Toward a Type-Theoretic Interpretation of Q#

and Statically Enforcing the No-Cloning Theorem

KARTIK SINGHAL, University of Chicago, USA
SARAH MARSHALL, Microsoft Quantum, USA
KESHA HIETALA, University of Maryland, USA
ROBERT RAND, University of Chicago, USA

Q# is a high-level programming language from Microsoft for writing and running quantum programs. Like most industrial languages, it was designed without a formal specification, which can naturally lead to ambiguity in its interpretation. Further, currently, the Q# type system cannot statically prevent cloning of qubits. We aim to provide a formal specification and semantics for Q#, placing the language on a solid mathematical foundation, enabling further evolution of its design and type system (including enforcing no-cloning). This paper describes our current progress in designing $\lambda_{Q\#}$ (an idealized version of Q#), our solution to the qubit cloning problem in $\lambda_{Q\#}$, and outlines the next steps.

1 INTRODUCTION

Microsoft's Q# programming language [Svore et al. 2018] is one of the most sophisticated quantum programming languages that have emerged in recent years. With a growing code base and popularity comes the demand for more features and complexity. Hence, Q# faces challenges familiar to many growing programming languages—maintaining correctness, ease of use, and intuitive understanding.

If the Q# programming language is going to have a lasting impact, it will be important to have a well-specified definition that can serve as a foundation for language extensions, multiple implementations, and formal verification of both programs written in the language and its compiler. A formal specification and further mechanization of the language spec will help ensure that Q# is robust enough to meet the unique needs of the developing field of quantum software engineering.

A tried-and-tested approach to achieving this ambitious goal is to define an idealized version of the full language, provide an elaboration from the full language to this core language, and provide statics and dynamics for the core language. In this talk, we argue that even though Q# is a relatively large language, we can condense it to a small core capturing most of its interesting features. This core language, which we call $\lambda_{Q\#}$, is the primary focus of this work. Taking inspiration from Rust [Matsakis and Klock II 2014; Rust Team 2021], we also propose a solution to statically prevent cloning of qubits in $\lambda_{Q\#}$, which we hope will be integrated into Q# itself.

We are encouraged by previous efforts in formal specification of relatively large programming languages such as Standard ML [Harper and Stone 2000; Lee, Crary, and Harper 2007], Java [Igarashi, Pierce, and Wadler 2001], JavaScript [Guha, Saftoiu, and Krishnamurthi 2010], Rust [Jung 2020; Jung et al. 2017], and, most recently, Go [Griesemer et al. 2020].

2 THE Q# PROGRAMMING LANGUAGE

Q# is a hybrid quantum-classical programming language that supports interlacing stateful quantum *operations* with pure classical *functions*, collectively referred to as *callables*. Q# encourages thinking about quantum programs as algorithms instead of as circuits, where quantum operations can be combined with classical control flow such as branches and loops. When qubits are measured,

Authors' addresses: Kartik Singhal, ks@cs.uchicago.edu, Department of Computer Science, University of Chicago, USA; Sarah Marshall, samarsha@microsoft.com, Microsoft Quantum, Redmond, USA; Kesha Hietala, kesha@cs.umd.edu, Department of Computer Science, University of Maryland, College Park, USA; Robert Rand, rand@uchicago.edu, Department of Computer Science, University of Chicago, USA.

arbitrary classical computation can be performed on the measurement results and the program execution can continue without requiring any qubits to be released. This computational model is what allows quantum and classical algorithms to be fully mixed. At the same time, Q# enforces a degree of separation between the quantum and classical components of a program. Operations can apply functions, but functions are forbidden from applying operations.

Callables in Q# can be *higher-order*: functions and operations are values and can be given as arguments to, or returned by, other functions and operations. Both functions and operations can be partially applied. Quantum algorithms that are parameterized by quantum subroutines are easily expressed in Q# using higher-order operations. For example, an operation implementing Grover's search can accept the oracle as a parameter and apply it in each iteration.

