Q# as a Quantum Algorithmic Language

Kartik Singhal\textsuperscript{1}  Kesha Hietala\textsuperscript{2}  Sarah Marshall\textsuperscript{3}  Robert Rand\textsuperscript{1}

\textsuperscript{1}University of Chicago  \textsuperscript{2}University of Maryland  \textsuperscript{3}Microsoft Quantum

Quantum Physics and Logic 2022
The Q# programming language (Microsoft, 2017–)

F#-like DSL in the skin of C#-like syntax.\(^a\)

Quantum computer as a co-processor to a classical machine (QRAM).

Clean separation between classical (function) and quantum (operation) callables. ALGOL-like.

Quantum operations are a monadic sequence of instructions; computation by side effects.

Immutable by default, metaprogramming support (adjoint & controlled operations).

\(^a\)F# & Q# – A tale of two languages [Azariah 2018]
Teleportation in Q#

```qsharp
open Microsoft.Quantum.Intrinsic; // for H, X, Z, CNOT, and M

operation Entangle (qAlice : Qubit, qBob : Qubit) : Unit is Adj {
    H(qAlice);
    CNOT(qAlice, qBob);
}

operation SendMsg (qAlice : Qubit, qMsg : Qubit) : (Bool, Bool) {
    CNOT(qMsg, qAlice);
    H(qMsg);
    return (M(qMsg) == One, M(qAlice) == One);
}

operation DecodeMsg (qBob : Qubit, (b1 : Bool, b2 : Bool)) : Unit {
    if b1 { Z(qBob); }
    if b2 { X(qBob); }
}

operation Teleport (qAlice : Qubit, qBob : Qubit, qMsg : Qubit) : Unit {
    Entangle(qAlice, qBob);
    let classicalBits = SendMsg(qAlice, qMsg);
    DecodeMsg(qBob, classicalBits);
}
```
The need to specify Q# formally

Sound language design principles lead to programming languages in which programs are easier to write, compose, and maintain.

Previous examples:
- Standard ML [Harper and Stone 2000]
- Featherweight Java [Igarashi, Pierce, and Wadler 2001]
- Featherweight Go [Griesemer et al. 2020]
- $\lambda_{JS}$ [Guha, Saftoiu, and Krishnamurthi 2009]
- $\lambda_{Rust}$ [Jung et al. 2017]

Q# is a living body of work that will grow and evolve over time.

— Design Principle 5 [Heim 2020, Ch. 8]

Having a well-founded meta-theory of a programming language helps with its evolution.
A recipe for formal language specification

Inside every large language is a small language struggling to get out...
— Tony Hoare

1. Define a well-behaved internal language (core) for Q#: $\lambda_Q$.
2. Define an elaboration relation from the external language to the internal language.
3. Specify static and dynamic semantics using the internal language.
   Statics (type system) rule out meaningless programs.
   Dynamics specify behavior of programs at a high abstraction level.
4. Prove meta-theorems such as type preservation and safety.

Study consequences of extensions and variations.

---

$^1$A Type-Theoretic Interpretation of Standard ML [Harper and Stone 2000]
Unsafe programs in Q#

```qsharp
operation Clone () : Unit {
    use q1 = Qubit();
    let q2 = q1;
    CNOT(q1, q2);
}
```

Using same qubit as both control and target.

```qsharp
operation NewQubit () : Qubit {
    use q = Qubit();
    return q;
}
```

Returning a qubit after its lifetime has ended.
$\lambda_{Q\#}$: a core calculus for Q# (Types)

<table>
<thead>
<tr>
<th>Typ  $\tau$ ::=</th>
<th>qref$\langle q \rangle$</th>
<th>qubit ref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fun($\tau_1 ; \tau_2$)</td>
<td>function</td>
</tr>
<tr>
<td></td>
<td>cmd($\tau$)</td>
<td>command</td>
</tr>
<tr>
<td></td>
<td>prod($i \leftrightarrow \tau_i \mid i \in n$)</td>
<td>product</td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>unit</td>
<td>unit</td>
</tr>
</tbody>
</table>

Key idea: **Qubit** type in Q# $\equiv$ qref$\langle q \rangle$ type of indexed qubit references in $\lambda_{Q\#}$.\(^2\)

---

\(^2\)Alias Types [Smith et al. 2000]
$\lambda_{Q\#}$: Expressions

\[
\text{Exp} \ e \ ::= \ x \quad \text{variable} \\
\text{let}[\tau_1 ; \tau_2](e_1 ; x . e_2) \quad \text{let binding} \\
\lambda[\tau_1 ; \tau_2](x . e) \quad \text{function} \\
\text{ap}[\tau_1 ; \tau_2](e_1 ; e_2) \quad \text{application} \\
\text{cmd}[\tau](m) \quad \text{encapsulated command} \\
\text{tuple}[\tau_n](i \mapsto e_i \mid i \in n) \quad \text{tuple} \\
\text{proj}<i>[\tau_n](e) \quad \text{projection} \\
\text{true} \quad \text{true} \\
\text{false} \quad \text{false} \\
\text{if}[\tau](e ; e_1 ; e_2) \quad \text{if expression} \\
\text{unit} \quad \text{unit}
\]

