Imperative Quantum Programming with Ownership and Borrowing in Guppy

MARK KOCH, AGUSTÍN BORGNA, CRAIG ROY, ALAN LAWRENCE, KARTIK SINGHAL, SEYON SIVARAJAH, and ROSS DUNCAN, Quantinuum, United Kingdom

Linear types enforce no-cloning and no-deleting theorems in functional quantum programming. However, in imperative quantum programming, they have not gained widespread adoption. This work aims to develop a quantum type system that combines ergonomic linear typing with imperative semantics and maintains safety guarantees. All ideas presented here have been implemented in Quantinuum's Guppy programming language.

1 INTRODUCTION

Given the high cost and long queue times associated with quantum hardware, rapid debugging and iteration of quantum programs is typically not feasible. Consequently, we rely on static techniques such as type systems to identify bugs *before* programs are submitted to quantum devices. One specific example we focus on is the application of *linear types* [18] to statically enforce the no-cloning and no-deleting theorems of quantum mechanics. The idea of utilizing linear typing in this context was pioneered by Selinger and Valiron [13] and has since been employed as the foundation for various functional quantum programming languages. Unfortunately, the same is not true in the setting of *imperative quantum programming*, where so far, linear types have not gained much traction (with the notable exception of Silq [1]). Previous attempts essentially boil down to equipping qubits with value semantics and treating quantum gates as functional primitives, effectively replicating the functional approach of Selinger and Valiron [13] with more verbose syntax (see Sec. 4.1 for a discussion). This approach fails to capture the semantics of languages like Q# [14, 17] or OpenQASM 3 [4], where qubits are treated as *opaque pointers*, and quantum operations are implemented via *side-effectful functions* acting on them.

In this work, we aim to develop a quantum type system that offers ergonomic linear typing while simultaneously capturing imperative semantics and maintaining the same safety guarantees. To achieve this, we draw inspiration from classical systems-level programming, where similar substructural type systems have found significant success over the past decade. The most prominent example is Rust [11, 12]. Unlike languages that enforce strict no-cloning rules, these languages employ linear or affine types to provide static memory safety by reasoning about pointers and aliasing. In particular, they frame the type system in terms of *ownership* [3], where certain actions on a resource (such as writing to a memory location) necessitate unique ownership of that resource. Other parties may temporarily *borrow* the resource, but must eventually return it to its original owner. We argue that this ownership and borrowing discipline is a perfect fit for imperative quantum programming and, in fact, represents the optimal approach to implementing linear typing ideas from Selinger and Valiron [13] in an imperative language.

Concretely, we develop a type system and associated ownership discipline that statically enforces the following safety properties: (1) qubits cannot be used after they are destructively measured or discarded, (2) multi-qubit gates cannot act on the same qubit, and (3) it is impossible to implicitly discard or leak qubits. We achieve this by differentiating between operations that consume qubits (such as destructive measurement or discarding) and ones that merely act on them (such as applying quantum gates). While a qubit cannot be used after being consumed by the former, it remains perfectly valid to reuse it after the latter. By enforcing this distinction in the type system, we make it easy for both the user and the type checker to reason about and infer which qubits are alive

Authors' address: Mark Koch, mark.koch@quantinuum.com; Agustín Borgna; Craig Roy; Alan Lawrence; Kartik Singhal; Seyon Sivarajah; Ross Duncan, Quantinuum, Cambridge, United Kingdom.

at various points in the program. Ownership, in turn, is used to constrain the places in which measurements of qubits are allowed and, on the other hand, to enforce that qubits are not implicitly discarded or leaked. Furthermore, we demonstrate that our ownership discipline enables us to translate imperative programs (where quantum gates resemble side-effectful functions) into pure functional programs better suited for quantum optimization (see Sec. 3).

All ideas discussed in this abstract have been implemented as an update to the Guppy quantum programming language [10]. In Sec. 2, we provide an overview of Guppy and its ownership system by way of examples. In Sec. 3, we discuss how imperative Guppy programs can be translated into a functional intermediate form. Finally, in Sec. 4, we discuss related work and outline future directions. While we explain our ideas in the context and syntax of Guppy, the ideas discussed are also applicable to other imperative quantum programming languages.

