Mutability in Quantum Programming

Alex Rice, `alex.rice@ed.ac.uk`

University of Edinburgh

THE UNIVERSITY *of* EDINBURGH

Imperative vs Functional

A key choice when developing a quantum language is whether to take an imperative or functional style.

Imperative:

```
def u(q1, q2):
  H(q1)
  CX(q1, q2)
```

Functional:

```
def u(q1, q2):
  q3 = H(q1)
  q4, q5 = CX(q3, q2)
  return q4, q5
```

The imperative code is more concise, and can leverage that unitary gates have equal numbers of inputs and outputs. This motivates the search for effective type systems for working with mutable references to qubits.

Linearity

Within the context of pure functional programs, it is well understood that quantum data should be treated linearly. Consider the following program:

```
def u(q1):
  q2 = H(q1)
  q3 = H(q1)
```

Programs like the one above violate the no-cloning theorem and are disallowed by a linear type system.

Of course, mutable references to quantum data should not be treated linearly, but it turns out that such a program cannot be written at all with mutable references to quantum data. Unfortunately, the need for some form of linearity is demonstrated by the following two cases.

Use after measure

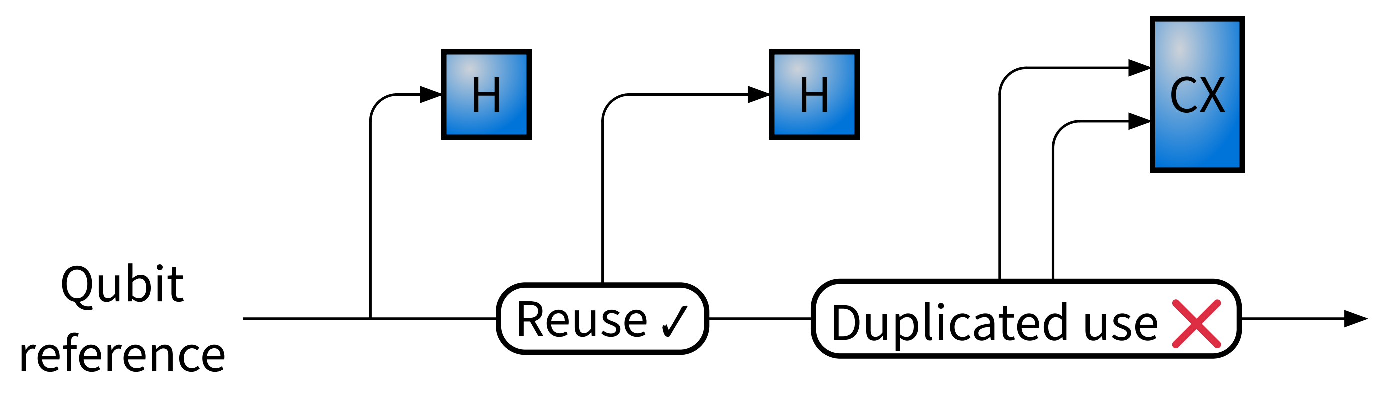
```
def use_after_measure(q):
  x = measure(q)
  H(q)
```

Parallel use

```
def use_twice(q):
  CX(q, q)
```

Both of the above programs can be disallowed by a borrowing based system, as used by Rust. The first can be disallowed by having measurement take quantum data by value instead of reference, and the second is disallowed as it aliases the mutable reference `q`.

Due to this we argue that the "mutability xor aliasing" paradigm is the key to safe quantum programming, combining the ergonomics of mutable references with the safety of functional linear types.



A Prototype Type System

The following toy language is:

- Simple and efficient to typecheck
- Easy to use from a user's perspective
- Ensures safe (linear) use of mutable quantum data

The system enforces single ownership of mutable references, maintaining the invariant that all references point to disjoint data. To achieve this, our judgements take the following form:

$$\Gamma \mid \mathcal{S} \vdash t : A \dashv \mathcal{S}'$$

where Γ is a context, t is a term, and A is a type, which ranges over Qubits $\mathbb{Q}$, (classical) booleans $\mathbb{2}$, and the unit type $\top$.

Instead of removing used variables from the context, we add them to a set $\mathcal{S}$ of used variables, returning a new set $\mathcal{S}'$. Such a system is more flexible, being able to deal with indexing and slicing, as well as producing more meaningful errors (see below).

We now consider the typing rules for the `CX` gate and `measure`:

$$\frac{\Gamma \mid \mathcal{S} \vdash s : \mathbb{Q} \dashv \mathcal{S}'}{\Gamma \mid \mathcal{S} \vdash \texttt{measure}(s) : \mathbb{2} \dashv \mathcal{S}'}$$

In this rule, the variables in s are stored in $\mathcal{S}$, preventing them from being used in subsequent statements, avoiding *use after measure*.

$$\frac{\Gamma \mid \mathcal{S} \vdash s : \mathbb{Q} \dashv \mathcal{S}' \qquad \Gamma \mid \mathcal{S}' \vdash t : \mathbb{Q} \dashv \mathcal{S}''}{\Gamma \mid \mathcal{S} \vdash \texttt{CX}(s, t) \dashv \mathcal{S}}$$

The rule for `CX` uses $\mathcal{S}'$ as the variable set when typechecking t, preventing it using the same variables as s. Further, the output variable set is equal to the input set, allowing the qubit references to be reused in later gates.

To allow for indexing and slicing, the set $\mathcal{S}$ can be changed to store triples (x, n, m) where x is a variable of a quantum register type and $n..m$ is the slice of that register which has been used. Such a set can be efficiently represented as a tree-based set.

Errors

By converting $\mathcal{S}$ from a set to a map from used variables to the text span where they are used, informative error messages can be given to the user:

```
× Variable q has been used in two incompatible locations
    ╭─[54:6]
 53 │ def Bad(q : Q):
 54 │   CX(q, q)
    ·      ┬  ┬
    ·      │  ╰── Second use
    ·      ╰── First use
    ╰────
```

Mutability without aliasing combines ergonomics with safety