Q# supports a restricted form of metaprogramming, where the compiler can automatically generate the adjoint (`Adjoint U`) and controlled (`Controlled U`) versions of some unitary operation, `U`. For example, the adjoint can be generated by replacing each operation applied by `U` with its adjoint and reversing the order, assuming every operation applied by `U` is adjointable.

Qubits in Q# are opaque types that act as references to logical qubits [Geller 2018]—their values are never exposed. Gate operations are inherently *effectful*: a (single-qubit) quantum gate application is a procedure that takes a qubit reference as input, and returns a trivial output of type `Unit`; the quantum state is altered by the operation. Passing references to a function effectively creates an *alias*. Arrays of qubits are another common scenario where items in the array are aliased during iteration in a `for` loop.

While aliasing is necessary for gate application in Q#, it can lead to unsafe behavior in violation of the *no-cloning theorem* [Wootters and Zurek 1982] which forbids duplication of qubits. In Listing 1, `q1` and `q2` both refer to the same qubit. Applying `CNOT` with `q1` as the control and `q2` as the target is equivalent to cloning the underlying qubit. Currently, it is not possible for Q# to statically prevent this issue.

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

Listing 1. Aliasing of qubits can lead to incorrect Q# programs.

An informal specification of the Q# language was recently published [Heim 2020; Heim and Q# Team 2020], but it arguably does not capture all subtle aspects of the language such as aliasing of qubit references. The goal of our work is to make these subtleties explicit and formal, and one of the first applications is to include a static check to prevent cloning.

3 $\lambda_{Q\#}$: A CORE CALCULUS FOR Q#

Our approach closely follows the type-theoretic interpretation of Standard ML [Harper and Stone 2000] where a well-typed internal language for Standard ML was developed, an elaboration relation between the external language and this internal language was defined, and properties of the metatheory of the language were proven using the internal language. This work was followed by the mechanization of the metatheory [Crary and Harper 2009; Lee, Crary, and Harper 2007] using the Twelf logical framework [Pfenning and Schürmann 1999]. As a first step, we identify and isolate the core language, $\lambda_{Q\#}$, that captures the essential aspects of Q#. This core language is explicitly typed and the safety properties of its type structure can be easily stated and proved.

Once we have identified the core, the overall strategy is to define an elaboration relation from the surface-level Q# language to $\lambda_{Q\#}$. The process of elaboration performs several functions: scope resolution by identifying the binding and scope of identifiers in Q#; type inference; and management of derived forms. A Q# program is well-formed when it has a well-typed elaboration

τ	::=		Types
	|	**qbit**	qbit
	|	**qref** $[\kappa]$	qref
	|	**arr** $(\tau_1; \tau_2)$	$\tau_1 \to \tau_2$
	|	**cmd** (τ)	τ cmd
	|	**unit**	unit

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

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

Fig. 1. $\lambda_{Q\#}$ grammar

and its semantics is defined to be that of its elaboration. The advantage of this approach is that proving properties about the metatheory of a large language becomes quite scalable because one needs to do it only for the small, well-formed core.

To mirror the separation between operations and functions in Q#, we base the design of $\lambda_{Q\#}$ on **MA** (Modernized Algol) [Harper 2016a], which maintains a separation between commands that modify state and expressions that do not.[1]

Before presenting $\lambda_{Q\#}$, let us motivate our design choices and establish some terminology. Variable bindings in Q# are divided into two kinds: those defined using the **`let`** keyword are the same as the variables in **MA** and follow the usual substitution-based semantics of functional programming languages; those defined using the **`mutable`** keyword correspond to *assignables* that can be reassigned similar to "variables" in imperative languages (we will ignore **`mutable`**s/assignables in the rest of this paper). Qubits in Q# have the **`Qubit`** type and syntactically look just like other variables, but are actually references to underlying qubits that are never exposed. We can think of qubits as indices into a global quantum register. We will model their values as *locations* in this work. Another distinction is that aliasing is allowed for qubits (unlike classical variables and assignables) which can lead to problems such as the violation of the no-cloning theorem that we discussed earlier. The only allowed operations on qubits are gate application and measurement. Qubits come into scope with either the **`use`** or the **`borrow`** keywords. The former provides access to freshly allocated qubits, while the latter provides access to (potentially entangled) qubits that may be previously allocated.