2 IMPERATIVE GUPPY BY EXAMPLE

Guppy is a domain-specific quantum programming language embedded into Python, with a particular focus on dynamic quantum programs where classical computations (conditioned on mid-circuit measurement results) can affect the execution flow of the remaining program.[1] Guppy enables users to write these programs using Python's native syntax for classical operations and control flow. However, unlike Python, Guppy is compiled ahead of time rather than being interpreted. It features a static type system that includes Python's standard data types (e.g. `int`, `float`, `list`, etc.) along with a dedicated `qubit` type to represent qubits. See Koch et al. [10] for more on Guppy.

2.1 Ownership & Borrowing

Each qubit in Guppy has an *owner*, uniquely identifying the scope that can consume it by destructively measuring or discarding it. Initially, every qubit is owned by the function scope that allocated it. Ownership may change during program execution, but there can be only one owner at any given point in time. For instance, consider the following function:

```
1  def bell() -> (qubit, qubit):
2      q1, q2 = qubit(), qubit()  # Allocated qubits owned by this scope
3      h(q1)
4      cx(q1, q2)
5      return q1, q2  # Transfers ownership to the caller
```

After the allocation in line 2, both qubits are owned by the `bell` function. The call to `h` in line 3 applies a Hadamard gate to `q1`. Notably, `h` does not acquire ownership of the qubit; it merely *borrows* it temporarily. Therefore, we know that `h` cannot destruct the qubit via measurement or discarding, so we can be sure that `q1` is still alive once `h` returns. Thus, `q1` can be borrowed again when we call `cx` in the next line. We must ensure that only a single borrow of a qubit is active at any given time. For instance, the following line would be rejected by the borrow checker:

```
    cx(q1, q1)  # Error: q1 already borrowed
```

To enable this check, we also need to prevent qubits from being aliased, which we achieve via standard linearity checking on owned qubits. Finally, owned qubits can be returned as in line 5 of `bell` function, which *transfers* their ownership into the calling scope:

```
def main():
    q1, q2 = bell()
    # Main now owns q1 and q2, so we can measure them
    assert measure(q1) == measure(q2)
```

[1] Guppy is open source and available at https://github.com/CQCL/guppylang.

As `main` now has ownership of the qubits, it has the right to deallocate them by measuring. Naturally, qubits can no longer be used after they have been deallocated:

```
h(q1)  # Error: q1 already consumed
```

2.2 Borrow by Default

Unlike in Rust, borrows in our type system are *not* first-class. It is not possible to return a borrowed value or store it in a data structure. Consequently, borrowing more closely resembles a calling convention rather than a first-class type. We chose this design to strike a balance between expressiveness (achieving our main goal of making imperative linear typing more ergonomic) and cognitive load (not requiring our Python user base to deal with explicit lifetime parameters).[2]

When writing a function, our default assumption is that qubit arguments are only borrowed from the caller and no ownership transfer occurs. For example, consider the following function:

```
1 def foo(q: qubit):
2     # Argument q is borrowed, not owned
3     h(q)
4     z(q)
```

The function argument `q` above is not owned by `foo`; it is merely borrowed from the caller. However, `foo` is still allowed to temporarily lend this borrowed qubit to someone else – e.g., applying `h` in line 3 – which is called *reborrowing*; analogous to the notion of reborrowing of mutable references in Rust. This works because all reborrows expire by the time `foo` returns, ensuring that the qubit can be safely returned to the caller. Conversely, this would not be possible if the qubit had been consumed in the interim. As discussed before, consuming operations like destructive measurement require ownership and are therefore prohibited for borrowed values:

```
measure(q)  # Error: Cannot measure qubit since it is not owned
```

If users wish to measure a passed qubit, they must explicitly assume ownership by adding an `@owned` annotation to the argument:[3]

```
def bar(q: qubit @owned) -> bool:
    # Ownership transferred from the caller
    return measure(q)
```

In turn, this means that `bar` is now also a consuming operation, and any passed qubit can no longer be used afterwards. Also, note that the signature of the `measure` function features the same annotation: `measure(q: qubit @owned) -> bool`.

This design facilitates both the user and the type checker in reasoning about the lifetime of qubits. The function signature explicitly indicates whether a qubit remains usable after the function returns. In particular, by making borrowing the default, we encourage users to think about and explicitly annotate whether a subroutine deallocates an input qubit.

2.3 Avoiding Qubit Leaks

So far, we have mostly focussed on ensuring the safety of qubits by preventing their reuse after deallocation. Conversely, there is also a safety concern if we forget to deallocate qubits after they are no longer in use, resulting in *qubit leaks*. Given the scarcity of qubits, this should be strictly avoided. Programming languages like Rust address this issue by automatically discarding values from the heap once their owners go out of scope. However, in the quantum context, we aim to avoid implicit discarding of qubits since dropped qubits often require post-selection or uncomputation to

[2] This design aligns with Graydon Hoare's initial vision for Rust, which was later abandoned in favour of explicit lifetimes [6].
[3] `@owned` syntax is inspired by Python's decorator notation and OCaml's mode syntax [16].

ensure algorithm correctness. Consequently, unmeasured temporary qubits typically point to an implementation bug. Therefore, we emit a compiler error whenever a qubit's owner goes out of scope without having measured the qubit or transferred its ownership. In such cases, we prompt the user to make their intention explicit. Consider the following example:

```
def baz(q: qubit):
    tmp = qubit()  # Error: Allocated qubit is not consumed
    cx(q, tmp)
```

Here, the allocated qubit `tmp` owned by `baz` would be leaked because there would be no remaining reference to it once `baz` returns. To fix this, we could either explicitly measure `tmp` or transfer ownership of the qubit by returning `tmp` to the caller. Additionally, note that this restriction does not apply to the qubit `q` since it is only borrowed, so the responsibility for deallocating it lies with its original owner. In other words, while we enforce linear typing for owned qubits, we only use affine typing for borrowed ones.

2.4 Interaction with Other Language Features

The ownership discipline laid out so far harmoniously interacts with the other language features of Guppy. For example, our linearity checking logic extends to arbitrary control flow (`if`, `for`, `while`, etc.) through dataflow analysis. Furthermore, users can define custom struct types that can simultaneously contain both qubits and classical data. Ownership and borrowing of these structs are tracked field-wise, so users can measure or transfer ownership of qubits in specific fields while the others remain valid.[4] The same principle applies to collection types like lists or statically sized arrays:

```
class MyStruct:
    q: qubit
    qs: array[qubit, 42]
    x: int
```

```
def example(s: MyStruct):
    for other in s.qs:
        cx(s.q, other)
    # Classical fields are not linear:
    s.x += 2 * s.x
```

The `for` loop reborrows each array element in `s.qs` into a temporary variable `other` that is active within the scope of the loop body. Note that other struct fields like `s.q` are not affected by reborrowing, so we are still allowed to borrow the qubit within the loop body when applying `cx`. In contrast, classical struct fields like `s.x` are unaffected by linearity and can be freely copied or discarded.

3 FUNCTIONAL TRANSLATION

One notable aspect of linear type systems, as observed by Wadler [18], is that they make it easier to translate pure programs into imperative ones. Here, we utilize the converse observation: imperative programs with a sufficiently robust ownership discipline can be compiled into a pure functional equivalent [2]. While translations of this form have been implemented in the past (the most advanced being the Aeneas [5] framework capable of translating a large subset of Rust), support for control flow in these systems remains limited. Fortunately, we benefit from the fact that our borrowing system is simpler than that in Rust and other languages, enabling us to perform a *complete* functional translation. Concretely, we compile imperative Guppy programs into HUGR [9], a functional intermediate representation based on dataflow graphs designed for optimization of dynamic quantum-classical programs. The key idea is that side-effectful functions that borrow

[4]This is analogous to the notion of *partial moves* in Rust.

qubits can be translated into pure functions that accept and *return* these qubits. Consider the following example:

```
def foo_imperative(q: qubit):
    h(q)
    z(q)
```

```
def foo_pure(q: qubit) -> qubit:
    q = h(q)
    q = z(q)
    return q
```

The correctness of this translation is ensured by the borrow checker. Since borrows of qubits are unique, it is safe to replace them with qubits passed by value without violating the no-cloning theorem. Similarly, borrowed qubits cannot be measured, so it is always safe to translate the expiration of a borrow into an explicit return of the borrowed qubit. Furthermore, since we do not permit borrowed qubits to be stored in data structures, we are certain that they will remain bound to a name within the scope of the function, making it feasible to track them during the translation.

4 DISCUSSION

4.1 Related Work

The imperative quantum programming style is employed in various quantum languages, such as Q# [17], OpenQASM 3 [4], Catalyst [7], and Silq [1]. Among those, Silq is the only language that provides static safety guarantees via linear typing. While Silq offers imperative control flow and looping, its quantum operations are functional primitives, resulting in code that closely resembles the `foo_pure` example in Sec. 3, involving threading of linear qubits, rather than the more ergonomic `foo_imperative` variant. This was also the case in an earlier version of Guppy [10].

