: Qubit, b: Qubit) : Unit {
                H(a);
                CNOT(a, b);
        }

        @EntryPoint()
        operation Teleport () : Result {
                use (q, a, b) = (Qubit(), Qubit(), Qubit());
                X(q);
                bell_pair(a, b);
                CNOT(q, a);
                H(q);
                if (MResetZ(q) == One) { Z(b); }
                if (MResetZ(a) == One) { X(b); }
                return MResetZ(b);
        }
}
```

Some things to note:

- We do get functions!
- But qubit lifetimes are now function-static
  - this is why the bell_pair fn is an awkward mutator, rather than a nice allocator
  - we cannot return qubits from the function they are allocated within
  - No in-editor warnings for functions that violate this lifetime constraint - you have to compile.
    - Give me my squiggles!
- I believe this is to make functors tractable

The downsides:

- Microsoft no longer support Q#.
- Doesn't target the standard QASM (it has it's own)
- no formal semantics

Q# is not going to be the language of quantum computing.

Research languages

Quipper

Quipper is a functional PL developed at Dalhousie.

Embedded in Haskell - install with cabal.

$\rightarrow$ **Still a circuit description language.**

Quipper does not compile to any quantum assembly.
It's not really intended to run on a real quantum computer.

It's designed for semantics research.

Quipper's killer feature:

Core fragments of quipper have been extracted and studied:

- proto-quipper-S: linear typing w/ subtyping & operational semantics
- proto-quipper-M: linear types, operational & categorical semantics
- proto-quipper-D: linear dependent types & fibrational category semantics
- proto-quipper-Dyn: dynamic lifting with operational and categorical semantics

Quipper

The overall goal: unify fragments and get a language with well defined semantics for everything.

Interest seems to have died down of late - last paper was published in 2022

```haskell
bell_pair :: Circ (Qubit, Qubit)
bell_pair = do
        a <- qinit False
        b <- qinit False
        a <- hadamard a
        b <- qnot b ‘controlled‘ a
        return (a, b)

teleport :: Qubit -> Circ Qubit
teleport q = do
        a, b <- bell_pair
        a <- qnot a ‘controlled‘ q
        q <- hadamard q
        (q', a') <- measure (q,a)
        b <- gate_Z b ‘controlled‘ q'
        b <- gate_X b ‘controlled‘ a'
        cdiscard (a, q)
        return b
```

Some things to note:

- This is an embedding in Haskell - so we get functions and dynamic lifetimes
- controlled operations are not primitive
  - rather there is a `controlled` function that can make any primitive operation into a controlled one.

Silq

A functional language with an operational semantics

Silq investigated automatic uncomputation

- automatically disposing of temporary values generated in a computation.
- Silq can only do this for 'qfree' functions
  $\rightarrow$ functions that can be described on a classical computer.
- So Like an automatic oracle generator, but more powerful
  - much more concise
  - less prone to error
- This is all captured automatically, by the type system

Silq

Other nice features:

- linear type system
- classical & quantum control-flow are treated the same syntactically;
    - there is no CNOT gate, as we will see.
- dynamic lifting

Quantum Teleportation in Silq

```
def bell_pair(){
    x := 0 : B;
    y := 0 : B;
    x := H(x);
    if x { y := X(y); }
    return (x, y);
}


