Toward Formalizing the Q# Programming Language

Sarah Marshall [1] Kartik Singhal [2] Kesha Hietala [3] Robert Rand [2]

[1]Microsoft Quantum [2]University of Chicago [3]University of Maryland

What is Q#?

Q# [1] is a **hybrid quantum-classical programming language** from Microsoft. It supports execution on existing quantum hardware using Azure Quantum, but is also designed for future large-scale, fault-tolerant quantum computing. Its key features include:

- **Classical computation** and **control flow** can be freely mixed with quantum gates and measurements. For example, see the `if` statements and `repeat` loop below.
- Classical functions in Q# are **pure**, while quantum operations are **effectful**.
- **Higher-order** operations and functions are supported.
- Unitary operations can be automatically converted to their **adjoint** and **controlled** versions.
- Qubits are **opaque types** that act as references to **logical qubits**.

```
operation Entangle(q1 : Qubit, q2 : Qubit) : Unit is Adj {
    H(q1);
    CNOT(q1, q2);
}

Entangle(register, target);
Adjoint Entangle(message, register);

if MResetZ(message)  == One { Z(target); }
if MResetZ(register) == One { X(target); }
```

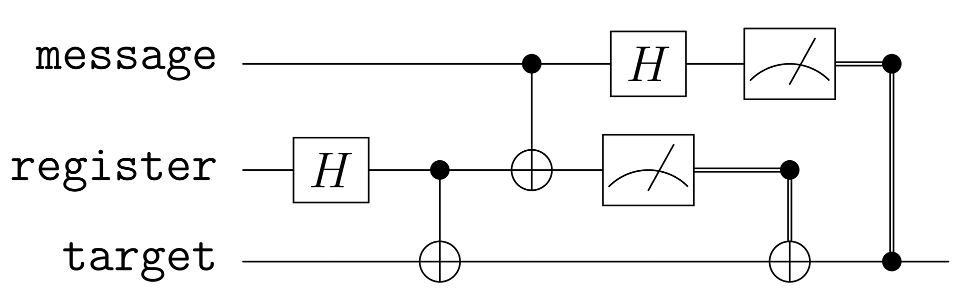

Figure 1. Quantum teleportation in Q# and an equivalent circuit.

```
mutable result = Zero;
mutable iterations = 0;

repeat {
    ApplyCircuit(register);
    set result = MResetZ(register[0]);
    set iterations += 1;
} until result == Zero or iterations > limit;
```

Listing 1. Non-deterministic circuit using a repeat-until-success loop [2] in Q#.

$\lambda_{Q\#}$: A core calculus for Q#

We aim to provide a **formal specification and semantics** for Q#. Following the approach of language formalization efforts like Standard ML [3], we begin by defining a small, well-typed **core language**, $\lambda_{Q\#}$, which captures the essential aspects of Q#. We will then prove properties about $\lambda_{Q\#}$ and define an **elaboration relation** from the full Q# language to $\lambda_{Q\#}$.

Grammar

τ ::=		Types
|	$\mathbf{qbit}$	qbit
|	$\mathbf{qref}$	qref
|	$\mathbf{arr}\,(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$
|	$\mathbf{cmd}\,(\tau)$	τ cmd
|	$\mathbf{unit}$	unit

m ::=		Commands
|	$\mathbf{ret}\,(e)$	ret e
|	$\mathbf{bnd}\,(e;x.m)$	bind $x \leftarrow e; m$
|	$\mathbf{dcl}\,(q.m)$	dcl q in m
|	$\mathbf{gateapr}\,(e;U)$	$U\,(e)$
|	$\mathbf{ctrlapr}\,(e_1;e_2;U)$	Controlled $U\,(e_1,e_2)$
|	$\mathbf{gateap}\,[q](U)$	opaque gate
|	$\mathbf{ctrlap}\,[q_1,q_2](U)$	opaque ctrl'd gate

e ::=		Expressions
|	x	variable
|	$\mathbf{let}\,(e_1;x.e_2)$	let x be e_1 in e_2
|	$\mathbf{lam}\,\{\tau\}(x.e)$	$\lambda(x:\tau)e$
|	$\mathbf{ap}\,(e_1;e_2)$	$e_1(e_2)$
|	$\mathbf{cmd}\,(m)$	cmd m, encapsulation
|	$\mathbf{qloc}\,[q]$	$\&q$, qubit location
|	$\mathbf{triv}$	(), unit constant

Statics

(Typing rules for gate application)

cmd-GateApRef

$$\frac{\Gamma \vdash_{\Sigma} e : \mathbf{qref}}{\Gamma \vdash_{\Sigma} \mathbf{gateapr}\,(e;U) : \mathbf{unit}}$$

cmd-GateAp

$$\frac{}{\Gamma \vdash_{\Sigma,q\sim\mathbf{qbit}} \mathbf{gateap}\,[q](U) : \mathbf{unit}}$$

Dynamics

(Evaluation rule for gate application with qubit reference)

trSm-GateApRefInstr

$$\frac{}{\mathbf{gateapr}\,(\mathbf{qloc}\,[q];U) \underset{\Sigma,q\sim\mathbf{qbit}}{\longmapsto} \mathbf{gateap}\,[q](U)}$$

Benefits of formalization

Like many programming languages, Q# was designed without a precise formal specification, which can lead to ambiguity in its interpretation. A formal specification allows us to prove properties about Q#'s type system and provides a foundation for the development of new features, like the one discussed below.

Statically preventing cloning

Q# supports **aliasing qubit references**. Some programs are accepted by the compiler that will fail at runtime. For example, applying `CNOT(q1, q2)` below will fail because `q1` and `q2` both refer to the same physical qubit.

```
use q1 = Qubit();
let q2 = q1;
CNOT(q1, q2); // Bad!
```

(a) An invalid use of a qubit alias.

dcl q in
let q_1 be $\&q$ in
let q_2 be q_1 in
Controlled $\mathbf{X}\,(q_1,q_2)$

(b) The elaboration of (a) to $\lambda_{Q\#}$.

However, many Q# programs leverage qubit aliasing with arrays to succinctly express quantum programs, like the example below, which applies `CNOT` to adjacent qubit pairs $(q_1,q_2),(q_2,q_3),\ldots,(q_{n-1},q_n)$.

```
operation ApplyCNOTChain(qs : Qubit[]) : Unit is Adj + Ctl {
    ApplyToEachCA(CNOT, Zipped(Most(qs), Rest(qs)));
}
```

Q# currently **cannot statically prevent cloning**. We are working on a solution to this issue as part of our formal specification, taking inspiration from λ_{Rust} [4].

Allowing correct programs like `ApplyCNOTChain` while rejecting incorrect programs like `CNOT(q1, q2)` is difficult. In `ApplyCNOTChain`, it is not as obvious that the arguments to `CNOT` are distinct. The type system must infer that `Most(qs)[i]` and `Rest(qs)[i]` are distinct qubits for all indices `i`.

References

[1] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proc. RWDLS '18*, pages 7:1–7:10. ACM, 2018, arXiv:1803.00652.

[2] Adam Paetznick and Krysta M. Svore. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries, 2014, arXiv:1311.1074.

[3] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, pages 341–387. MIT Press, Cambridge, MA, 2000. URL `https://www.cs.cmu.edu/~rwh/papers/ttisml/ttisml.pdf`.

[4] Ralf Jung. *Understanding and Evolving the Rust Programming Language*. PhD thesis, Saarland University, 2020. URL `https://people.mpi-sws.org/~jung/thesis.html`.