Toward Formalizing the Q# Programming Language
Poster Abstract

Sarah Marshall
Microsoft Quantum
samarsha@microsoft.com

Kartik Singhal
University of Chicago
ks@cs.uchicago.edu

Kesha Hietala
University of Maryland
kesha@cs.umd.edu

Robert Rand
University of Chicago
rand@uchicago.edu

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. 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, and leading to research in program correctness and verified compiler implementation. This poster describes our current progress and outlines the next steps.

1 Introduction

Microsoft's Q# [Svore et al. 2018] programming language 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 and the compiler. This 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 and provide an elaboration from the full language to this core language. In this poster, 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 focus of this work.

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, 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.

Q# supports a restricted form of metaprogramming, where the compiler can automatically generate the adjoint and controlled versions of some unitary operations. For example, given an operation U, `Adjoint U` 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.

Preliminary Report. Final version to appear in:
QPL 2021

Qubits in Q# are opaque types that act as references to logical qubits. Gate operations are inherently *effectful*: a quantum gate application is like a function that takes a qubit reference as input, and returns a trivial output of type `Unit`; the referenced qubit 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* which forbids duplication of qubits. For example, the following Q# code fragment demonstrates incorrect use of aliasing, since the `CNOT` operation cannot be applied with the same qubit as both the control and the target. Currently, it is not possible for Q# to statically prevent this issue.

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

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. One goal of our work is to make these subtleties explicit 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 effort behind the formal (re-)definition 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 was followed by the mechanization of its metatheory [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 $\lambda_{Q\#}$. A Q# program is well-formed when it has a well-typed elaboration 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.

We present a core language for Q#, based on **MA** (Modernized Algol) [Harper 2016], that maintains a separation between commands that modify state and ones that do not. Note that this is different from the approach taken in IQu [Paolini, Roversi, and Zorzi 2019] based on Idealized Algol [Reynolds 1981].

Q# is interesting in that it avoids exposing quantum states, but still provides a way to refer to qubits using its `Qubit` type. The classical bindings in Q# are divided into two kinds: those defined using the `let` keyword are the same as the variables in **MA** that follow the usual substitution-based semantics of functional programming languages; those defined using the `mutable` keyword correspond to *assignables* that can be reassigned using mutation similar to "variables" in imperative languages (we will ignore mutable assignables in the rest of this abstract). Qubits in Q#, even though they look just like another kind of variables, are actually references to an abstract Qubit type—their values are never exposed. We can think of them 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 the 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 first provides access to fresh qubits, while the second may provide mutated and potentially entangled qubits.

Note that in the $\lambda_{Q\#}$ grammar below, we ignore the pure expression language by not including any classical base types except **unit**. This way we can expose the interesting quantum-classical interface at play in Q#. Single-qubit unitary operations, U, are typed as $\mathbf{qref} \to \mathbf{cmd}$. The qubit reference type **qref** is inhabited by qubit locations $\mathbf{qloc}[q]$ that serve as its values and can be compared for equality.

			Types
τ	::=		
	|	**qref**	qubit reference
	|	$\mathbf{arr}(\tau_1;\tau_2)$	function
	|	**cmd**	command
	|	**unit**	unit

			Commands
m	::=		
	|	$\mathbf{ret}(e)$	return
	|	$\mathbf{bnd}(e;x.m)$	sequence
	|	**newqref**	qubit creation
	|	$\mathbf{gateap}(e;U)$	gate application
	|	$\mathbf{ctrlap}(e_1;e_2;U)$	controlled gate app.

			Expressions
e	::=		
	|	x	variable
	|	$\mathbf{let}(e_1;x.e_2)$	let binding
	|	$\mathbf{lam}\{\tau\}(x.e)$	abstraction
	|	$\mathbf{ap}(e_1;e_2)$	application
	|	$\mathbf{cmd}(m)$	encapsulation
	|	$\mathbf{qloc}[q]$	qubit location
	|	**triv**	unit constant

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

$$\mathbf{bnd}(\mathbf{cmd}(\mathbf{newqref});q_1.\mathbf{let}(q_1;q_2.\mathbf{ctrlap}(q_1;q_2;X)))$$

or with some syntactic sugar as $q_1 \leftarrow \mathbf{newqref}$; $\mathbf{let}\ q_2\ \mathbf{be}\ q_1\ \mathbf{in}\ \mathbf{ctrlap}(q_1;q_2;X)$. This makes it explicit that q_1 and q_2 are aliases of the same logical qubit. Extending the type system of $\lambda_{Q\#}$ (and later of Q#) with affine types (à la Rust [Matsakis and Klock II 2014]) will make it possible to statically prevent cloning in the gate application.

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 this to be straightforward.

4 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 and make aliasing explicit in $\lambda_{Q\#}$, but several interesting features remain to be added, including parametric polymorphism, borrowed qubits, arrays, metaprogramming using Q#'s adjoint and controlled constructs, and measurement.

After identifying a fairly complete core in $\lambda_{Q\#}$, we have several future directions to work on. One immediate step is to formally define an elaboration relation between the surface-level Q# language and $\lambda_{Q\#}$. In parallel, we plan to take up the mechanization of the metatheory of $\lambda_{Q\#}$. These two steps together 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] 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.

References

Alan Geller (Sept. 23, 2020). *Introducing Quantum Intermediate Representation (QIR)*. Q# Blog. URL: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/.

Robert Harper (2016). "Practical Foundations for Programming Languages". In: 2nd ed. Cambridge, UK: Cambridge University Press. Chap. 34 - Modernized Algol, pp. 301–312. DOI: 10.1017/CBO9781316576892.036.

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

Bettina Heim (2020). "Development of Quantum Applications". PhD thesis. Zurich: ETH Zurich. Chap. 8: "Domain-Specific Language Q#". DOI: 10.3929/ethz-b-000468201.

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

Daniel K. Lee, Karl Crary, and Robert Harper (2007). "Towards a Mechanized Metatheory of Standard ML". In: *Proc. POPL '07*. New York, NY, USA: ACM, pp. 173–184. DOI: 10.1145/1190216.1190245. URL: https://www.cs.cmu.edu/~dklee/papers/tslf-popl.pdf.

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)*. New York, NY, USA: ACM, pp. 103–104. DOI: 10.1145/2663171.2663188.

Luca Paolini, Luca Roversi, and Margherita Zorzi (2019). "Quantum Programming Made Easy". In: *Proc. Linearity-TLLA 2018*. Waterloo, NSW, Australia: Open Publishing Association, pp. 133–147. DOI: 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*. Berlin, Heidelberg: Springer, pp. 202–206. DOI: 10.1007/3-540-48660-7_14. URL: https://www.cs.cmu.edu/~fp/papers/cade99.pdf.

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

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*. New York, NY, USA: ACM, 7:1–7:10. DOI: 10.1145/3183895.3183901. arXiv: 1803.00652.

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*. New York, NY: ACM, pp. 427–440. DOI: 10.1145/2103656.2103709. URL: https://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf.