Mutability in Quantum Programming Languages

ALEX RICE, The University of Edinburgh, United Kingdom

Quantum data cannot in general be copied or deleted, forcing it to be treated differently to classical data, and often preventing the use of classical programming language abstractions in a quantum setting. Within the setting of functional quantum programming languages, this property is often captured by linear typing, ensuring that any quantum data must be consumed exactly once. However, many mainstream quantum toolkits adopt an imperative style, where variables are understood to represent *mutable references* to quantum data.

In this abstract, we discuss how the linearity translates to this setting, arguing that safe use of quantum data corresponds to preventing shared ownership of mutable references. We further demonstrate that this restricted mutability allows programs to defined concisely, and further facilitates convenient constructions which appear unnatural in the functional setting. Finally, we present a type system for enforcing single ownership of quantum registers which can be sliced and indexed, which can be checked efficiently and produces informative localised errors.

1 Mutability, Immutability, and Aliasing

Many of the most popular programming languages in use today are imperative; variables can be arbitrary reassigned and the data they refer to can be *mutated*. This expressivity can make understanding and reasoning about programs more difficult. Consider the pseudocode example below:

```
c1 = Coord { x: 1, y: 1 }
c2 = c1
...
c1.x += 1
...
print(c2.x)
```

In many languages, this will output "2", even though at first glance it would appear that `c2` has not been mutated after its initialisation. As the size of a program grows, this non-local behaviour can increasingly obscure its functionality.

Functional programming languages respond to this by completely prohibiting the mutation of data, forcing the use of immutable data structures. This far more restrictive paradigm avoids the ambiguity of the above program, as `c2` cannot be mutated in these languages and so will always take its initialisation value.

A middle ground, typically referred to as "mutability xor aliasing", has been explored, notably pushed by the systems language Rust [Matsakis and Klock 2014]. The claim behind this approach is that the difficulty in reasoning about programs such as the example above is not caused by the ability to mutably reference data, but the ability to have multiple mutable references to the same data. "Mutability xor aliasing" enforces that some data can either be the target of a (unique) mutable reference, or be referred to by any number of immutable references, but not both of these.

Author's Contact Information: Alex Rice, The University of Edinburgh, United Kingdom, alex.rice@ed.ac.uk.

A large downside of "mutability xor aliasing" is that it is far more complex to enforce than strict immutability. In Rust, this is enforced by the borrow checker, a flow-sensitive non-lexical typechecking phase. This borrow checker if often the main obstacle beginners face when learning Rust; programs must often be annotated with abstract lifetime variables, and many programs cannot be verified by the borrow checker, despite not causing aliased mutable references. Beginners often end up "fighting with the borrow checker" when trying to write programs.

Recently, *mutable value semantics* [Racordon et al. 2022] has been proposed as an alternative to borrow checking. In this simpler (and more restrictive) system, one only needs to check that mutable references are non-aliasing at function boundaries. At any other point in the program, aliasing a mutable reference is to be understood as a deep copy. Therefore, if we were to reason about the above program in this system, the line `c2 = c1` would make a copy of `c1`, ensuring that `c1` and `c2` point to distinct data (and causing the output of the program to be "1").

2 Mutability for Quantum Programming

Many quantum toolkits, for example Qiskit [Javadi-Abhari et al. 2024] or Q# [Svore et al. 2018], already take an imperative style, where each primitive gate is realised by a side-effective operation; the functions have computation behaviour other than that given by their outputs.

```
def u(q1: qubit, q2: qubit):

  H(q1)
  CX(q1, q2)
```

(a) Imperative style

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

(b) Functional style

Fig. 1. A simple quantum program in an imperative and functional styles

In fig. 1, we write a pseudocode representation of a simple circuit in both the imperative and functional styles.

In the functional style, we can understand how quantum data may be used through linear typing (e.g. [Wadler 1990]). Each identifier representing a qubit must be used in exactly one location. These identifiers represent the actual data of a qubit, rather than a reference to the location of a qubit, and their linear use corresponds directly to the properties of quantum information; the inability to copy or discard it.

The imperative style is more concise, as it removes the need to assign the outputs of each gate to new fresh variables and can exploit that all unitaries will have an equal number of input and output qubits. Furthermore, mutable variables allow new constructions. Consider the (imperative) program below on qubits a, b, and c, where `qif` is a quantum if (a subprogram controlled on a qubit).

```
qif a then Swap(b, c)
```

Here, the qubits b and c can be "captured" from the outer context. Attempting to achieve similar in the functional setting leads to ambiguity (as it is unclear whether operations occur before the quantum if starts).

It is however less clear how properties of quantum information should manifest in the imperative setting. The identifiers in this version are no longer linear: `q1` is used twice. In some sense, using the variables mutably has enforced some of the properties of linearity, it is not possible to reuse the input to the Hadamard gate, but we still must be careful to disallow programs such as `CX(q1, q1)`. A simple local check that the inputs to each primitive are distinct may not be sufficient; a user may try to use the function `u` with `q1` and `q2` being the same qubit.

Due to this we argue that the key to safe quantum programming is preventing the aliasing of mutable variables. By embracing the "mutability xor aliasing" paradigm in quantum programming languages, one can combine the ergonomics of mutable references with the safety of functional linear types.

3 A Type System for Mutable Quantum Programs

We now demonstrate a method for implementing a type system for single ownership mutable variables. Such a system should ideally be:

- Simple and efficient to typecheck,
- Easy to use and understand from the users perspective,
- Ensures the safe (linear) use of quantum data.

We demonstrate how such a system can be implemented by sketching the type system of a toy language below. We will largely follow the approach of [Racordon et al. 2022], by maintaining the invariant that the available (mutable) references at any program point reference disjoint data, and checking this invariant is maintained at function boundaries.