In the $\lambda_{Q\#}$ grammar in Figure 1, the binding structure of the syntax is precisely specified following the notion of abstract binding trees [Harper 2016b, Chapter 1]. We show the convenience syntax on the right in blue color along with a description of productions where unclear. We ignore the pure expression language by not including any classical base types except **unit**. This way we can focus on the interesting quantum-classical interface at play in Q#. The qubit reference type **qref**$[\kappa]$ is inhabited by qubit locations **qloc**[q] that serve as its values and can be compared for equality. The green colored symbol, κ, is used to associate a *lifetime* with each qubit reference;

[1] Another quantum language, IQu [Paolini, Roversi, and Zorzi 2019], takes a similar route by extending Idealized Algol (IA) [Reynolds 1981], but neither IA nor IQu maintains such separation.

$\boxed{\mathbf{L} \vdash \Gamma_1 \overset{ctx}{\Longrightarrow} \Gamma_2}$ *(Type Coercions)*

$$\textsc{coe-Use}\quad \frac{}{\mathbf{L} \vdash \Gamma, q : \mathbf{qbit} \overset{ctx}{\Longrightarrow} \Gamma, x : \mathbf{qref}\ [\kappa], q :^{\dagger\kappa} \mathbf{qbit}}$$

$$\textsc{coe-Loan}\quad \frac{\mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathbf{L} \vdash \Gamma, x : \mathbf{qref}\ [\kappa] \overset{ctx}{\Longrightarrow} \Gamma, x' : \mathbf{qref}\ [\kappa'], x :^{\dagger\kappa'} \mathbf{qref}[\kappa]}$$

Fig. 2. Type coercions for qubit and qubit reference types

we will say more in the next section. Qubit symbols are shown in orange color to distinguish them from other variables, we use them to model the underlying qubit that is not exposed in the surface Q# language. Single-qubit (intrinsic) unitaries, U, are typed as $\mathbf{qref}[\kappa] \to \mathbf{cmd}(\mathbf{unit})$. The $\mathbf{gateapr}(e; U)$ command applies the given operation to a qubit reference and $\mathbf{ctrlapr}(e_1; e_2; U)$ applies U to its second argument controlled by its first argument.

In $\lambda_{Q\#}$ syntax, the unsafe code fragment shown in Listing 1 can be written as:

$$\mathbf{dcl}(q.\mathbf{let}(\mathbf{qloc}[q]; q_1.\mathbf{let}(q_1; q_2.\mathbf{ctrlapr}(q_1; q_2; \mathbf{X}))))$$

or with some syntactic sugar as:

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

which makes it explicit that q_1 and q_2 are aliases of the same logical qubit.

4 STATICALLY PREVENTING CLONING

In this section, we demonstrate how we can statically track aliasing in $\lambda_{Q\#}$ so that the code in the previous section will not type check. We take inspiration from the type system of λ_{Rust} [Jung 2020; Jung et al. 2017], which provides a concise type-theoretical model for the Rust borrow checker, and similarly introduce the notion of lifetimes in $\lambda_{Q\#}$. Like λ_{Rust}, our typing context, Γ, is substructural, and we use an additional lifetime context, $\mathbf{L}$, for tracking lifetime inclusion relationships.

Each qubit reference is typed as $\mathbf{qref}[\kappa]$, where κ denotes a particular lifetime variable associated with the reference. When a qubit reference is first introduced (we call this *using the qubit*), which corresponds to the **use** keyword in Q# and $\mathbf{dcl}(q.\mathbf{let}(\mathbf{qloc}[q]; x.(\ldots)))$ in $\lambda_{Q\#}$, the type system ensures that q can never be used again with the type coercion rule COE-USE. This rule introduces a new lifetime variable κ, creates a new reference with that lifetime, and blocks the type of the qubit until the lifetime of the reference ends. (The λ_{Rust}-inspired notation $x :^{\dagger\kappa} \tau$ says that the type of x is blocked until the end of lifetime κ.) This newly created reference immediately gets rebound to the fresh variable introduced in the **let** expression (c.f. type rule TY-LETLOAN) so that the only type relation for q in the resulting context is $q :^{\dagger\kappa} \mathbf{qbit}$.