Essentially, PFPL’s language MA (Modernized Algol) [Harper 2016, Ch. 34]
Key idea: The allocation command, `newqref(x . m)`, comes with its own binding form so that qubits can never escape lexical scope resulting in an Algol-like stack discipline.
Typing of quantum commands

\[ \Gamma \vdash_{\Sigma} m \leadsto \tau \]  
\( m \) is a well-formed command relative to \( \Sigma \), returning a value of type \( \tau \)

**CMD-NEWQREF**
\[
\begin{align*}
\Gamma, x : \textit{qref}\langle q \rangle & \vdash_{\Sigma} q \ m \leadsto \tau \\
\Gamma & \vdash_{\Sigma} \textit{newqref}(x.m) \leadsto \tau
\end{align*}
\]

**CMD-GATEAPREF**
\[
\begin{align*}
\Gamma & \vdash_{\Sigma} e : \textit{prod}\left( i \mapsto \textit{qref}\langle q_i \rangle \right)_{i \in 1..n} \\
\Gamma & \vdash_{\Sigma} \textit{gateap}\langle U^{2n} \rangle(e) \leadsto \textit{unit}
\end{align*}
\]

**CMD-DIAGAPREF**
\[
\begin{align*}
\Gamma & \vdash_{\Sigma} e_1 : \textit{qref}\langle q \rangle \\
\Gamma & \vdash_{\Sigma} e_2 : \textit{prod}\left( i \mapsto \textit{qref}\langle r_i \rangle \right)_{i \in 1..n} \\
\Gamma & \vdash_{\Sigma} \textit{diagap}\langle U^{2n}, V^{2n} \rangle(e_1; e_2) \leadsto \textit{unit}
\end{align*}
\]

**CMD-MEASREF**
\[
\Gamma \vdash_{\Sigma} e : \textit{qref}\langle q \rangle \\
\Gamma \vdash_{\Sigma} \textit{meas}(e) \leadsto \textit{bool}
\]

*Signature, \( \Sigma \), keeps track of qubit symbols in scope, each qubit symbol is distinct.*
Statically preventing cloning

```
operation Clone () : Unit {
    use q1 = Qubit();
    let q2 = q1;
    CNOT(q1, q2);
}
```

In \(\lambda_{Q\#}\) abstract syntax:

```
CNOT \triangleq \begin{array}{c}
\left(
\begin{array}{c}
X
\end{array}
\right)
\equiv \begin{array}{c}
\left(
\begin{array}{c}
I_2
\end{array}
\right)
\begin{array}{c}
X
\end{array}
\end{array}
\end{array}
\end{array}
```

```
newqref(q_1).ret(let(q_1 ; q_2 . cmd(diagap(\{I_2, X\})(q_1 ; q_2))))))
```
Statically preventing cloning

**operation** `Clone()` : `Unit`

```plaintext
use q1 = Qubit();
let q2 = q1;
CNOT(q1, q2);
```

**CMD-DIAGAPREF**

\[
\begin{align*}
\Gamma \vdash \Sigma \quad & e_1 : \text{qref}\langle q \rangle \\
\Gamma \vdash \Sigma \quad & e_2 : \text{prod}(\frac{i \mapsto \text{qref}\langle r_i \rangle}{i \in 1..n}) \\
\Gamma \vdash \Sigma \quad & \text{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1; e_2) \sim \text{unit}
\end{align*}
\]

In $\lambda_{Q\#}$ abstract syntax:

```plaintext
newqref(q1.ret(let(q1; q2.cmd(diagap\langle I_2, X\rangle(q1; q2))))))
```

but $q_1 : \text{qref}\langle q_1 \rangle$ and $q_2 : \text{qref}\langle q_1 \rangle$ have the same type!
Safe qubit management

```
operation NewQubit () : Qubit {
    use q = Qubit();
    return q;
}
```

In $\lambda Q#$ abstract syntax:

$$
\lambda (\_ . \text{cmd(newqref}(x . \text{ret}(x))))
$$
Safe qubit management

```
operation NewQubit () : Qubit {
    use q = Qubit();
    return q;
}
```

\[\text{CMD-NEWQREF} \]
\[
\Gamma, x : qref\langle q\rangle \vdash \Sigma, q . m \leadsto \tau
\]
\[
\Gamma \vdash \Sigma \text{newqref}(x . m) \leadsto \tau
\]

In \(\lambda_{Q\#}\) abstract syntax:

\[
\lambda(_. \text{cmd(newqref}(x . \text{ret}(x))))
\]

\[
\Gamma, x : qref\langle q\rangle \vdash \Sigma, q \text{ret}(x) \leadsto qref\langle q\rangle
\]

but \(\Gamma \not\vdash \Sigma \text{newqref}(x . \text{ret}(x)) \leadsto qref\langle q\rangle\)
Future steps

Solution for no-cloning problem for Q# arrays.

Coverage of other major Q# features.

Mechanized metatheory for the core language.

Semantics preserving compilation to major quantum intermediate languages:

- QIR [Geller 2020] (LLVM-based)
- OpenQASM 3 [Cross et al. 2021]

Integration with existing tools such as Vellvm (verified LLVM).
\(\lambda_{Q\#}\) and Q# are Algol-like quantum languages

Safely combine pure (classical) and effectful (quantum) computation.

Obey strict stack discipline for (qubit) memory management.

In the paper:

More details.

An equational semantics based on Staton’s fully complete equational theory for quantum computation.

Elaboration rules from Q# to \(\lambda_{Q\#}\).

See arXiv:2206.03532 / ks.cs.uchicago.edu/publication/q-algol