Toward a Type-Theoretic Interpretation of Q#

and Statically Enforcing the No-Cloning Theorem

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

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

Programming Languages for Quantum Computing (PLanQC) 2021

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]

Having a well-founded meta-theory of a programming language helps with its evolution.

> *Q# is a living body of work that will grow and evolve over time.*
> — *Design Principle 5* [Heim 2020, Ch. 8]

The Q# programming language

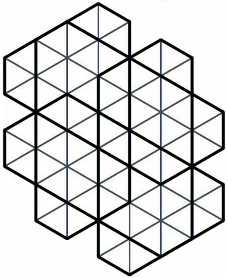
Announced by Microsoft in 2017.

F#-like domain-specific language
in the skin of C#-like syntax.

Running Q# programs:

- standalone (command line)
- Jupyter Notebooks
- host languages: Python or .NET (F#/C#)

Extensive library (chemistry, numerics, machine learning) and learning resources (*quantum katas*).



A tale of two languages: F# & Q# [Azariah 2018]

Q# language model and features

Quantum computer as a co-processor to a classical host (QRAM); computation by side effects.

Quantum operations are a monadic sequence of instructions.

Clean separation between classical (`function`) and quantum (`operation`) callables.

Metaprogramming support using `adjoint` and `controlled` operations.

First class callables, higher-order programming, immutable-by-default.

Teleportation in Q#

```
operation Teleport (msg : Qubit, there : Qubit) : Unit {
    use here = Qubit( );

    // Create an entangled state
    H(here);
    CNOT(here, there);

    // Send the message
    CNOT(msg, here);
    H(msg);

    // Measure out the entanglement
    if (M(msg)  == One) { Z(there); }
    if (M(here) == One) { X(there); }
}
```

A recipe for formal language specification

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.

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

e	$::=$			Expressions	
	$\mid$	x	x		variable
	$\mid$	q	q		(opaque) qubit
	$\mid$	$\mathbf{let}\,(e_1; x.e_2)$	let x be e_1 in e_2		let binding
	$\mid$	$\mathbf{lam}\,\{\tau\}(x.e)$	$\lambda(x : \tau)e$		abstraction
	$\mid$	$\mathbf{ap}\,(e_1; e_2)$	$e_1(e_2)$		application
	$\mid$	$\mathbf{cmd}\,(m)$	cmd m		encapsulation
	$\mid$	$\mathbf{qloc}\,[q]$	&q		qubit reference
	$\mid$	$\mathbf{triv}$	()		unit constant
τ	$::=$			Types	
	$\mid$	$\mathbf{qbit}$	qbit		
	$\mid$	$\mathbf{qref}\,[\kappa]$	qref		
	$\mid$	$\mathbf{arr}\,(\tau_1; \tau_2)$	$\tau_1 \rightarrow \tau_2$		
	$\mid$	$\mathbf{cmd}\,(\tau)$	τ cmd		
	$\mid$	$\mathbf{unit}$	unit		

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

m	$::=$		Commands
	$\mid$	$\textbf{ret}\,(e)$	ret e
	$\mid$	$\textbf{bnd}\,(e; x.m)$	bind $x \leftarrow e; m$
	$\mid$	$\textbf{dcl}\,(q.m)$	dcl q in m
	$\mid$	$\textbf{gateapr}\,(e; U)$	$U\,(e)$
	$\mid$	$\textbf{ctrlapr}\,(e_1; e_2; U)$	Controlled $U\,(e_1, e_2)$

$\lambda_{Q\#}$ maintains a separation between classical and quantum code just like Q#.

The `Qubit` type in Q# corresponds to the qref type of qubit references in $\lambda_{Q\#}$.

Aliasing of qubits can lead to incorrect Q# programs

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

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

The Q# type system currently cannot statically prevent this error.

Can we do better in $\lambda_{Q\#}$?

Statically preventing cloning of qubits

dcl q in

Statically preventing cloning of qubits

$$\begin{aligned} & \text{dcl}\ q\ \text{in} \\ q :\quad \textbf{qbit} \vdash\ & \text{let}\ q_1\ \text{be}\ \&q\ \text{in} \end{aligned}$$

Statically preventing cloning of qubits

$$\begin{array}{rrl} & & \text{dcl } q \text{ in} \\ & q : \mathbf{qbit} \vdash & \text{let } q_1 \text{ be } \&q \text{ in} \\ q_1 : & \mathbf{qref}[\kappa_1], q :^{\dagger\kappa_1} \mathbf{qbit} \vdash & \text{let } q_2 \text{ be } q_1 \text{ in} \end{array}$$

Statically preventing cloning of qubits

$$\begin{aligned}
& \mathsf{dcl}\ q\ \mathsf{in} \\
q : \mathbf{qbit} & \vdash \mathsf{let}\ q_1\ \mathsf{be}\ \&q\ \mathsf{in} \\
q_1 : \mathbf{qref}[\kappa_1], q :^{\dagger\kappa_1} \mathbf{qbit} & \vdash \mathsf{let}\ q_2\ \mathsf{be}\ q_1\ \mathsf{in} \\
q_2 : \mathbf{qref}[\kappa_2], q_1 :^{\dagger\kappa_2} \mathbf{qref}[\kappa_1], q :^{\dagger\kappa_1} \mathbf{qbit} & \nvdash \mathsf{Controlled}\ \mathbf{X}\ (q_1, q_2)
\end{aligned}$$

$\lambda_{\textbf{Rust}}$-like lifetimes and typing

(Coercion for qubit loaning)

COE-LOAN

$$\frac{\mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathbf{L} \vdash \Gamma, x : \mathbf{qref}\,[\kappa] \xRightarrow{ctx} \Gamma, x' : \mathbf{qref}\,[\kappa'], x :^{\dagger\kappa'} \mathbf{qref}[\kappa]}$$

(Select typing rules)

TY-LETLOAN

$$\frac{\Gamma_1 \mid \mathbf{L} \vdash e_1 : \mathbf{qref}\,[\kappa] \dashv x.\Gamma_2 \qquad \Gamma_2, \Gamma \mid \mathbf{L} \vdash e_2 : \tau_2}{\Gamma_1, \Gamma \mid \mathbf{L} \vdash \mathbf{let}\,(e_1; x.e_2) : \tau_2}$$

CMD-CTRLAPREF

$$\frac{\Gamma \mid \mathbf{L} \vdash e_1 : \mathbf{qref}\,[\kappa_1] \qquad \Gamma \mid \mathbf{L} \vdash e_2 : \mathbf{qref}\,[\kappa_2]}{\Gamma \mid \mathbf{L} \vdash \mathbf{ctrlapr}\,(e_1; e_2; U) : \mathbf{unit}}$$

Ongoing work

Generalized product and sum types for encoding `Bool`, `Pauli`, and `Result` types.

Mutable bindings, arrays, and measurement.

Type and lifetime polymorphism.

Explicit treatment of `adjoint` and `controlled` operations for metaprogramming support.

Mechanization of metatheory in Coq using ott and LNgen for a locally-nameless representation.

Future steps

Formalize elaboration from the surface Q# language to $\lambda_{Q\#}$.

Semantics preserving compilation from Q# to next-generation quantum intermediate languages:

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

Integration with existing tools such as Vellvm (verified LLVM).

Conclusion

We presented our ongoing work on formally specifying the Q# programming language.

Our core language, $\lambda_{Q\#}$:

- maintains separation between pure classical and effectful quantum sub-languages.
- treats underlying qubits more explicitly to control aliasing between qubit references.

We proposed a solution to prevent cloning of qubits statically following λ_{Rust}.

We look forward to exciting ongoing and future work ahead.