The only types in our toy language will be the unit type $\top$, the classical boolean type `bool` and quantum register types `qn` for each natural number n. A judgement on terms takes the following form:

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

where Γ is a context, t is a term, A is a type, and $\mathcal{S}$ (and $\mathcal{S}'$) are sets of triples (v, l, u), where $v : qn \in \Gamma$ and $0 \leq l \leq u < n$ forms a valid slice of the quantum register v. Intuitively, $\mathcal{S}$ records which parts of Γ may not be used in the term t. For such triples, We write $(v, l, u)|(v', l', u')$ when they are disjoint, i.e. $v \neq v'$ or the ranges $l..u$ and $l'..u'$ are disjoint, and similarly define $\mathcal{S}|(v, l, u)$ when (v, l, u) is disjoint to all triples in $\mathcal{S}$.

Instead of fully specifying a type theory, we explore a few key cases. We first consider the operations on quantum register types. For variables, and slices of variables we have:

$$\frac{v : qn \in \Gamma \qquad \mathcal{S}|(v, 0, n)}{\Gamma \mid \mathcal{S} \vdash v : qn \dashv \mathcal{S} \cup \{(v, 0, n)\}} \qquad \frac{v : qn \in \Gamma \qquad \mathcal{S}|(v, l, u)}{\Gamma \mid \mathcal{S} \vdash v[l..u] : q(u - l) \dashv \mathcal{S} \cup \{(v, l, u)\}}$$

These rules simply add the used variables to the set $\mathcal{S}$. We can further combine quantum registers with a concatenation operation:

$$\frac{\Gamma \mid \mathcal{S} \vdash s : qm \dashv \mathcal{S}' \qquad \Gamma \mid \mathcal{S}' \vdash t : qn \dashv \mathcal{S}''}{\Gamma \mid \mathcal{S} \vdash s, t : q(m+n) \dashv \mathcal{S}''}$$

Here, the impact of the variable sets can be seen. The term v, v would not be typeable, as v would necessarily have been added to the set $\mathcal{S}'$, preventing its use in the second term.

As an example of a unitary operation, the controlled not operation can be typed as follows:

$$\frac{\Gamma \mid \mathcal{S} \vdash s : q2 \dashv \mathcal{S}'}{\Gamma \mid \mathcal{S} \vdash \texttt{CX}(s) : \top \dashv \mathcal{S}}$$

The term `CX(a, a)` being not well-typed follows from the inability to type the concatenation `a, a`. Crucially, this rule ignores the output variable set $\mathcal{S}'$, allowing the qubits used in the gate to be reused in later gates. Conversely, a measurement operations should not ignore this set, as the qubit is consumed by the operation:

$$\frac{\Gamma \mid \mathcal{S} \vdash s : q1 \dashv \mathcal{S}'}{\Gamma \mid \mathcal{S} \vdash \texttt{measure}\ s : \texttt{bool} \dashv \mathcal{S}'}$$

Finally the quantum if with variable capture can take the following form:

$$\frac{\Gamma \mid \mathcal{S} \vdash s : q1 \dashv \mathcal{S}' \qquad \Gamma \mid \mathcal{S}' \vdash t : \top \dashv \mathcal{S}'}{\Gamma \mid \mathcal{S} \vdash \texttt{qif}\ s\ \texttt{then}\ t : \top \dashv \mathcal{S}}$$

Note that this typing rule prevents variables in s being reused in the body t.

The set $\mathcal{S}$ can be conveniently (and efficiently) represented by tree-based set, using a lexicographic ordering on the triples (v, l, u). We maintain the invariant that each triple in the set is disjoint, making it easy to check for clashes when inserting a new variable. To produce informative error messages, this structure can be replaced by a tree-based map which sends each triple to the position in the code where that triple was used. This location can then be displayed to the use in the case a clash occurs.

4 Related Work

We are aware that similar ideas have been explored in Quantinuum's quantum programming language Guppy [Koch et al. 2025], but do not know how their approach compares to the one presented here.

References

Ali Javadi-Abhari et al.. June 18, 2024. *Quantum Computing with Qiskit.* (June 18, 2024). arXiv: 2405.08810 [quant-ph]. Retrieved Sept. 2, 2024 from http://arxiv.org/abs/2405.08810. Pre-published.

Mark Koch, Agustín Borgna, Craig Roy, Alan Lawrence, Kartik Singhal, Seyon Sivarajah, and Ross Duncan. Oct. 15, 2025. *Imperative Quantum Programming with Ownership and Borrowing in Guppy.* (Oct. 15, 2025). arXiv: 2510.13082 [cs]. Retrieved Oct. 31, 2025 from http://arxiv.org/abs/2510.13082. Pre-published.

Nicholas D. Matsakis and Felix S. Klock. Oct. 18, 2014. "The Rust Language." In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (HILT '14). Association for Computing Machinery, New York, NY, USA, (Oct. 18, 2014), 103–104. ISBN: 978-1-4503-3217-0. doi:10.1145/2663171.2663188.

Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. "Implementation Strategies for Mutable Value Semantics." *The Journal of Object Technology*, 21, 2, 2:1. doi:10.5381/jot.2022.21.2.a2.

Krysta Svore et al.. Feb. 24, 2018. "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL." In: *Proceedings of the Real World Domain Specific Languages Workshop 2018* (RWDSL2018). Association for Computing Machinery, New York, NY, USA, (Feb. 24, 2018), 1–10. ISBN: 978-1-4503-6355-6. doi:10.1145/3183895.3183901.

Philip Wadler. 1990. "Linear Types Can Change the World!" In: *Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*. Ed. by Manfred Broy and Cliff B. Jones. North-Holland, 561.