Similarly, when a qubit reference is aliased (which we call *loaning the qubit*), the coercion rule COE-LOAN introduces a new qubit reference of type $\mathbf{qref}[\kappa']$ with a different lifetime, where

$$\boxed{\Gamma \mid \mathbf{L} \vdash m : \tau}, \quad \boxed{\Gamma \mid \mathbf{L} \vdash e : \tau}, \quad \boxed{\Gamma_1 \mid \mathbf{L} \vdash e : \tau \dashv x.\Gamma_2} \qquad \textit{(Select typing rules)}$$

$$\textsc{cmd-Dcl}\ \frac{\Gamma, q : \mathbf{qbit} \mid \mathbf{L} \vdash m : \tau}{\Gamma \mid \mathbf{L} \vdash \mathbf{dcl}\,(q.m) : \tau} \qquad \textsc{ty-LetLoan}\ \frac{\Gamma_1 \mid \mathbf{L} \vdash e_1 : \mathbf{qref}\,[\kappa] \dashv x.\Gamma_2 \quad \Gamma_2, \Gamma \mid \mathbf{L} \vdash e_2 : \tau_2}{\Gamma_1, \Gamma \mid \mathbf{L} \vdash \mathbf{let}\,(e_1; x.e_2) : \tau_2}$$

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

Fig. 3. Select typing rules

the type system ensures that the two lifetimes follow a lifetime inclusion relationship $\kappa' \sqsubseteq \kappa$. Further, the type of the original qubit reference is blocked until the end of the lifetime κ', i.e., $x :^{\dagger\kappa'} \mathbf{qref}[\kappa]$.

Essentially, the type system enforces the invariant that at any point in a program, there can only be a single unique reference to each logical qubit.

In Figure 3, we show select typing rules that let us show that our example program fails type checking. Notice that the rule TY-LETLOAN uses the special judgment, $\Gamma \mid \mathbf{L} \vdash e : \tau, \Gamma_1 \mid \mathbf{L} \vdash e : \tau \dashv x.\Gamma_2$, which returns a modified typing context (using the coercion rules of Figure 2) demonstrating the substructural nature of this rule.

Type checking the controlled gate application using rule CMD-CTRLAPREF requires that its first two arguments type check to qubit references. But at that point in the program, after the two applications of rule TY-LETLOAN and the two coercion rules, our typing context Γ is:

$$q :^{\dagger\kappa_1} \mathbf{qbit}, q_1 :^{\dagger\kappa_2} \mathbf{qref}[\kappa_1], q_2 : \mathbf{qref}[\kappa_2]$$

ensuring that the type rule CMD-CTRLAPREF cannot proceed since the type of q_1 is blocked until the end of lifetime κ_2 associated with the qubit reference q_2.

5 CONCLUSION AND PERSPECTIVES

We present our ongoing work on defining a core calculus for the Q# programming language, dubbed $\lambda_{Q\#}$. We maintain a separation between the quantum effectful portion of the language, make aliasing explicit in $\lambda_{Q\#}$, and propose a solution for statically preventing cloning of qubits.

Q# has limited support for parametric polymorphism that we have not tried to tackle yet, but $\lambda_{Q\#}$ is close enough to System F (polymorphic typed λ-calculus) that we expect it to be straightforward. Sum types are a missing feature in Q#; we are considering adding them in $\lambda_{Q\#}$ to easily encode features such as the `Bool` type, the `Pauli` and `Result` types, and features being considered for addition to Q# such as option types and algebraic data types. Other interesting features that we plan to add to $\lambda_{Q\#}$ include mutable bindings, borrowed qubits, arrays, metaprogramming using Q#'s `Adjoint` and `Controlled` constructs, and measurement.

