Q# as a Quantum Algorithmic Language

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

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

Quantum Physics and Logic 2022

The Q# programming language (Microsoft, 2017–)

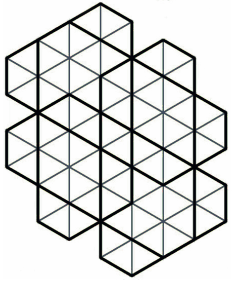
F#-like DSL in the skin of C#-like syntax.[a]

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#

```
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]
- λ_{JS} [Guha, Saftoiu, and Krishnamurthi 2009]; λ_{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[1]

> 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#

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

Using same qubit as both control and target.

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

Returning a qubit after its lifetime has ended.

$\lambda_{Q\#}$: a core calculus for Q# (Types)

Typ	τ	$::=$	$\texttt{qref}\langle q \rangle$	qubit ref
			$\texttt{fun}(\tau_1 \,;\, \tau_2)$	function
			$\texttt{cmd}(\tau)$	command
			$\texttt{prod}(i \hookrightarrow \tau_i \mid i \in n)$	product
			`bool`	boolean
			`unit`	unit

Key idea: **`Qubit`** type in Q# $\equiv \texttt{qref}\langle q \rangle$ type of indexed qubit references in $\lambda_{Q\#}$.[2]

[2] Alias Types [Smith et al. 2000]

$\lambda_{Q\#}$: Expressions

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

Essentially, **PFPL**'s language **MA** (Modernized Algol) [Harper 2016, Ch. 34]

$\lambda_{Q\#}$: Commands

Cmd	m	$::=$	$\texttt{ret}[\tau](e)$	return
			$\texttt{bnd}[\tau_1 ; \tau_2](e ; x \,.\, m)$	bind
			$\texttt{newqref}[\tau](x \,.\, m)$	new qbit ref
			$\texttt{gateap}\langle U_{2^n}\rangle(e)$	gate application
			$\texttt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1 ; e_2)$	controlled gate app.
			$\texttt{meas}(e)$	measure

Key idea: The allocation command, $\texttt{newqref}(x \,.\, m)$, 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

$\boxed{\Gamma \vdash_\Sigma m \div \tau}$ *(m is a well-formed command relative to Σ, returning a value of type τ)*

CMD-NEWQREF

$$\frac{\Gamma, x : \mathbf{qref}\,\langle q\rangle \vdash_{\Sigma,q} m \;\div\; \tau}{\Gamma \vdash_\Sigma \mathbf{newqref}\,(x.m) \;\div\; \tau}$$

CMD-GATEAPREF

$$\frac{\Gamma \vdash_\Sigma e : \mathbf{prod}\,(\,\overline{i \hookrightarrow \mathbf{qref}\,\langle q_i\rangle}^{\,i\in 1..n}\,)}{\Gamma \vdash_\Sigma \mathbf{gateap}\,\langle U_{2^n}\rangle(e) \;\div\; \mathbf{unit}}$$

CMD-DIAGAPREF

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathbf{qref}\,\langle q\rangle \qquad \Gamma \vdash_\Sigma e_2 : \mathbf{prod}\,(\,\overline{i \hookrightarrow \mathbf{qref}\,\langle r_i\rangle}^{\,i\in 1..n}\,)}{\Gamma \vdash_\Sigma \mathbf{diagap}\,\langle U_{2^n}, V_{2^n}\rangle(e_1; e_2) \;\div\; \mathbf{unit}}$$

CMD-MEASREF

$$\frac{\Gamma \vdash_\Sigma e : \mathbf{qref}\,\langle q\rangle}{\Gamma \vdash_\Sigma \mathbf{meas}\,(e) \;\div\; \mathbf{bool}}$$

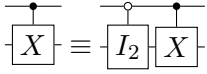
Signature, Σ, keeps track of qubit symbols in scope, each qubit symbol is distinct.

Statically preventing cloning

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

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

$$\texttt{newqref}(q_1\,.\,\texttt{ret}(\texttt{let}(q_1\,;q_2\,.\,\texttt{cmd}(\texttt{diagap}\langle I_2, X\rangle(q_1\,;q_2)))))$$

$CNOT \triangleq$ 

Statically preventing cloning

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

CMD-DIAGAPREF

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathbf{qref}\,\langle q \rangle \qquad \Gamma \vdash_\Sigma e_2 : \mathbf{prod}\,(\overline{i \hookrightarrow \mathbf{qref}\,\langle r_i \rangle}^{\,i \in 1..n})}{\Gamma \vdash_\Sigma \mathbf{diagap}\,\langle U_{2^n}, V_{2^n} \rangle (e_1; e_2) \;\dot{:}\; \mathbf{unit}}$$

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

$$\texttt{newqref}(q_1\,.\,\texttt{ret}(\texttt{let}(q_1\,;q_2\,.\,\texttt{cmd}(\texttt{diagap}\langle I_2, X\rangle(q_1\,;q_2)))))$$

but $q_1 : \texttt{qref}\langle q_1 \rangle$ and $q_2 : \texttt{qref}\langle q_1 \rangle$ have the same type!

Safe qubit management

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

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

$$\lambda(\_\,.\,\texttt{cmd}(\texttt{newqref}(x\,.\,\texttt{ret}(x))))$$

Safe qubit management

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

CMD-NEWQREF

$$\frac{\Gamma, x : \mathbf{qref}\,\langle q\rangle \vdash_{\Sigma,q} m \;\dot{\sim}\; \tau}{\Gamma \vdash_{\Sigma} \mathbf{newqref}\,(x.m) \;\dot{\sim}\; \tau}$$

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

$$\lambda(\_\,.\,\mathtt{cmd}(\mathtt{newqref}(x\,.\,\mathtt{ret}(x))))$$

$\Gamma, x : \mathtt{qref}\langle q\rangle \vdash_{\Sigma,q} \mathtt{ret}(x) \dot{\sim} \mathtt{qref}\langle q\rangle$

but $\Gamma \nvdash_{\Sigma} \mathtt{newqref}(x\,.\,\mathtt{ret}(x)) \dot{\sim} \mathtt{qref}\langle q\rangle$

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