Also inspired by Rust, Singhal et al. [15] proposed a solution for statically preventing qubit cloning in $\lambda_{Q\#}$ using controlled aliasing of qubits. However, their approach was restricted to an idealized functional version of Q# and, adhering to Q# semantics, did not account for ownership transfer.

4.2 Future Work

So far, we have only used ownership and borrowing to constrain the places in which destructive measurement of qubits is permitted. An interesting generalisation would be to consider more fine-grained versions of borrowing that further restrict the kind of quantum operations that can be applied to qubits. For example, a kind of borrow that only permits operations that can be described classically (i.e. ones that do not introduce superposition or phase effects) could enable automatic uncomputation in the style of Silq [1].

Furthermore, to increase confidence in our type system, we want to define a core language fragment of Guppy and formally prove the safety claims made in this abstract. In that context, the λ_{Rust} fragment of the RustBelt project [8] and its type system could serve as an interesting starting point. Finally, we want to prove soundness of the functional translation sketched in Sec. 3, drawing from the work done in Aeneas [5].

REFERENCES

[1] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proc. 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007 https://files.sri.inf.ethz.ch/website/papers/pldi20-silq.pdf

[2] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/1411204.1411235 https://www.chargueraud.org/research/2008/capa/capabilities_icfp_08.pdf

[3] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. ACM, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

[4] Andrew W. Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (September 2022), 50 pages. https://doi.org/10.1145/3505636

[5] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. https://doi.org/10.1145/3547647

[6] Graydon Hoare. 2023. *The Rust I Wanted Had No Future.* Dreamwidth. https://graydon2.dreamwidth.org/307291.html

[7] Josh Izaac. 2023. *Introducing Catalyst: quantum just-in-time compilation.* Pennylane. https://pennylane.ai/blog/2023/03/introducing-catalyst-quantum-just-in-time-compilation/

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[9] Mark Koch, Agustin Borgna, Seyon Sivarajah, Alan Lawrence, Alec Edgington, Douglas Wilson, Craig Roy, Luca Mondada, Lukas Heidemann, and Ross Duncan. 2024. *Hugr: A quantum-classical intermediate representation.* Informal Proceedings of the Sixth International Workshop on Programming Languages for Quantum Computing (IWQC '24). https://quantum-compilers.github.io/iwqc2024/papers/IWQC2024_paper_14.pdf

[10] Mark Koch, Alan Lawrence, Kartik Singhal, Seyon Sivarajah, and Ross Duncan. 2024. *Guppy: Pythonic quantum-classical programming.* Informal Proceedings of the Fourth International Workshop on Programming Languages for Quantum Computing (PLanQC '24). https://ks.cs.uchicago.edu/publication/guppy-planqc/

[11] Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust Language. In *Proc. ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[12] The Rust Team. 2024. *Rust Programming Language.* https://www.rust-lang.org/

[13] Peter Selinger and Benoît Valiron. 2006. A Lambda Calculus for Quantum Computation with Classical Control. *Mathematical Structures in Computer Science* 16, 3 (2006), 527–552. https://doi.org/10.1017/S0960129506005238 https://www.mscs.dal.ca/~selinger/papers/papers/qlambda-mscs.pdf

[14] Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. 2023. Q# as a Quantum Algorithmic Language. In *Proceedings 19th International Conference on Quantum Physics and Logic, Wolfson College, Oxford, UK, 27 June–1 July 2022 (EPTCS, Vol. 394)*, Stefano Gogioso and Matty Hoban (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 170–191. https://doi.org/10.4204/EPTCS.394.10

[15] Kartik Singhal, Sarah Marshall, Kesha Hietala, and Robert Rand. 2021. *Toward a Type-Theoretic Interpretation of Q# and Statically Enforcing the No-Cloning Theorem.* Informal Proceedings of the Second International Workshop on Programming Languages for Quantum Computing (PLanQC '21). https://ks.cs.uchicago.edu/publication/tttiq/

[16] Max Slater. 2023. *Oxidizing OCaml: Locality.* Jane Street Tech Blog. https://blog.janestreet.com/oxidizing-ocaml-locality/

[17] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher E. Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL. In *Proc. Real World Domain Specific Languages Workshop 2018 (RWDSL '18)*. ACM, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901 arXiv:1803.00652

[18] Philip Wadler. 1990. Linear Types can Change the World!. In *Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, Sea of Galilee, Israel, 561. https://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linear-types