In parallel, we are working on the mechanization of metatheory of $\lambda_{Q\#}$. After identifying and mechanizing a fairly complete core, we have several future directions to work on. One obvious step is to formally define an elaboration relation between the surface-level Q# language and $\lambda_{Q\#}$, which will lead to a complete formal specification of Q# in the style of Lee, Crary, and Harper [2007].

From a verification perspective, we plan to explore semantics-preserving compilation from Q# to the recently announced QIR [Geller 2020], a quantum intermediate representation based on the popular LLVM framework. This will also require formally specifying the semantics of QIR, for which we will draw upon the Verified LLVM (Vellvm) project [Zhao et al. 2012]. We also aim to formalize QIR's *profiles*, which specify what kinds of quantum operations are allowed on a given quantum architecture.

ACKNOWLEDGMENTS

We would like to thank Bettina Heim and Alan Geller for lending us their expertise in the design of Q# and helping to get this project off the ground. Thanks to Bob Harper for helpful conversations about the mechanized definition of Standard ML. This material is based upon work supported by EPiQC, an NSF Expedition in Computing, under Grant No. 1730449 and the Air Force Office of Scientific Research under Grant No. FA95502110051.

REFERENCES

Karl Crary and Robert Harper. 2009. *Mechanization of Type Safety of Standard ML.* https://github.com/SMLFamily/The-Mechanization-of-Standard-ML

Alan Geller. 2018. *Qubits in Q#.* Q# Blog. https://devblogs.microsoft.com/qsharp/qubits-in-qsharp/

Alan Geller. 2020. *Introducing Quantum Intermediate Representation (QIR).* Q# Blog. https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/

Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (2020), 29 pages. https://doi.org/10.1145/3428217

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP '10.* Springer, Berlin, Heidelberg, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7 arXiv:1510.00925

Robert Harper. 2016a. *Practical Foundations for Programming Languages* (2 ed.). Cambridge University Press, Cambridge, UK, Chapter 34 - Modernized Algol, 301–312. https://doi.org/10.1017/CBO9781316576892.036

Robert Harper. 2016b. *Practical Foundations for Programming Languages* (2 ed.). Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/CBO9781316576892

Robert Harper and Chris Stone. 2000. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner.* MIT Press, Cambridge, MA, 341–387. https://www.cs.cmu.edu/~rwh/papers/ttisml/ttisml.pdf

Bettina Heim. 2020. *Development of Quantum Applications.* Ph.D. Dissertation. ETH Zurich, Zurich. https://doi.org/10.3929/ethz-b-000468201

Bettina Heim and Q# Team. 2020. Q# Language Specification. https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language.* Ph.D. Dissertation. Saarland University. https://doi.org/10.22028/D291-31946

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 (2017), 34 pages. https://doi.org/10.1145/3158154

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proc. POPL '07.* ACM, New York, NY, USA, 173–184. https://doi.org/10.1145/1190216.1190245

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

Luca Paolini, Luca Roversi, and Margherita Zorzi. 2019. Quantum Programming Made Easy. In *Proc. Linearity-TLLA 2018.* Open Publishing Association, Waterloo, NSW, Australia, 133–147. https://doi.org/10.4204/eptcs.292.8

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In *Automated Deduction—CADE-16.* Springer, Berlin, Heidelberg, 202–206. https://doi.org/10.1007/3-540-48660-7_14

John C. Reynolds. 1981. The Essence of Algol. In *Proceedings of the International Symposium on Algorithmic Languages.* North-Holland, Amsterdam, 345–372. http://www.cs.cmu.edu/afs/cs/user/crary/www/819-f09/Reynolds81.ps

The Rust Team. 2021. *Rust Programming Language.* https://www.rust-lang.org/

Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher 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. RWDLS '18.* ACM, New York, NY, USA, 7:1–7:10. https://doi.org/10.1145/3183895.3183901 arXiv:1803.00652

W. K. Wootters and W. H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803. https://doi.org/10.1038/299802a0

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. POPL '12.* ACM, New York, NY, 427–440. https://doi.org/10.1145/2103656.2103709