def teleport(q: B) {
    (x, y) := bell_pair();
    if q { x := X(x); }

    if measure(q) { y := Z(y); }
    if measure(x) { y := X(y); }

    return y
}
```

Some things to note:

- nice native functions
- with qubits that can leave their defining function!
    - this doesn't really demonstrate the killer feature for silq
    - A lot of the time, you don't need to reset/free/discard qubits when you are done.

Low Level approaches

Low level approaches

Still a wide variety of topics to explore

- cost modelling
- hardware-invariant optimisation
  - e.g. bit width reduction
- hardware-aware optimisation
  - gate selection & approximation

For almost all of these topics, we need a well defined quantum assembly

Quantum IRs/Assemblies

- There are a number of competing assembly languages for quantum computing
- None of these are native to an actual QC though, so they're all really just IRs.

What do we want from a QIR/QASM?

- allow expression of most useful quantum programs
    - including dynamic lifting &
    - classical control flow structures
- support analyses and symbolic execution
    - Or at least have a way to generate/earmark code friendly to such techniques
- support modularisation
    - support for functions and general code reuse
    - Ideally with dynamic allocation

Choosing the right primitives: Parametric gates

- One of the features we (probably) don't want
- Gates like X, Y, and Z correspond to rotations in a higher dimensional space
- X = 180deg rotation about the X-axis
- but there is nothing to stop you rotating by some other amount
- Parametric gates allow you to specify the angle of rotation
- These are bad because they make certain analyses intractable.

ScaffCC/Scaff

- A compiler for a language called 'scaffold' - which is embedded in C.
- An early (2014) attempt at a quantum assembly ("QASM")
- scaffCC's QASM uses a "flat" format, which avoids classical control flow

ScaffCC: Flat codes

- A Flat format assembly is a straight-line code with no branching.
    - the entire program is a single basic block
- all classical control flow is unrolled/evaluated,
- To get this to work, you specify the input size and all classical parameters of the problem you are trying to solve at compile time
- Imagine the paralel for classical codes

Flat codes are intractable for conventional compiler approaches. To get around this they introduced:

- vectorisation (called 'looping')
  - you can loop (or really, apply a gate) over slices of qubit vectors
- functions (call 'heirarchy')
  - you can define reusable "modules" which mutate their input data (no return statement)

→ In other words, they had to un-flatten the code

ScaffCC targets 'QASM-HL'

ScaffCC: Other features

- ScaffCC includes a quantum oracle generator (called "classical-to-quantum-gate" or CTQG in the paper).
- Also explored instrumentation-based approach to compilation
  - Rather than passes, Scaff evaluates classical component of the program
  - quantum gate calls are emitted as output
- automatically determines what classical elements need to be statically evaluated vs integrated by CTQG.

Scaff - In conclusion:

ScaffCC Explored lots of interesting ideas
But ScaffCC's QASM is not the QASM of the future.

- I have no code sample for this because quantum teleportation cannot be represented in Scaffold/QASM-HL no dynamic lifting
- Scaff also uses parametric gates

OQASM

Self described 'quantum circuit intermediate language'

- supported by IBM & partners
- currently, the de-facto standard
- developed alongside qiskit

Quantum teleportation in OpenQASM 3.0

```
OPENQASM 3.0;
include "stdgates.inc";
bit[2] result;
qubit[1] q;
qubit[1] a;
qubit[1] b;
h a[0];
cx a[0], b[0];
cx q[0], a[0];
h q[0];
result[0] = measure q[0];
result[1] = measure a[0];
if (result[0]) {
  x b[0];
}
if (result[1]) {
  z b[0];
}
```

Some things to note:

- though no example here, OQASM has parametric gates
    - in fact, U (x, y, z) is the primitive from which X, Y and Z are derived.
- No functions - only subroutines
    - subroutines cannot allocate qubits.

Conclusions

Key Takeaways:

- QPL is a hot research topic with lots of interesting problems
  - even non-quantum elements ripe for exploration
- QPL suffers from a lack of cohesion
  - work often does not connect together well
  - Particularly bad in terms of academic ideas moving across to industry
- Even basic features are sporadically distributed
  - e.g. classical control flow and dynamic lifting
  - though this is getting better in newer languages

Current state of the art:

- OpenQASM 3.0 is the QASM standard de-jour
  - It has most features you need
  - but it also has some problems
- for language frontends, research presents some promising stuff
  - lots of beautiful, expressive and verified languages
- Despite this, python remains depressingly dominant in industry, because of its portability, flexibility and speed.
- Semantics research is promising, but if we want to see industrial strength languages with formal semantics, a lot more work needs to be done, and quickly

What I didn't talk about:

The section of quantum assemblies used to be much longer; in particular its worth looking at:

- QIR
- QSSA

Other approaches to quantum computing

- e.g. quantum annealing

Other interesting problems:

- Networking
  - many interesting quantum algs. are cryptographic & meant to involve communication.
- Memory management
  - currently all qubit allocation is strictly static
  - In fact it's often even worse than that - tools like Qiskit don't even have a stack; all qubits must be allocated at startup.

(More) Interesting problems

- Cost modelling
  - I briefly mentioned this, but modelling costs for QC is critically important for effective optimisation
  - its also quite complicated, and very hardware dependent.
  - is there space for parametric approaches?
- quantum control flow
  - this is chronically under-explored, and might hide lots of interesting things.

Slides and markdown source can be found here:

https://github.com/JakeTrevor/talks/tree/main/qpl-survey

or use this QR code:

Thank you for your time and attention!